

ОГЛАВЛЕНИЕ

Предисловие	12
Глава 1. Введение: распределенные системы	14
1.1. Что такое распределенная система?	15
1.1.1. Мотивировка	15
1.1.2. Вычислительные сети	17
1.1.3. Глобальные сети	19
1.1.4. Локальные сети	21
1.1.5. Многопроцессорные компьютеры	24
1.1.6. Взаимодействующие процессы	27
1.2. Архитектура и языки.	31
1.2.1. Архитектура	31
1.2.2. Эталонная модель OSI	33
1.2.3. Модель OSI для локальных сетей: стандарт IEEE	36
1.2.4. Языковая поддержка.	37
1.3. Распределенные алгоритмы	40
1.3.1. Сравнение распределенных и централизованных алгоритмов.	40
1.3.2. Пример связи с передачей одного сообщения.	42
1.3.3. Область исследований.	48
1.4. Обзор содержания книги	49
Глава 2. Модель	54
2.1. Системы переходов и алгоритмы	55
2.1.1. Системы переходов.	56
2.1.2. Системы с асинхронным обменом сообщениями	56
2.1.3. Системы с синхронным обменом сообщениями.	58
2.1.4. Справедливость	60
2.2. Как обосновывать свойства систем переходов	61
2.2.1. Свойства безопасности	62
2.2.2. Свойства живости	64
2.3. Причинно-следственный порядок событий и логические часы	66
2.3.1. Зависимые и независимые события	67
2.3.2. Эквивалентность выполнений: вычисления	69
2.3.3. Логические часы	74
2.4. Дополнительные допущения. Сложность.	76
2.4.1. Топология сети.	77
2.4.2. Свойства каналов.	79
2.4.3. Допущения реального времени	81
2.4.4. Осведомленность процессов	81
2.4.5. Сложность распределенных алгоритмов	82
2.5. Упражнения к главе 2.	84

Глава 3. Коммуникационные протоколы	86
3.1. Симметричный протокол раздвижного окна	88
3.1.1. Описание протокола	89
3.1.2. Доказательство корректности протокола	92
3.1.3. Обсуждение протокола	94
3.2. Протокол с таймерами	98
3.2.1. Описание протокола	100
3.2.2. Доказательство корректности протокола	105
3.2.3. Обсуждение протокола	111
3.3. Упражнения к главе 3	114
Глава 4. Алгоритмы маршрутизации	116
4.1. Маршрутизация на основе узлов-адресатов	118
4.2. Задача построения кратчайших путей для всех пар вершин	123
4.2.1. Алгоритм Флойда—Уоршалла	123
4.2.2. Алгоритм Туэга построения кратчайших путей	126
4.2.3. Другие алгоритмы	132
4.3. Алгоритм Netchange	137
4.3.1. Описание алгоритма	137
4.3.2. Корректность алгоритма Netchange	143
4.3.3. Обсуждение алгоритма	145
4.4. Маршрутизация с использованием компактных таблиц	147
4.4.1. Схема древесной разметки	148
4.4.2. Интервальная маршрутизация	151
4.4.3. Префиксная маршрутизация	160
4.5. Иерархическая маршрутизация	164
4.5.1. Сокращение числа узлов, в которых выбирается маршрут	165
4.6. Упражнения к главе 4	167
Глава 5. Неблокируемая коммутация пакетов	170
5.1. Введение	171
5.2. Структурированные решения	173
5.2.1. Буферные графы	173
5.2.2. Ориентации сети G	178
5.3. Неструктурированные решения	182
5.3.1. Контроллеры с прямым и обратным отсчетом	182
5.3.2. Контроллеры с состояниями прямого и обратного отсчета	183
5.4. Прочие вопросы	186
5.4.1. Изменения топологии	187
5.4.2. Другие типы блокировок	188
5.4.3. Активный тупик	189
5.5. Упражнения к главе 5	192
Глава 6. Волновые алгоритмы и алгоритмы обхода	193
6.1. Определение и применение волновых алгоритмов	194
6.1.1. Определение волновых алгоритмов	194
6.1.2. Основные свойства волновых алгоритмов	196
6.1.3. Распространение информации с обратной связью	198

6.1.4. Синхронизация	199
6.1.5. Вычисление функций точной нижней грани	200
6.2. Семейство волновых алгоритмов	202
6.2.1. Кольцевой алгоритм	202
6.2.2. Древесный алгоритм	203
6.2.3. Алгоритм эха	206
6.2.4. Алгоритм опроса	208
6.2.5. Фазовый алгоритм	209
6.2.6. Алгоритм Финна	212
6.3. Алгоритмы обхода	214
6.3.1. Обход клик	216
6.3.2. Обход торов	216
6.3.3. Обход гиперкубов	217
6.3.4. Обход связных сетей	218
6.4. Сложность по времени: поиск в глубину	220
6.4.1. Распределенный поиск в глубину	222
6.4.2. Алгоритмы поиска в глубину, требующие линейного времени	223
6.4.3. Поиск в глубину с учетом сведений о соседях	228
6.5. Прочие вопросы	229
6.5.1. Обзор волновых алгоритмов	229
6.5.2. Вычисление сумм	230
6.5.3. Альтернативные определения сложности по времени	233
6.6. Упражнения к главе 6	236
Глава 7. Алгоритмы избрания лидера	239
7.1. Введение	239
7.1.1. Допущения, которых придерживаются в этой главе	240
7.1.2. Выборы и волны	241
7.2. Кольцевые сети	244
7.2.1. Алгоритмы Ле—Ланна и Ченя—Робертса	245
7.2.2. Алгоритм Петерсона/Долева—Клейва—Роде	249
7.2.3. Результат о нижних оценках	253
7.3. Произвольные сети	258
7.3.1. Эффект угасания и быстрый алгоритм	259
7.3.2. Алгоритм Галладжера—Хамблета—Спиры	261
7.3.3. Глобальное описание алгоритма GHS	263
7.3.4. Подробное описание алгоритма GHS	265
7.3.5. Обсуждение GHS-алгоритма и его модификаций	271
7.4. Алгоритм Корача—Каттена—Морана	272
7.4.1. Модульная конструкция	273
7.4.2. Приложения алгоритма Корача—Каттена—Морана	278
7.5. Упражнения к главе 7	279
Глава 8. Обнаружение завершения	281
8.1. Основные понятия	282
8.1.1. Определения	282
8.1.2. Две нижние оценки	286
8.1.3. Как остановить процессы	288
8.2. Деревья и леса вычислений	289

8.2.1.	Алгоритм Дейкстры—Шолтена	289
8.2.2.	Алгоритм Шави—Франчеца	294
8.3.	Решения на основе волновых алгоритмов	299
8.3.1.	Алгоритм Дейкстры—Фейджена—ван Гастерена	300
8.3.2.	Подсчет базовых сообщений: алгоритм Сафры	304
8.3.3.	Использование подтверждений	309
8.3.4.	Обнаружение завершения вычислений при помощи волн	312
8.4.	Прочие решения	314
8.4.1.	Алгоритм возвращения кредита	314
8.4.2.	Решения, использующие штампель времени	317
8.5.	Упражнения к главе 8	320
Глава 9.	Анонимные сети	322
9.1.	Основные понятия	323
9.1.1.	Определения	324
9.1.2.	Классификация вероятностных алгоритмов	327
9.1.3.	Задачи, рассматриваемые в этой главе	329
9.1.4.	Синхронный и асинхронный обмен сообщениями	331
9.2.	Детерминированные алгоритмы	331
9.2.1.	Детерминированные выборы: отрицательные результаты	332
9.2.2.	Вычисление функций на кольце	333
9.3.	Вероятностный алгоритм избрания лидера	338
9.4.	Вычисление размера сети	342
9.4.1.	Отрицательные результаты	342
9.4.2.	Алгоритм вычисления размера кольца	345
9.5.	Упражнения к главе 9	348
Глава 10.	Моментальные состояния системы	351
10.1.	Начальные сведения	352
10.2.	Два алгоритма построения моментального состояния	357
10.2.1.	Алгоритм Ченди—Лэмпорта	357
10.2.2.	Алгоритм Лая—Янга	359
10.3.	Применение алгоритмов построения моментальных состояний	361
10.3.1.	Вычисление состояния канала	361
10.3.2.	Своевременность моментальных состояний системы	362
10.3.3.	Обнаружение устойчивых свойств	363
10.4.	Приложения: обнаружение тупиков	366
10.4.1.	Модель базового вычисления и постановка задачи	366
10.4.2.	Алгоритм глобальной разметки	368
10.4.3.	Обнаружение тупиков в ограниченных моделях	371
10.5.	Упражнения к главе 10	372
Глава 11.	Восприятие направления и ориентация	374
11.1.	Введение и определения	375
11.1.1.	Определения и характеристические свойства восприятия направления	375
11.1.2.	Применение восприятия направления	379
11.1.3.	Широковещательная рассылка сообщений при наличии восприятия направления	380

11.2.	Выборы на кольцах и хордовых кольцах	383
11.2.1.	Алгоритм Франклина	383
11.2.2.	Усовершенствование Аттьи	384
11.2.3.	Минимизация числа хорд	386
11.2.4.	Однохордовый линейный алгоритм	389
11.3.	Вычисления в гиперкубах	393
11.3.1.	Базис: отсутствие осведомленности о топологии сети	394
11.3.2.	Алгоритм посредничества	394
11.3.3.	Алгоритм потока по многим путям	397
11.3.4.	Эффективные алгоритмы для гиперкубов на основе трафаретов	401
11.3.5.	Выборы в неразмеченных гиперкубах	404
11.4.	Сложностные аспекты	405
11.4.1.	Ориентация клики или произвольного графа	405
11.4.2.	Битовая сложность и алгоритм потока по многим путям	407
11.4.3.	Алгоритм случайного блуждания Вервейджа	409
11.5.	Заклучение и открытые вопросы	411
11.5.1.	Использование восприятия направления	411
11.5.2.	Понижение сложности	412
11.5.3.	Современные исследования	413
11.6.	Упражнения к главе 11	413
Глава 12.	Синхронизация в сетях	415
12.1.	Начальные сведения	416
12.1.1.	Синхронные сети	416
12.1.2.	Повышение эффективности за счет синхронизации	418
12.1.3.	Асинхронные сети с ограниченной задержкой	419
12.2.	Выборы в синхронных сетях	423
12.2.1.	Размер сети известен	424
12.2.2.	Размер сети неизвестен	425
12.2.3.	Дополнительные результаты	426
12.3.	Алгоритмы-синхронизаторы	427
12.3.1.	Простой синхронизатор	428
12.3.2.	α , β , и γ -синхронизаторы	430
12.4.	Приложения: поиск в ширину	433
12.4.1.	Синхронный BFS-алгоритм	434
12.4.2.	Соединение с синхронизатором	435
12.4.3.	Асинхронные BFS-алгоритмы	435
12.5.	Допущение Архимеда	439
12.6.	Упражнения к главе 12	441
Глава 13.	Отказоустойчивость в распределенных системах	444
13.1.	Зачем нужны отказоустойчивые алгоритмы	444
13.2.	Робастные алгоритмы	446
13.2.1.	Модели неисправностей	446
13.2.2.	Проблемы принятия решения	448
13.2.3.	Краткое содержание глав 14–16	450
13.2.4.	Темы, которые не были включены в эту книгу	452
13.3.	Стабилизирующиеся алгоритмы	453

Глава 14. Отказоустойчивость в асинхронных системах	455
14.1. Невозможность консенсуса	455
14.1.1. Обозначения, определения и простейшие результаты	455
14.1.2. Доказательство невозможности построения робастных асинхронных систем	458
14.1.3. Обсуждение	460
14.2. Изначально бездействующие процессы	461
14.3. На что способны детерминированные системы?	464
14.3.1. Разрешимая задача: переименование	465
14.3.2. Более общие результаты о невозможности построения алгоритмов	469
14.4. Вероятностные алгоритмы консенсуса	471
14.4.1. Протоколы консенсуса, робастные относительно выхода процессов из строя	472
14.4.2. Робастные в византийской модели неисправностей протоколы консенсуса	477
14.5. Слабая завершаемость	483
14.6. Упражнения к главе 14	487
Глава 15. Отказоустойчивость в синхронных системах	490
15.1. Синхронные протоколы принятия решения	491
15.1.1. Границы устойчивости	492
15.1.2. Алгоритм широковещательного распространения информации при наличии неисправностей византийского типа	494
15.1.3. Алгоритм широковещательного распространения информации, имеющий полиномиальную сложность	497
15.2. Протоколы с аутентификацией	504
15.2.1. Протоколы, обладающие высокой устойчивостью	504
15.2.2. Как устроены схемы цифровой подписи	508
15.2.3. Схема цифровой подписи Эль-Гамала	509
15.2.4. Схема цифровой подписи RSA	511
15.2.5. Схема цифровой подписи Фиата—Шамира	512
15.2.6. Подведение итогов	514
15.3. Синхронизация часов	517
15.3.1. Как снимать показания с удаленных часов	518
15.3.2. Синхронизация часов в распределенной системе	521
15.3.3. Реализация раундовой модели	526
15.4. Упражнения к главе 15	527
Глава 16. Обнаружение неисправностей	530
16.1. Модель и определения	531
16.1.1. Четыре основных класса детекторов	531
16.1.2. О том, как использовать детекторы неисправностей и подвохи	534
16.2. Решение задачи консенсуса при помощи слабо точных детекторов	535
16.3. Слабо точные в конечном итоге детекторы	537
16.3.1. Верхняя оценка устойчивости	537
16.3.2. Алгоритм консенсуса	539
16.4. Реализация детекторов неисправности	541
16.4.1. Синхронные системы: совершенное обнаружение	542

16.4.2. Частично синхронные системы: в конечном итоге совершенное обнаружение	542
16.4.3. Заключительные замечания	543
16.5. Упражнения к главе 16	544
Глава 17. Стабилизация	546
17.1. Введение	547
17.1.1. Определения	547
17.1.2. Коммуникация в стабилизирующихся системах	549
17.1.3. Пример Дейкстры: кольцо с маркерами	550
17.2. Графовые алгоритмы	553
17.2.1. Ориентация колец	553
17.2.2. Максимальное покрытие	557
17.2.3. Выборы лидера и построение остовных деревьев	560
17.3. Общие методы стабилизации	563
17.3.1. Композиция протоколов	563
17.3.2. Вычисление минимальных путей	570
17.3.3. Заключительные выводы и обсуждение	577
17.4. Упражнения к главе 17	577
Приложение А. Соглашения об использовании псевдокода	579
Приложение Б. Графы и сети	584
Б.1. Определения и термины	584
Б.1.1. Неориентированные графы	584
Б.1.2. Ориентированные графы	586
Б.1.3. Взвешенные графы	586
Б.2. Наиболее распространенные виды графов	587
Б.2.1. Кольца	587
Б.2.2. Деревья, леса и звезды	588
Б.2.3. Клики	590
Б.2.4. Решетки и торы	591
Б.2.5. Гиперкубы	592
Б.3. Восприятие направления	593
Б.4. Упражнения к главе Б	597
Литература	600
Предметный указатель	615

ПРЕДИСЛОВИЕ

В последние годы все большее внимание уделяется распределенным системам и распределенной обработке информации, и почти в каждом университете есть хотя бы один курс лекций по разработке распределенных алгоритмов. Имеется немало книг, посвященных принципам устройства распределенных систем; к их числу относятся, например монографии Таненбаума [182], а также Сломана и Крамера [176]. Однако в этих книгах рассматриваются преимущественно вопросы архитектуры распределенных систем, а не распределенные алгоритмы. После публикации первого издания настоящей книги вышли в свет монографии Барбосы [24], Линч [136], Аттьи и Велча [12].

Как было верно замечено, алгоритмы образуют стеновой хребет каждого компьютерного приложения, и поэтому появление книг, посвященных исключительно распределенным алгоритмам, вполне оправдано. Цель настоящей книги — ознакомить читателей с наиболее значительными достижениями теории распределенных алгоритмов, которые были получены за последние двадцать лет. Эту книгу можно использовать в качестве учебного пособия по курсу распределенных алгоритмов, который может охватывать один или два семестра; преподаватель, задумавший прочитать односеместровый курс, может выбирать темы из этой книги по своему желанию. Профессиональные программисты и научные работники, имеющие дело с распределенными системами, могут найти в этой книге полезный справочный и библиографический материал.

Упражнения. Каждая глава книги (за исключением глав 1 и 13) оканчивается списком упражнений и тем для небольших самостоятельных работ. В этих работах от читателя обычно требуется разработать небольшое, но, тем не менее, нетривиальное развитие или приложение того материала, который был представлен в той или иной главе книги. В большинстве случаев я не могу предложить «готового» решения этих задач, и если читателю удастся завершить один из таких мини-проектов, то автор книги будет рад ознакомиться с полученными результатами. Преподаватели могут получить список ответов (иногда всего лишь частичных) к большинству упражнений, обратившись к автору этой книги или воспользовавшись анонимным ftp.

Исправления и замечания. К тем читателям, которые обнаружат в этой книге ошибки или упущения, автор обращается с просьбой сообщить ему об этом (лучше всего это сделать, воспользовавшись электронной почтой). Все конструктивные замечания, включая предложения новых упражнений, тепло приветствуются.

Признательность. Предварительный вариант этой книги внимательно прочитали мои коллеги: Эрвин Беккер, Ганс Бодлаэндер, Стефан Добрев, Петра ван Хаафтен, Тед Херман, Ян ван Льювен, Патрик Лентферт, Фридеманн Маттерн, Паскаль ван дер Пут, Питер Ружичка, Мартин Рудаликс, Анник Шун и Каса Сере. Их замечания способствовали значительному улучшению качества рукописи. Я получил также полезные предложения от студентов, посещавших курс лекций «Распределенные алгоритмы» в осеннем семестре в Утрехтском университете. Факультет вычислительной математики снабдил меня всем необходимым для обработки и печати текстов. Редакторскую работу выполнила Сьюзан Паркинсон.

ГЛАВА 1

ВВЕДЕНИЕ: РАСПРЕДЕЛЕННЫЕ СИСТЕМЫ

В этой главе объясняется, почему нужно изучать распределенные алгоритмы. Мы начнем с того, что рассмотрим вкратце основные типы электронной аппаратуры и программного обеспечения, для которых были разработаны распределенные алгоритмы. Под распределенной системой мы понимаем всякую вычислительную систему, в которой несколько компьютеров или процессоров так или иначе вступают во взаимодействие. Под это определение наряду с глобальными коммуникационными вычислительными сетями подпадают локальные сети, многопроцессорные компьютеры, в которых каждый процессор наделен своим собственным устройством управления, а также системы взаимодействующих процессов.

В § 1.1 мы рассмотрим разные виды распределенных систем и обсудим вопрос о том, зачем нужны распределенные системы. Там же будут приведены некоторые примеры существующих систем. В этой книге мы не будем, однако, заниматься вопросом о том, как выглядят такие системы и как они используются; главная тема книги — как заставить эти системы работать. И даже в этой теме мы выделим в дальнейшем еще более узкую часть, посвященную *алгоритмам*, которые используются в распределенных системах.

Конечно, изучение одних лишь алгоритмов не приведет нас к полному пониманию того, как устроены распределенные системы и как они функционируют. Чтобы как следует понять такую систему, нужно также полностью изучить архитектуру ее аппаратуры и программного обеспечения, разделение ее функциональной составляющей на модули. Кроме того, есть много важных вопросов, касающихся особенностей тех языков программирования, при помощи которых создается программное обеспечение распределенных систем. Эти вопросы будут являться предметом нашего обсуждения в § 1.2.

К счастью, однако, есть превосходные книги, посвященные распределенным системам, в которых основное внимание уделяется архитектуре и языкам программирования. Среди них можно выделить монографии Таненбаума [182], Сломана и Креймера [176], Бала [21], Колориса и Доллиморе [56], Госцинского [101]. Как было уже сказано, в центре внимания нашей книги будут алгоритмы для распределенных систем. В § 1.3 будет объяснено, почему проектирование распределенных алгоритмов отличается от проектирования централизованных алгоритмов. Мы очертим также области исследования распределенных алгоритмов и коротко опишем содержание этой книги.

1.1. Что такое распределенная система?

В этой главе мы будем использовать термин «распределенная система» для обозначения взаимосвязанной совокупности автономных компьютеров или процессоров. Компьютеры, процессы или процессоры будем называть *узлами* распределенной системы. (В последующих главах мы будем использовать более формальное понятие, см. определение 2.6.) Чтобы узлы могли считаться «автономными», они должны, по меньшей мере, обладать собственным независимым управлением. По этой причине параллельный компьютер, архитектура которого устроена по схеме «одна команда для многих данных» (SIMD¹⁾), не может считаться распределенной системой. Чтобы считаться «взаимосвязанными», узлы должны обладать способностью обмениваться информацией.

Поскольку в качестве узлов системы могут выступать процессы программы, приведенное определение охватывает также и такие программные системы, которые представляют собой совокупность взаимосвязанных процессов, выполняемых на одном и том же вычислительном устройстве. Однако в большинстве случаев в распределенной системе содержится хотя бы несколько процессоров, взаимосвязанных друг с другом при помощи средств коммуникации.

В научной литературе встречаются также и более ограниченные варианты определения распределенных систем. Например, Таненбаум в своей книге [182] называет программную систему распределенной только тогда, когда существование автономных узлов представляется «прозрачным» для пользователей этой системы. В этом случае система, которая считается распределенной, ведет себя так, как положено виртуальной автономной вычислительной системе, но при этом реализация такой прозрачности требует разработки изощренных распределенных алгоритмов управления.

1.1.1. Мотивировка

Существует несколько причин, по которым распределенные системы либо оказываются более предпочтительными, нежели последовательные системы, либо их использование становится просто неизбежным. Мы обсудим здесь некоторые из этих причин, не претендуя на то, что список их окажется исчерпывающе полным. Иногда в пользу выбора распределенной системы можно привести сразу несколько доводов из числа перечисленных ниже, а иногда их преимущества обнаруживаются в качестве побочного эффекта, после того как они были выбраны по какой-то другой причине. Характеристики распределенной системы могут также варьироваться в зависимости от тех обстоятельств, в силу которых потребовалась эта система, но эти вопросы мы обсудим более подробно в §§ 1.1.2–1.1.6.

1. *Обмен информацией.* Потребность в обмене данными между разными компьютерами возникла в шестидесятых годах, когда большинство ведущих университетов и компаний начали обзаводиться своими собственными большими вычислительными машинами. Обмен данными между компьютерами, принадлежа-

¹⁾От англ. Single Instruction – Multiple Data. — Прим. перев.

щими разным организациям, способствовал сотрудничеству между людьми, работающими в этих организациях, и это послужило толчком к развитию так называемых *глобальных сетей*. ARPANET, предшественник современной сети Internet, появилась в 1969 г. Вычислительные установки, подключенные к глобальной сети (их иногда называют *сетями дальней связи*), как правило, снабжены всем, что необходимо для пользователя, включая резервное запоминающее устройство, дисковую память, принтеры, разнообразные прикладные программы.

Позднее компьютеры стали более дешевыми и компактными, и вскоре буквально в каждой организации появилось много вычислительной техники (сейчас обычно каждому работнику выделяется персональный компьютер или рабочая станция). Тогда для электронного обмена информацией между сотрудниками одной и той же организации потребовалось соединить друг с другом автономные компьютеры. Теперь уже нет ничего необычного в том, что в доме, в котором живет одна семья или даже один человек, имеется несколько компьютеров, соединенных друг с другом и образующих небольшую персональную домашнюю сеть.

2. *Совместное использование ресурсов*. Хотя удешевление компьютеров дало организациям возможность снабдить каждого работника персональным компьютером, этого нельзя сказать о таких периферийных устройствах, как резервные запоминающие устройства, дисковые накопители, принтеры. В небольших организациях каждый компьютер, испытывающий потребность в компиляторах и других прикладных программах, может получить поддержку от сервера, к которому он подключен. Кроме того, представляется неэффективным размножать все прикладные программы и сопутствующие им файлы на всех компьютерах; ведь помимо расходования дисковой памяти это создает ненужные трудности, связанные с поддержкой программного обеспечения. Таким образом, компьютеры, которым требуются печатающие устройства или дисковые накопители, могут рассчитывать на поддержку от соответствующих узлов сети. Сеть, объединяющая компьютеры на уровне одной организации, принято называть *локальной сетью*.

Причины, по которым организации выгоднее завести у себя сеть небольших компьютеров вместо одной большой вычислительной машины, — это снижение расходов и возможность наращивания ресурсов. Прежде всего, небольшие компьютеры имеют лучшее соотношение между ценой и производительностью, нежели большие вычислительные машины: типичная большая вычислительная машина работает в 50 раз быстрее типичного мини-компьютера, но стоит в 500 раз дороже. Кроме того, если вычислительная мощность системы становится недостаточной, то к сети можно добавить новые машины (дисковые накопители, серверы, принтеры), чтобы удовлетворить потребности организации. А вот если мощность автономной вычислительной системы становится недостаточной, то в этом случае придется менять всю систему.

3. *Повышение надежности за счет дублирования*. Распределенные системы имеют более высокую потенциальную надежность, чем автономные системы, поскольку они обладают свойством *частичного отказа*. Под этим подразумевается способность системы перераспределять выполнение задач, возло-

женных на вышедшие из строя компоненты системы, между теми узлами, которые продолжают исправно функционировать. А вот выход из строя автономного компьютера наносит непоправимый ущерб всей системе, и продолжать работу в этом случае нет никакой возможности. По этой причине при проектировании высоконадежных вычислительных систем принято обращаться к распределенным архитектурам.

Типичная высоконадежная система состоит из нескольких (двух, трех или четырех) абсолютно идентичных монопроцессоров, на которых выполняется одна и та же прикладная программа, а специальный механизм голосования обеспечивает фильтрацию тех результатов, которые вычисляются на отдельных машинах. Для правильного функционирования распределенных систем, в которых отдельные компоненты могут выходить из строя, требуется достаточно сложная алгоритмическая поддержка.

4. *Повышение производительности за счет параллельного выполнения*. При наличии в распределенных системах большого числа процессоров открывается возможность сокращения общего времени выполнения задач большой вычислительной сложности за счет разделения работы между несколькими процессорами.

Для этой цели создаются специальные параллельные компьютеры, но даже пользователи локальной сети могут получить выгоду от параллельного выполнения, если им удастся возложить решение задач на несколько рабочих станций, функционирующих параллельно.

5. *Упрощение проектирования за счет специализации*. Проектирование компьютерных систем может оказаться очень сложным делом, особенно в тех случаях, когда от них требуются широкие функциональные возможности. И очень часто это проектирование можно упростить, разделив всю систему на отдельные модули так, чтобы каждый модуль мог выполнять только часть функциональной нагрузки, но при этом был соединен с другими модулями.

На уровне одной-единственной программы модулярность достигается путем введения абстрактных типов данных и выделения отдельных процедур для решения разных задач. Системы большего размера представляются в виде семейств взаимодействующих процессов. В обоих случаях эти модули могут выполняться и на однопроцессорном компьютере. Но можно создать и локальную сеть, объединяющую компьютеры разных типов, одни из которых предназначены для проведения численных расчетов, другие — для графических построений, третьи — для работы с файлами и т. д.

1.1.2. Вычислительные сети

Под вычислительной сетью мы понимаем совокупность компьютеров, соединенных между собой при помощи механизмов коммуникации, позволяющих компьютерам сети обмениваться информацией. Этот обмен осуществляется путем отправления и приема сообщений. Компьютерные сети подпадают под наше определение распределенных систем. В зависимости от удаленности компьютеров и их

владельцев друг от друга вычислительные сети могут быть *глобальными* или *локальными*.

Глобальные сети обычно связывают компьютеры, принадлежащие разным организациям (промышленным компаниям, университетам и др.). Узлы сети обычно удалены друг от друга на десятки километров или более. Каждый узел такой сети представляет собой отдельную вычислительную установку, имеющую все необходимые периферийные устройства и достаточно развитое программное обеспечение. Основное назначение глобальных сетей — обеспечить обмен информацией между пользователями, которые подключены к сети в разных узлах.

Локальные сети обычно объединяют компьютеры, принадлежащие одной и той же организации. Узлы сети обычно удалены менее чем на 10 километров. Узлами такой сети, как правило, являются рабочие станции, файловые серверы, серверы печатающих устройств и пр., т. е. сравнительно небольшие станции, на которые возложено решение специальных задач для нужд той организации, которой они принадлежат. Обычно локальные сети предназначены прежде всего для обмена информацией и обеспечения совместного использования ресурсов.

Между этими двумя классами сетей не всегда можно провести четкую границу, и обычно различие между ними не очень существенно с алгоритмической точки зрения, поскольку для всех вычислительных сетей нужно уметь решать сходные алгоритмические задачи. Однако можно выделить следующие отличия, которые нужно учитывать при разработке алгоритмов.

1. *Параметры надежности.* В глобальных сетях никогда нельзя пренебрегать вероятностью того, что по ходу передачи сообщений что-то может сработать неверно, и распределенные алгоритмы для глобальных сетей обычно проектируются так, чтобы они учитывали эту возможность. Локальные сети гораздо более надежны, и алгоритмы для этих сетей можно проектировать исходя из предположения, что связь между узлами является абсолютно надежной. Однако в этом случае необычное событие, в котором происходит что-то неверное, может остаться без внимания, и это приведет к тому, что вся система будет функционировать неверно.

2. *Время связи.* Время передачи сообщений в глобальных сетях на несколько порядков превышает время передачи сообщений в локальных сетях. В глобальных сетях почти всегда можно пренебречь временем, необходимым для обработки сообщений, по сравнению со временем, которое требуется для передачи сообщений.

3. *Однородность.* Хотя в локальных сетях не все узлы оказываются одинаковыми, обычно в рамках одной организации есть возможность достичь согласия по вопросу о том, какое программное обеспечение и протоколы следует использовать. В глобальных сетях используются разнообразные протоколы, и это порождает проблему согласования протоколов и программного обеспечения, разработанных в соответствии с разнообразными стандартами.

4. *Взаимное доверие.* Все пользователи одной и той же организации могут считаться заслуживающими доверия. Это, конечно, неприемлемо для глобальных сетей. Для глобальных сетей необходимо разрабатывать безопасные алгоритмы,

способные обеспечить защиту от враждебных пользователей, подключенных к сети в других узлах.

Глобальные сети коротко обсуждаются в § 1.1.3, а § 1.1.4 посвящен локальным сетям.

1.1.3. Глобальные сети

Исторический очерк. Первые достижения в создании глобальных компьютерных сетей были получены в рамках проекта, который выполнялся Агентством перспективных исследовательских проектов (Advanced Research Projects Agency) министерства обороны Соединенных Штатов Америки. В 1969 г. была запущена сеть ARPANET, которая в то время соединяла четыре узла. Затем к этой сети были подключены еще несколько сот узлов, и на основе подобной технологии были созданы и другие сети (MILNET, CYPRESS). В сети ARPANET имеются специальные узлы (их называют обработчиками интерфейсов сообщений), единственное предназначение которых — обработка потоков сообщений.

Как только система UNIX²⁾ получила широкое распространение, возникла потребность в обмене информацией между различными машинами, на которых установлена операционная система UNIX. Для этого была написана программа *uucp program* (UNIX-to-UNIX CoPy). Эта программа позволила обмениваться файлами по телефонным линиям связи, и вскоре образовалась сеть пользователей системы UNIX (UUCP сеть). Еще одна крупная сеть под названием BITNET была создана в восьмидесятих годах, ввиду того что сеть ARPANET принадлежала министерству безопасности, и лишь немногие пользователи могли подключаться к ней.

В наши дни все эти сети взаимосвязаны между собой: существуют узлы, которые входят в состав сразу двух сетей (такие узлы называются *шлюзами*), и благодаря этому узлы из разных сетей могут обмениваться информацией. После того как были введены стандартные протоколы и унифицированное адресное пространство, все сети превратились в единую виртуальную сеть, которая известна всем под названием Internet. Она отличается от «монолитной» сети тем, что у нее много владельцев и нет единого административного органа, но при этом ее организационная неоднородность скрыта от пользователей. Электронный адрес автора этой книги (gerard@cs.uu.nl) не дает никакой информации о той сети, к которой подключен его факультет.

Устройство и алгоритмические проблемы. Глобальные сети всегда устроены на основе принципа *двухточечного соединения*. Это означает, что механизм соединения между парой узлов является индивидуальным для каждой такой пары. Соединение может осуществляться по телефонной линии, по оптоволоконному кабелю, при помощи космического спутника и т. д. Структуру взаимосвязи сети с двухточечным соединением удобно представлять при помощи рисунка, в котором каждый узел изображается кружком или квадратиком, а линии, проведенные между этими фигурами, соответствуют линиям связи между узлами (см. рис. 1.1).

²⁾UNIX — это торговая марка лабораторий Белла компании AT&T — Прим. автора.

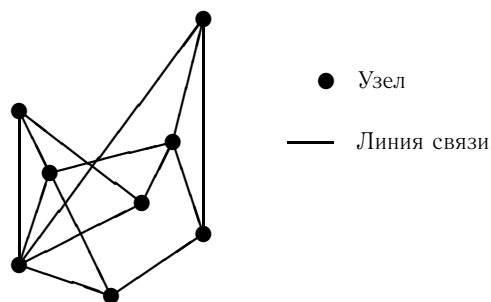


Рис. 1.1. Пример сети с двухточечным соединением

Если воспользоваться более формальной терминологией, то структуру сети можно представить в виде *графа*, ребра которого соответствуют линиям связи сети. Общие сведения о терминологии теории графов можно найти в гл. Б.

Глобальные сети предназначены, в первую очередь, для обмена информацией посредством электронной почты, досок объявлений, удаленных файлов, распределенных баз данных, и т. д. Чтобы воспользоваться большинством этих услуг, достаточно одной прикладной программы — *сетевого обозревателя (веб-браузера)*. Чтобы реализовать удобную коммуникационную систему, подходящую для указанных целей, необходимо решить ряд алгоритмических проблем, некоторые из которых обсуждаются в первой части этой книги.

1. *Обеспечение надежного обмена данными по двухточечной линии связи* (см. гл. 3). Два узла, соединенные линией связи, обмениваются данными по этой линии, но им нужно каким-то образом учитывать возможность того, что линия связи является ненадежной. Из-за атмосферных помех, падения напряжения и целого ряда других физических эффектов сообщение, отправленное по линии связи, может быть частично искажено или даже вообще потеряно. Такие сбои в передаче данных должны быть обнаружены и устранены.

Эта задача возникает не только для узлов, непосредственно связанных друг с другом линией связи, но также и для узлов, которые не имеют прямого соединения, а связаны друг с другом при помощи промежуточных узлов. В этом случае задача становится еще более сложной, поскольку, ко всему прочему, сообщения могут поступать к адресату не в том порядке, в котором они были отправлены, могут поступать, спустя значительный промежуток времени, или дублироваться.

2. *Выбор каналов связи* (см. гл. 4). Обычно бывает слишком накладно выделять отдельную линию связи для каждой пары узлов в сетях с двухточечным соединением. Ввиду этого некоторые пары узлов оказываются соединенными друг с другом при посредничестве других узлов. Задача *маршрутизации* заключается в выборе канала (каналов) связи для соединения двух узлов. Алгоритм, применяемый для выбора канала связи, зависит от схемы именования узлов сети, т. е. от формата того адреса, который может использовать один узел, чтобы отправить сообщение другому узлу. Выбор канала связи в промежуточных вершинах осуществляется на основе указанного адреса, и этот выбор можно сделать более эф-

фективно, если в адресах узлов удастся каким-нибудь образом «закодировать» топологическую информацию.

3. *Устранение перегрузки*. Пропускная способность коммуникационной сети может значительно понизиться, если одновременно передается большое количество сообщений. Поэтому нужно уметь управлять формированием сообщений в различных узлах сети в зависимости от сложившейся пропускной способности сети. Некоторые методы устранения перегрузки обсуждаются в книге [182, § 5.3].

4. *Предотвращение блокировки (тупиков)* (см. гл. 5). Сети с двухточечным соединением иногда называют сетями с *промежуточным хранением данных*, ввиду того что сообщения, которые при пересылке проходят через ряд промежуточных узлов, должны быть вначале помещены в память каждого из этих узлов, а затем переправлены следующему по порядку узлу. Так как в промежуточных узлах объем доступной для этих целей памяти ограничен, с памятью нужно обращаться аккуратно, чтобы избежать тупиковых ситуаций. Это такие ситуации, когда в сети есть некоторое множество сообщений, ни одно из которых не может быть переправлено дальше, поскольку память в следующем узле маршрута полностью занята другими сообщениями.

5. *Обеспечение безопасности*. При помощи сетей соединяются компьютеры, принадлежащие разным владельцам; некоторые из этих владельцев могут попытаться неправомерно воспользоваться ресурсами других владельцев или даже повредить их вычислительные устройства. Так как есть возможность запустить вычислительное устройство из любого места земного шара, нужно иметь надежные методы аутентификации пользователей, криптографии и сканирования поступающей информации (например, для поиска вирусов). Можно воспользоваться методами криптографии (см., например, [182, § 7.1]) для шифрования данных, чтобы обезопасить их от неразрешенного прочтения, и для внесения электронных подписей, чтобы обезопасить их от недозволенной записи.

1.1.4. Локальные сети

Локальные сети используются для того, чтобы установить соединение между компьютерами, принадлежащими одной и той же организации. Обычно такое соединение предназначено, главным образом, для совместного использования ресурсов (как файлов, так и периферийной аппаратуры) и для обеспечения обмена информацией между сотрудниками организации. Время от времени сети применяются также для ускорения вычислений (путем раздельного решения задачи несколькими узлами) и для резервной замены одних узлов другими в случае возникновения неполадок.

Примеры сетей и устройство сетей. В первой половине семидесятых годов XX в. корпорация Xerox разработала локальную сеть Ethernet³⁾. Названия глобальных сетей ARPANET, BITNET и т. п. служат для наименования отдельных

³⁾Ethernet — это торговая марка компании Xerox Corporation. — Прим. автора.

сетей, тогда как названия локальных сетей обычно используются в качестве торговых марок. Существует одна сеть ARPANET, одна сеть BITNET и одна сеть UUCP, существует единая сеть Internet, но каждая компания может установить свою собственную частную сеть Ethernet, Token Ring или SNA.

В отличие от глобальных сетей, сеть Ethernet построена на основе *шины*, т. е. связь между узлами сети обеспечивает единый механизм, к которому подключены все узлы, как это показано на рис. 1.2. Использование шин в локальных сетях получило широкое распространение, хотя при этом могут быть различия в том, как устроены и как применяются эти шины.

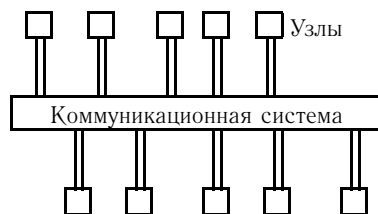


Рис. 1.2. Сеть с шинной организацией

Устройство сети Ethernet позволяет в каждый момент времени передавать только одно сообщение; другие виды шин (к их числу относится, например, *кольцевая сеть с маркерами*, разработанная в Цюрихской лаборатории фирмы IBM) допускают *повторное использование коммуникационного пространства*. Под этим подразумевается, что в одно и то же время по шине могут передаваться несколько сообщений. Для построения шины не требуется сложной аппаратуры, и потому этот способ связи сравнительно дешев. Но у него есть и недостаток — такая организация сети ограничивает возможность ее *наращивания*. Это означает, что максимальное количество узлов, которые можно подключить к одной шине, жестко ограничено. Большие компании, имеющие много компьютеров, должны использовать для создания сети несколько шин, соединенных друг с другом при помощи *мостов*. В результате этого образуются сети, имеющие иерархическую организацию.

Не все локальные сети имеют шинную организацию. В компании IBM удалось спроектировать сетевой продукт под названием SNA, в котором применяется двухточечный тип соединения. Это дало возможность клиентам фирмы подключать к сети разнообразные продукты фирмы IBM. Проектирование сети SNA было осложнено требованием ее совместимости со всеми сетевыми продуктами, уже созданными к тому времени компанией IBM.

Алгоритмические проблемы. При реализации локальных сетей приходится решать некоторые (хотя и не обязательно все) из задач, перечисленных в предыдущем параграфе, посвященном глобальным сетям. Надежный обмен данными уже не представляется значительной проблемой, поскольку шины обычно очень

надежны и передают сообщения очень быстро. Проблемы маршрутизации также не возникают в сетях с шинной организацией, поскольку к каждому адресату можно обращаться непосредственно через шину. В кольцевых сетях все сообщения обычно перемещаются в одном и том же направлении по кольцу и изымаются либо получателем, либо отправителем; поэтому задача маршрутизации становится практически вырожденной. В шинах не возникает перегрузки, так как каждое сообщение может быть получено (изъято из шины) сразу после его отправления, но при этом необходимо ограничивать очередь сообщений, ожидающих в узлах доступа в шину. Так как сообщения не накапливаются в промежуточных узлах, не возникает и блокировки, связанной с промежуточным хранением. Если все компьютеры принадлежат одной семье или одной компании, которая доверяет своим сотрудникам, то нет необходимости и в использовании механизмов безопасности, выходящих за пределы обычной защиты данных, которая обеспечивается операционной системой.

Чтобы применять локальные сети для распределенного выполнения прикладных программ (совокупности процессов, приписанных узлам сети), нужно решить следующие задачи распределенного управления (некоторые из них обсуждаются во второй части этой книги).

1. *Широковещательное распространение информации и синхронизация* (см. гл. 6). Если мы должны добиться того, чтобы информация была доступна всем процессам или чтобы все процессы пребывали в ожидании выполнения некоторого глобального условия, то нам необходимо иметь некоторую схему передачи сообщений, которая каким-то образом будет «затрагивать» все процессы.

2. *Избрание лидера* (см. гл. 7). Некоторые задачи (например, выдача результата или инициализация структур данных) должны выполняться в точности одним процессом из заданного множества. Иногда бывает так, что нам желательно или необходимо не назначать для этого заранее никаких процессов; тогда распределенный алгоритм должен выполняться так, чтобы для решения задачи был избран один из процессов.

3. *Обнаружение завершения* (см. гл. 8). Процессы распределенной системы не всегда способны непосредственно убедиться в том, что распределенное вычисление, в котором они участвуют, завершилось. Для того чтобы можно было получить результаты вычисления, необходимо уметь обнаруживать это завершение.

4. *Назначение ресурсов*. Узлу может потребоваться доступ к некоторому ресурсу, который размещен где-то в сети, хотя точное место его расположения неизвестно. Содержать таблицу, в которой регистрируется расположение каждого ресурса, не всегда выгодно, поскольку количество потенциальных ресурсов может оказаться слишком большим или сами ресурсы могут перемещаться из одного узла в другой. В таком случае заинтересованный узел может обратиться ко всем или некоторым узлам с запросом о доступности этого ресурса, используя для этого, например, механизм широковещательной связи. В основу алгоритмов решения этой задачи могут быть положены волновые методы, описанные в гл. 6 (см. также работу Баратца и др. [23]).

5. *Взаимное исключение.* Проблема взаимного исключения возникает в тех случаях, когда несколько процессов претендуют на некоторый общий ресурс, который в каждый период времени может быть использован только одним процессом. Таким ресурсом может быть печатающее устройство или файл, в который нужно провести запись. Если несколько процессов одновременно обращаются с одним и тем же запросом, распределенному алгоритму приходится определять, какому процессу следует в первую очередь открыть доступ к ресурсу. Кроме того, необходимо сделать так, чтобы следующий процесс приступил к использованию ресурса только после того, как предыдущий процесс завершит работу с этим ресурсом.

6. *Обнаружение и устранение тупиков.* Если процессы должны ожидать друг друга (а это может случиться, если они совместно используют какой-то ресурс, а также если вычисление каждого из них зависит от тех данных, которые должны вычислить другие процессы), может возникнуть циклическая блокировка, которая не позволит продолжить дальнейшее вычисление. Такие тупиковые ситуации должны быть обнаружены, после чего должны быть предприняты адекватные действия для перезапуска или продолжения вычисления.

7. *Распределенное обслуживание файлов.* Когда узел обращается с запросом на запись в файл на удаленной машине или чтение из такого файла, эти запросы могут обрабатываться в произвольном порядке, и поэтому нужно иметь какие-то средства, позволяющие каждому узлу иметь целостное представление об одном или нескольких файлах. Обычно для этой цели запросам, а также информации, содержащейся в файле, придается *штемпель времени*, и запросы упорядочиваются на основании этих штемпелей (см., например, работу [133]).

1.1.5. Многопроцессорные компьютеры

Многопроцессорный компьютер — это вычислительная машина, которая содержит несколько процессоров, размещенных неподалеку друг от друга обычно в корпусе одного устройства. Вычислительные системы этого типа отличаются от локальных сетей: физические размеры машин невелики (обычно они не превышают одного метра), они состоят из совершенно одинаковых процессоров, которые совместно участвуют в проведении одного вычисления либо с целью повышения быстродействия машины, либо с целью повышения надежности. Если многопроцессорный компьютер предназначен для ускорения вычислений, то его часто называют параллельным компьютером. А если он предназначен для повышения надежности вычислений, то его обычно называют вычислительной системой с дублированием.

В зависимости от принципа функционирования параллельные компьютеры подразделяются на два типа: одни из них функционируют по принципу «одна команда для множества данных» (SIMD⁴⁾), а другие — по принципу «множество команд для множества данных» (MIMD⁵). SIMD-машины имеют один-един-

⁴ Single Instruction, Multiple Data. — Прим. перев.

⁵ Multiple Instruction, Multiple Data. — Прим. перев.

ственный интерпретатор команд, но каждая команда выполняется сразу на многих арифметических устройствах. Ясно, что эти устройства не обладают той степенью автономности, которая требуется в нашем определении распределенных систем, и поэтому мы не будем рассматривать в этой книге SIMD-компьютеры. MIMD-машины состоят из нескольких независимых процессоров, и их можно считать распределенными системами.

Все процессоры обычно снабжены необходимым оборудованием для взаимодействия с другими процессорами. Коммуникация между процессорами осуществляется либо через шину, либо посредством двухточечной связи. Если была выбрана шинная организация связи, то возможности для наращивания системы ограничены.

При разработке многопроцессорных компьютеров большую популярность приобрели микроэлектронные схемы Transputer⁶⁾, разработанные компанией Inmos (см. рис. 1.3). Микроэлектронная схема Transputer состоит из центрального

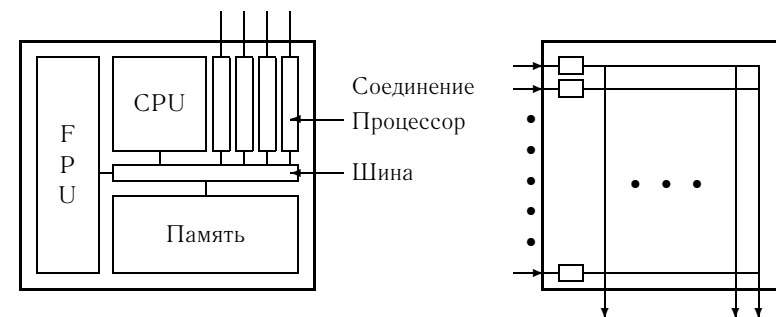


Рис. 1.3. Схема Transputer и маршрутизатор

процессорного устройства (CPU), которое снабжено устройством выполнения операций с плавающей точкой (FPU), локальной памятью и четырьмя коммуникационными процессорами. Эти микроэлектронные схемы очень хорошо приспособлены для построения сетей степени четыре (т.е. таких сетей, в которых каждый узел соединен с четырьмя другими узлами). Компания Inmos также производит специальные микроэлектронные схемы, которые называются *маршрутизаторами* и предназначены для коммуникации. Каждый маршрутизатор может одновременно управлять потоками информации от 32 соединений со схемами Transputer. Каждое поступающее сообщение обследуется, для того чтобы определить, каким соединением нужно воспользоваться для его передачи. Пик популярности этих машин пришелся на первую половину девяностых годов XX в.

Другим примером параллельного компьютера может служить система Connection Machine CM-5⁷⁾, разработка которой была инициирована компанией

⁶ Transputer — торговая марка компании Inmos (в настоящее время SGS-Thomson Microelectronics). — Прим. автора.

⁷ Connection Machine — это лицензионная марка, а CM-5 — это торговая марка компании Thinking Machines Corporation. — Прим. автора.

Thinking Machines Corporation (см. [130]) и продолжена корпорацией Connection Machines Services. Каждый узел этой машины состоит из быстрого процессора и векторного процессорного устройства; таким образом, наряду с параллелизмом, обусловленным наличием множества узлов, в этой машине задействован внутренний параллелизм. Поскольку потенциальная производительность каждого узла составляет 128 миллионов операций в секунду и при этом одна машина может иметь 16384 узлов, вся машина в целом может выполнять до 10^{12} операций в секунду. (Самая большая машина, состоящая из 16384 процессоров, занимает площадь 900 квадратных метров и, вероятно, стоит очень дорого.) Узлы системы CM-5 связаны друг с другом при помощи трех сетей, организованных по принципу двухточечной связи. *Сеть данных*, имеющая топологию *толстого дерева*, служит для обмена данными между процессорами, связанными друг с другом посредством двухточечного соединения. *Сеть управления*, имеющая топологию двоичного дерева, служит для выполнения специальных операций, таких как глобальная синхронизация и комбинирование входных данных. *Сеть диагностики* остается невидимой для программиста; она служит для распространения информации о тех компонентах, которые вышли из строя. На таком компьютере можно осуществлять программирование как в режиме SIMD, так и в (синхронном) режиме MIMD.

В параллельном компьютере всякое вычисление распадается на подвычисления, каждое из которых выполняется в одном из его узлов. В системе с дублированием в каждом узле выполняется все вычисление целиком; после этого полученные результаты сравниваются между собой для того чтобы обнаружить и исправить ошибки.

При построении многопроцессорных компьютеров требуется решить ряд алгоритмических проблем, некоторые из которых аналогичны тем задачам, которые возникают при проектировании вычислительных сетей. Часть из этих задач мы также обсудим в этой книге.

1. *Реализация системы передачи сообщений*. Если многопроцессорный компьютер устроен как сеть с двухточечным соединением, то для этой сети нужно спроектировать коммуникационную систему. В связи с этим становятся актуальными все те же задачи, которые возникают при реализации компьютерных сетей, такие как управление передачей данных, маршрутизация, предотвращение перегрузки и блокировки. Решение этих задач для многопроцессорных компьютеров обычно проще, чем для вычислительных сетей общего вида. Решение проблемы маршрутизации, например, значительно облегчается в связи с тем, что сеть имеет регулярную топологию (например, является кольцом или решеткой), а узлы сети обладают высокой надежностью.

В микросхеме-маршрутизаторе машины Inmos C104 реализован очень простой алгоритм маршрутизации, который мы рассмотрим в параграфе 4.4.2. Этот алгоритм называется алгоритмом *интервальной маршрутизации*, и его нельзя эффективно использовать в сетях с произвольной топологией. В связи с этим возникает вопрос о том, нельзя ли проводить решение других задач, например, задачи предотвращения блокировки, совместно с маршрутизацией (см. курсовую работу 5.5).

2. *Реализация виртуальной разделяемой памяти*. Многие параллельные алгоритмы проектируются на основе модели памяти с параллельным произвольным доступом (PRAM); в рамках этой модели каждый процессор имеет доступ к разделяемой памяти. Те архитектуры, в которых задействована физически разделяемая память, не обладают свойством наращиваемости: существует строгое ограничение на число процессоров, которые могут быть обслужены одним и тем же запоминающим устройством. Поэтому проводятся исследования по разработке таких архитектур, в которых имеется несколько узлов памяти, соединенных с процессорами посредством некоторой сети.

3. *Уравновешивание нагрузки*. Вычислительная мощность параллельных компьютеров проявляется в полной мере лишь тогда, когда вычислительная работа распределяется равномерно между процессорами; сосредоточение всей работы в одном узле приводит к тому, что вся система вырождается в единственный узел. Если все шаги вычисления могут быть выделены во время компиляции программы, то их разнесение по узлам можно провести статически. Гораздо большие трудности возникают тогда, когда отдельные элементарные части вычислительной работы образуются динамически по ходу вычисления; в этом случае приходится применять изощренные методы. Для всех процессоров необходимо регулярно просматривать очереди задач и после этого перемещать задачи от одних процессоров к другим. С обзором некоторых методов и алгоритмов уравновешивания нагрузки можно ознакомиться в работе Госцинского [101, гл. 9] или в работе Харгета и Джонсона [102].

4. *Устойчивость к необнаруживаемым сбоям* (см. часть 3). В системах с дублированием должен быть механизм, позволяющий преодолеть неполадки, возникающие в одном или нескольких процессорах. Конечно, вычислительная сеть обязана продолжать функционировать, невзирая на выход из строя одного из ее узлов, но в этом случае обычно предполагается, что такая неисправность может быть обнаружена другими узлами (см., например, описание алгоритма Netchange в § 4.6.3). Допущения, при которых система с дублированием обязана оставаться правильно работающей, гораздо более жесткие, поскольку процессор может вычислять неверные результаты, но при этом продолжать выполнять протокол взаимодействия подобно правильно работающему процессору. Чтобы провести фильтрацию результатов работы процессоров, нужно задействовать такой механизм голосования, который позволял бы получать только правильные результаты, до тех пор пока большинство процессоров работает исправно.

1.1.6. Взаимодействующие процессы

Часто для упрощения проектирования сложного программного обеспечения можно представить программу в виде совокупности процессов, каждый из которых выполняет строго очерченную простую задачу.

Классическим примером, который приводится в качестве иллюстрации такого упрощения, может служить *схема Конвея преобразования строк*. Задача состоит в том, чтобы прочитать текст, разбитый на строки, состоящие из 80 символов, и представить его в виде записи, разбитой на строки, состоящие из

125 символов. При этом после каждой входной строки нужно поставить пробел и каждую пару звездочек («**») нужно заменить восклицательным знаком («!»). Каждая строка выходного текста должна завершаться признаком конца строки (EOR⁸⁾). Это преобразование может быть выполнено и одной программой, но написать такую программу весьма непросто. Ведь все функции, включая замену «**» на «!», вставку дополнительных пробелов и признаков конца строки, нужно выполнять в теле одного единственного цикла.

Лучше всего представить такую программу в виде двух взаимодействующих процессов. Первый из этих процессов (назовем его p_1) прочитывает подаваемые на вход перфокарты и преобразует прочитанные данные в поток символов, который уже не будет разбит на строки. Второй процесс (назовем его p_2) получает поток символов и вставляет признак EOR после каждой серии из 125 символов. Удобное представление в виде систем взаимодействующих процессов допускают операционные системы, автоматические телефонные станции и, как будет показано в п. 1.2.1, коммуникационное программное обеспечение в вычислительных сетях.

Если в основу проекта положено представление программы в виде семейства взаимодействующих процессов, то полученная в результате программная система будет *логически* распределенной, хотя вполне возможно, что процессы при этом будут выполняться на одном и том же вычислительном устройстве, и в этом случае такая система не будет *физически* распределенной. Конечно же, логически распределенную систему проще сделать физически распределенной. Операционная система вычислительного устройства должна управлять параллельным выполнением процессов и обеспечивать коммуникацию и синхронизацию процессов.

Процессы, которые выполняются на одном процессоре, имеют доступ к одному и тому же запоминающему устройству, поэтому именно это устройство памяти целесообразно использовать для связи между процессорами. Один процесс проводит запись в определенную область памяти, а другой процесс считывает данные из этой области. Такая модель *параллельных процессов* использовалась в работах Дейкстры [60], а также Овицкого и Гриса [155]. В рамках этой модели был изучен ряд следующих задач.

1. *Атомарность операций записи и чтения.* Считается само собой разумеющимся, что при записи или считывании информации из одного машинного слова совершается атомарная операция, т.е. считывание или запись, которую совершает один процесс, завершается до того, как другой процесс приступает к считыванию или записи. В том случае, когда проводится обновление данных в области памяти, состоящей из нескольких слов, нужно проводить тщательную синхронизацию во избежание считывания из частично обновленных областей памяти. Этого можно добиться, воспользовавшись, например, механизмом *взаимно исключенного доступа* к областям памяти [60]: до тех пор пока один процесс имеет доступ к какой-то области памяти, никакой другой процесс не может проводить считывание или запись в эту область. Реализация взаимно исключенного

доступа на основе разделяемых переменных представляется довольно затруднительной из-за возможности того, что сразу несколько процессов могут одновременно претендовать на доступ к одной и той же области памяти.

Проверка условий ожидания, которые возникают при использовании взаимно исключенного доступа к совместно используемым данным, может понизить производительность процессов; это происходит, например, в тех случаях, когда «быстрый» процесс вынужден ожидать разрешения на доступ к тем данным, к которым сейчас обращается «медленный» процесс. В последние годы все большее внимание уделяется разработке разделяемых атомарных переменных, не требующих ожидания; эти переменные позволяют процессу проводить считывание и запись данных, не ожидая других процессов. Считывание и запись теперь могут перекрываться, но при аккуратной реализации алгоритмов считывания и записи атомарность операций может быть тем не менее сохранена. Обзор алгоритмов для разделяемых атомарных переменных, не требующих ожидания, можно найти в работах Кироузиса и Кранакиса [116], а также Атты и Велча [12].

2. *Задача «производитель—потребитель».* Два процесса, один из которых проводит запись в совместно используемый буфер, а другой проводит считывание из этого буфера, должны координировать свои действия, чтобы избежать ситуаций, когда первый процесс пытается записывать данные в переполненный буфер или когда второй процесс пытается считывать данные из пустого буфера. Задача «производитель—потребитель» возникает, например, в схеме Конвея преобразования строк: процесс p_1 порождает поток символов, а процесс p_2 «потребляет» их.

3. *Сборка «мусора».* Прикладные программы, в которых используются динамические структуры данных, могут занимать ячейки памяти, доступ к которым впоследствии утрачивается; такие области памяти называются *мусором*. Раньше выполнение прикладной программы должно было прерываться по исчерпанию свободной памяти, чтобы позволить специальной программе, которая называется *сборщиком мусора*, обнаружить и восстановить неиспользуемую память. Дейкстра и др. в работе [64] придумали, как проводить сборку мусора *оперативно* (on-the-fly): сборщик мусора реализуется в виде отдельного процесса, который выполняется параллельно с прикладной программой.

Взаимодействие между прикладной программой и сборщиком мусора оказывается весьма замысловатым, потому что прикладная программа может изменять в памяти структуру указателей, пока сборщик выясняет, какие из ячеек памяти оказываются недоступными. Нужно провести тщательный анализ алгоритма, чтобы убедиться в том, что исключены ситуации, когда доступные ячейки ошибочно утилизируются при модификации структуры. Алгоритм оперативной сборки мусора, имеющий простое доказательство корректности, был предложен в работе Бен-Ари [27].

Решения всех тех задач, которые были перечислены выше, свидетельствуют о том, что даже очень трудные проблемы взаимодействия процессов могут быть решены для процессов, связь между которыми осуществляется при помощи совместно используемой памяти. Однако решения эти часто оказываются чрезвычайно изощренными и порой даже небольшие изменения порядка выполнения

⁸⁾End Of Record. — Прим. перев.

действий разными процессами приводят к тому, что решения, которые на первый (и даже на второй) взгляд казались правильными, приводят к ошибочным результатам. Поэтому в операционных системах и языках программирования заложены некоторые простые базовые средства для более структурированной организации взаимосвязи между процессами.

1. *Семафоры*. *Семафором* (см. [60]) называется переменная, принимающая неотрицательные целые значения, запись и считывание значений которой осуществляется при помощи одной атомарной операции. Операция **V** служит для увеличения ее значения, а операция **P** — для уменьшения ее значения в том случае, когда это значение оказывается положительным (если значение оказывается отрицательным, то эта операция приостанавливает выполнение процесса, до тех пор пока это значение не станет положительным).

Семафоры являются подходящим инструментальным средством для реализации взаимно исключенного доступа к совместно используемым структурам данных: начальное значение семафора полагается равным 1, перед тем как осуществить доступ к структуре данных, выполняется операция **P**, а после завершения работы с данными выполняется операция **V**. Семафоры возлагают на каждый процесс значительную ответственность за их правильное использование; целостность совместно используемых данных будет нарушена, если процесс будет неправильно преобразовывать данные или будет некорректно выполнять операции **P** и **V**.

2. *Мониторы*. *Монитор* (см. [107]) представляет собой структуру данных и семейство процедур, выполняемых на этих данных путем взаимно исключающих вызовов. Доступ к данным осуществляется исключительно посредством процедур, определенных в мониторе, и это гарантирует корректное использование данных в том случае, когда монитор описан корректно. Таким образом, монитор предотвращает неограниченный доступ к данным и позволяет синхронизировать доступ разных процессов к этим данным.

3. *Трубки*. *Трубкой* (см. [34]) называется механизм перемещения потока данных от одного процесса к другому и синхронизации двух взаимосвязанных процессов; по сути дела трубка — это предварительно запрограммированное решение задачи «производитель—потребитель». Трубка — это основной механизм связи в операционной системе UNIX. Если программы **p1** и **p2** представляют собой реализации процессов p_1 и p_2 в схеме Конвея преобразования строк, то при помощи команды **p1 | p2** в системе UNIX осуществляется вызов этих программ и установление связи между ними посредством трубки. Выходные данные программы **p1** поступают в буфер и становятся входными данными для программы **p2**; программа **p1** приостанавливает свое выполнение, если буфер переполнен, а программа **p2** приостанавливает свое выполнение, если буфер опустошен.

4. *Обмен сообщениями*. В некоторых языках программирования (к их числу относятся языки программирования Оссам или Ада) в качестве средств межпроцессного взаимодействия используются операторы обмена сообщениями. При помощи обменов сообщениями сравнительно легко решаются проблемы синхронизации: поскольку сообщение нельзя получить прежде, чем оно было отправле-

но, в результате проведенного обмена сообщениями между событиями возникает темпоральное отношение предшествования.

Обмен сообщениями можно реализовать при помощи мониторов или трубок; это естественный способ организации связи в программных системах, которые выполняются на распределенных вычислительных устройствах без совместного использования памяти. На самом деле языки Оссам и Ада создавались в расчете на разработку физически распределенных прикладных программ.

1.2. Архитектура и языки

Программное обеспечение, необходимое для создания компьютерных сетей связи, устроено очень сложно. В этом параграфе мы расскажем о том, каким образом это программное обеспечение подразделяется на ациклически зависящие друг от друга модули, которые называются *уровнями* (§1.2.1). Мы рассмотрим два стандарта сетевой архитектуры: модель взаимодействия открытых систем ISO⁹⁾, которая служит стандартом для глобальных сетей, и дополнительный IEEE стандарт для локальных сетей (§1.2.2 и §1.2.3). Мы также обсудим особенности языков программирования распределенных систем (§1.2.4).

1.2.1. Архитектура

Обеспечение связи между компонентами распределенной системы является сложной задачей, и поэтому коммуникационная подсистема должна быть очень хорошо структурирована. Для этого сети обычно приходится строить из отдельных модулей, каждый из которых служит для выполнения очень специальной функции при поддержке других модулей. Сеть устроена так, что эти модули образуют строгую *иерархию*: каждый модуль может обращаться только к тем модулям, которые непосредственно предшествуют ему в этом иерархическом порядке. Эти модули называются *уровнями* или *слоями* (см. рис. 1.4).

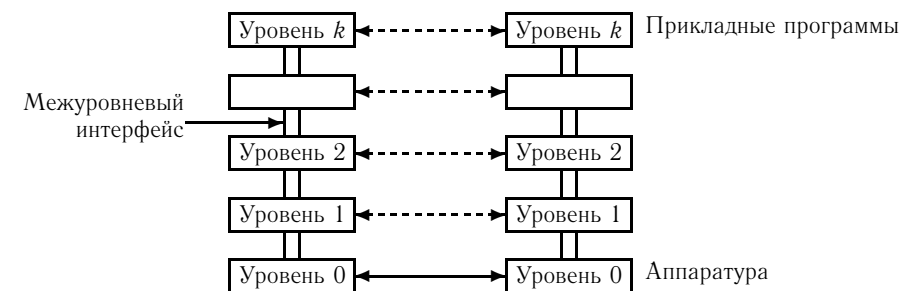


Рис. 1.4. Многоуровневая архитектура сети

⁹⁾International Standards Organization — Прим. перев.

На каждом уровне реализована только часть тех функций, которые необходимы для работы сети, и каждый уровень может опираться только на те функции, которые реализованы на предшествующем уровне. Те сервисные функции, которые уровень i предоставляет уровню $i+1$, подробно описаны в *интерфейсе взаимодействия* уровня $i+1$ с уровнем i (его коротко называют $i/(i+1)$ -интерфейс). Проектирование сети начинается с того, что определяется необходимое число уровней и прописывается интерфейс взаимодействия соседних уровней.

Все функции каждого уровня должны вычисляться распределенным алгоритмом, и при этом алгоритм на уровне i предназначен для решения тех «задач», которые определены в $i/(i+1)$ -интерфейсе на основе тех «допущений», которые определены в $(i-1)/i$ -интерфейсе. Например, в $(i-1)/i$ -интерфейсе может быть объявлено о возможности передавать сообщения из узла p в узел q без гарантии надежной доставки этих сообщений, а в $i/(i+1)$ -интерфейсе указано, что имеются средства для надежной передачи сообщений из узла p в узел q . Таким образом, на уровне i требуется решить следующую алгоритмическую проблему: располагая средством передачи сообщений, обеспечить надежную передачу сообщений. Это обычно достигается при помощи подтверждений и повторной передачи потерянных сообщений (см. § 1.3.1 и гл. 3). При решении этой задачи определяется тип сообщений, которыми обмениваются процессы на уровне i , и смысл этих сообщений, или, иными словами, то, как на эти сообщения должны реагировать процессы. Правила и соглашения, которыми руководствуются процессы в «разговоре» друг с другом на уровне i , называют *протоколом уровня i* .

Самым низким уровнем иерархии (уровнем 0 на рис. 1.4) обычно является аппаратный уровень. В 0/1-интерфейсе описаны процедуры, которые позволяют передавать непереработанную информацию на уровне 1 по проводам связи, а в определении самого уровня 0 приводятся описания типов используемых проводов, напряжения в цепи, служащего для представления единицы и нуля, и т. д. Очень важно заметить, что изменения, происходящие в реализации системы на уровне 0 (например, замена проводов одного типа проводами другого типа или спутниковой связью), не требуют внесения изменений в 0/1-интерфейс. То же самое происходит и на верхних уровнях: межуровневый интерфейс скрывает подробности реализации одного уровня от других уровней, и поэтому изменения, происходящие в этих реализациях, никак не сказываются на других уровнях.

Архитектурой сети принято называть совокупность всех модулей (уровней) вместе с определениями соответствующих интерфейсов и протоколов. Так как в сети могут быть узлы, изготовленные разными производителями, имеющие программное обеспечение от разных компаний, важно, чтобы продукты, созданные разными компаниями, были совместимы. Потребность в совместимости была осознана во всем мире, и поэтому были разработаны стандартные сетевые архитектуры. Два таких стандарта получили «официальный» статус, поскольку они были утверждены авторитетными организациями (Международной организацией стандартов, ISO и Институтом инженеров по электротехнике и электронике, IEEE¹⁰⁾). Мы рассмотрим их в последующих параграфах этой главы.

¹⁰⁾Institute of Electrical and Electronics Engineers. — Прим. перев.

Протокол управления передачей/Интернет-протокол (TCP/IP¹¹⁾) представляет собой семейство протоколов, которые используются сетью Internet. Он не является официально учрежденным стандартом, но используется столь широко, что стал стандартом *de facto*. Семейство протоколов TCP/IP (для ознакомления с ним см. работу Девидсона [59]) разбито на уровни, которые соответствуют уровням эталонной модели взаимодействия открытых систем и будут рассмотрены нами в следующем параграфе. Вместе с тем, эти протоколы могут быть использованы как в глобальных, так и в локальных сетях. На более высоких уровнях представлены протоколы для электронной почты (простой протокол электронной почты, SMTP¹²⁾), передачи файлов (FTP¹³⁾), двусторонней связи дистанционных входов в систему (Telnet).

1.2.2. Эталонная модель OSI

Международная организация стандартов (ISO) утвердила стандарт для тех компьютерных сетевых продуктов, которые используются в глобальных сетях. Разработанный стандарт получил название эталонная модель *взаимодействия открытых систем* (OSI¹⁴⁾). В этом параграфе мы приведем его краткое описание. Ввиду того что этот стандарт не очень удобен для использования применительно к локальным сетям, в следующем параграфе мы рассмотрим дополнительные стандарты IEEE, предназначенные для локальных сетей.

Эталонная модель OSI имеет семь уровней: *физический уровень, уровень передачи данных, сетевой уровень, транспортный уровень, сеансовый уровень, представительский уровень и прикладной уровень*. В эталонной модели описаны межуровневые интерфейсы, и для каждого уровня представлены один или несколько стандартных протоколов (распределенных алгоритмов, которые являются реализациями рассматриваемого уровня).

Физический уровень (1). Физический уровень предназначен для передачи последовательностей битов по каналам связи. Как следует из названия этого уровня, поставленная цель достигается за счет физического соединения двух узлов телефонной линией, оптоволоконным кабелем или при помощи спутниковой связи. Устройством этого уровня занимаются исключительно специалисты в области электротехники, а в 1/2-интерфейсе указываются процедуры, посредством которых следующий уровень обращается к служебным функциям физического уровня. Служебные функции физического уровня ненадежны: при передаче потока данных в нем может возникнуть беспорядок.

Уровень передачи данных (2). Уровень передачи данных предназначен для того, чтобы исправить и тем самым скрыть ненадежность физического уровня, т. е. предоставить вышестоящим уровням надежные каналы передачи данных. На

¹¹⁾Transmission control protocol/Internet protocol. — Прим. перев.

¹²⁾Simple Mail Transfer Protocol. — Прим. перев.

¹³⁾File Transfer Protocol. — Прим. перев.

¹⁴⁾Open-Systems Interconnection. — Прим. перев.

уровне передачи данных надежное соединение обеспечивается только между теми узлами, которые непосредственно соединены друг с другом при помощи какого-нибудь физического средства связи, потому что этот уровень опирается только на физический уровень. Если узлы не являются смежными, то коммуникация между ними осуществляется на сетевом уровне.

Чтобы выполнить поставленную задачу, битовый поток на уровне передачи данных разбивают на кусочки одинаковой длины, которые называются *фреймами*. При получении фрейма проводится проверка его целостности; для этого применяется метод *контрольных сумм*, предусматривающий внесение в каждый фрейм некоторой избыточной информации. Между получателем и отправителем имеется обратная связь, дающая возможность оповестить отправителя о том, не подвергся ли полученный фрейм искажению. Эта обратная связь осуществляется передачей *подтверждающих* сообщений. В том случае, если фрейм был искажен или потерян, отправитель должен отправить его повторно.

Эти общие принципы устройства уровня передачи данных требуют более подробного уточнения, и это приводит к разнообразным протоколам передачи данных. Например, подтверждающее сообщение может быть отправлено тогда, когда фрейм был получен (положительное подтверждение), или тогда, когда фрейм был потерян (отрицательное подтверждение). Основная ответственность за правильность доставки всех фреймов может быть возложена как на отправителя сообщений, так и на их получателя. Подтверждения могут быть отправлены как для отдельных фреймов, так и сразу для нескольких фреймов, фреймы могут иметь или не иметь порядковых номеров, и т. п.

Сетевой уровень (3). Сетевой уровень предназначен для того, чтобы обеспечить связь между всеми узлами сети, а не только теми, между которыми есть физический канал связи. На этом уровне проводится выбор маршрутов в сети для соединения удаленных друг от друга узлов, а также контроль нагрузки в каждом узле и канале.

Выбор маршрутов обычно проводится на основании информации о топологии сети, которая отражена в *таблицах маршрутизации*, хранящихся в каждом узле. На сетевом уровне имеются алгоритмы, которые обновляют таблицы маршрутизации всякий раз, когда топология сети изменяется вследствие повреждения или восстановления узлов и каналов связи. Такие повреждения или восстановления выявляются на уровне передачи данных.

Хотя на уровне передачи данных обеспечиваются надежные соединения, связь на сетевом уровне оказывается ненадежной. Сообщения (на сетевом уровне они называются *пакетами*), отправленные из одного узла в другой, могут следовать разными путями, и в связи с этим одни сообщения могут обгонять другие. Из-за сбоев в работе узлов некоторые сообщения могут быть потеряны (узел, в котором временно хранится сообщение, может, например, выйти из строя), а ненужные повторные передачи могут привести к дублированию сообщений. Время жизни пакетов на сетевом уровне ограничено: существует такая константа s , что каждый пакет либо будет доставлен не более чем за s секунд, либо будет считаться потерянным.

Транспортный уровень (4). Транспортный уровень предназначен для того, чтобы исправить и тем самым скрыть ненадежность сетевого уровня, т. е. обеспечить надежную *сквозную передачу* данных из одного узла в другой. Эта задача очень похожа на аналогичную задачу, которая решается на уровне передачи данных, но здесь она осложняется тем, что сообщения могут дублироваться, и порядок их следования может быть изменен. Поэтому циклические числовые последовательности можно использовать для нумерации пакетов только тогда, когда на сетевом уровне определено время жизни пакетов.

Алгоритмы, управляющие передачей данных на транспортном уровне, устроены на основе тех же принципов, что и алгоритмы уровня передачи данных: нумерация пакетов, обратная связь с подтверждениями, повторная передача.

Сеансовый уровень (5). На сеансовом уровне реализованы необходимые средства для взаимосвязи процессов, работающих в разных узлах. Сеанс связи может открываться и закрываться, и в то время, когда сеанс связи открыт, соединение может быть использовано для обмена данными с применением сеансового адреса вместо повторения в каждом сообщении адреса удаленного процесса. На сеансовом уровне используется надежная сквозная коммуникация, которую предоставляет транспортный уровень, но обмен сообщениями при этом разбивается на сеансы связи.

Во время сеансов связи выполняется передача файлов или осуществляется дистанционный вход в систему. На сеансовом уровне разработаны как средства восстановления сообщений, если какой-либо узел выходит из строя во время сеанса связи, так и механизмы взаимного исключения, если критические операции запрещается выполнять одновременно обоими участниками сеанса.

Представительский уровень (6). Представительский уровень предназначен для преобразования данных в тех случаях, когда формы представления информации в одном узле отличаются от форм представления информации в другом узле или являются непригодными для передачи данных. На нижележащем уровне (т. е. в рамках 5/6-интерфейса) считается, что все данные представлены в стандартной форме и пригодны для передачи, а на вышележащем уровне (т. е. в рамках 6/7-интерфейса) данные могут быть представлены в самых разных формах в зависимости от типов компьютеров и пожеланий пользователей. На представительском уровне проводится *сжатие* и *восстановление* данных с целью сокращения объема информации, передача которой осуществляется на более низких уровнях. Кроме того, на этом же уровне проводится *шифрование* и *дешифрование* сообщений с целью обеспечения свойств конфиденциальности и целостности, позволяющих предотвратить попытки злоумышленников получить доступ к передаваемым данным или внести в них изменения.

Прикладной уровень (7). На прикладном уровне реализованы те средства, которые обеспечивают потребности пользователей в передаче файлов, в использовании электронной почты, виртуальных терминалов и пр. Разнообразие прикладных программ очень обширно, и поэтому провести стандартизацию функциональных возможностей на прикладном уровне невозможно, но для некоторых

прикладных программ, которые были упомянуты здесь, были разработаны определенные стандарты.

1.2.3. Модель OSI для локальных сетей: стандарт IEEE

В устройстве эталонной модели OSI нашли отражение многие особенности архитектуры существовавших в то время глобальных сетей. В локальных сетях применяется технология, которая предъявляет к программному обеспечению иные требования, и поэтому в локальных сетях некоторые из рассмотренных нами уровней становятся совсем излишними. Если сеть устроена так, что все узлы связаны друг с другом общей шиной (см. § 1.1.4), то сетевой уровень становится практически ненужным, потому что любые две вершины соединены друг с другом при помощи шины. Транспортный уровень тоже заметно упрощается, поскольку поведение шины не столь сложно по сравнению с поведением сети с двухточечным соединением. Напротив, на уровне передачи данных трудности возрастают, потому что большое число узлов имеют доступ к одной и той же физической среде.

Откликнувшись на возникшие задачи, IEEE утвердил дополнительные стандарты, охватывающие только те нижние уровни иерархии OSI, которые используются в локальных сетях (точнее говоря, во всех сетях, связь в которых осуществляется посредством шины, а не двухточечного соединения). Ввиду того что многообразие всех широко распространенных сетей нельзя было охватить единым общим стандартом, IEEE утвердил три разных несовместимых стандарта: CSMA/CD, шина с маркерами и кольцо с маркерами. Уровень передачи данных был расщеплен на два подуровня, а именно на подуровень *управления доступом к среде* и подуровень *управления логической связью*.

Физический уровень (1). В стандартах IEEE физический уровень предназначен для решения тех же самых задач, что и в исходном стандарте ISO, — передавать последовательности битов. Однако в новых стандартах определения (тип проводов и пр.) уже совсем другие ввиду того что все соединения проводятся в одной и той же среде, в отличие от двухточечной связи.

Подуровень управления доступом к среде (2а). Этот подуровень предназначен для разрешения тех конфликтов, возникающих между узлами, использующими общую коммуникационную среду. Статический метод предполагает составление неизменного расписания, в котором каждому узлу выделяются определенные отрезки времени для использования коммуникационной среды. В этом методе пропускная способность канала связи расходуется неэкономно, если данными обмениваются всего лишь несколько узлов, а остальные хранят молчание: средства связи простаивают все то время, которое выделяется этим безмолвным узлам.

В шинах с маркерами и в кольцах с маркерами доступ к средствам связи передается по кругу: узлы передают друг другу определенную привилегию, которая называется *маркером*, и тот узел, который владеет маркером, может обращаться

к средствам связи. Если узлу, владеющему маркером, нечего передавать, он передает маркер следующему узлу. В кольце с маркерами циклическая очередность предоставления доступа к связи определяется топологией физического соединения узлов (которая, как следует из названия, является кольцом), а в шине с маркерами циклический порядок определяется динамически в соответствии с порядком, который задан на множестве адресов узлов.

В стандарте множественного доступа с контролем несущей и обнаружением коллизий (CSMA/CD¹⁵) узлы следят за тем, когда освободятся средства связи, и, как только это происходит, предпринимают попытку отправить сообщения. Если два или более узлов попытаются отправить сообщения (почти) одновременно, возникает *коллизия*. После обнаружения коллизии каждый из узлов прерывает передачу сообщения и предпринимает очередную попытку связи некоторое время спустя.

Подуровень управления логической связью (2б). Этот подуровень имеет почти такое же назначение, что и уровень передачи данных в модели OSI, — он служит для управления обменом данными между узлами. Этот уровень нужен для того, чтобы следить за ошибками и управлять потоком информации; для этого применяются почти такие же приемы, какие используются в протоколах OSI, а именно последовательная нумерация и подтверждение.

С точки зрения вышележащих уровней подуровень управления логической связью выглядит точно так же, как сетевой уровень в модели OSI. И в самом деле, коммуникация между любой парой узлов осуществляется без привлечения промежуточных узлов, и ее обслуживание можно проводить непосредственно на уровне управления логической связью. Поэтому в локальных сетях сетевой уровень не реализуется, и транспортный уровень надстраивается прямо над уровнем управления логической связью.

1.2.4. Языковая поддержка

Для реализации программного обеспечения на любом уровне коммуникационной сети, равно как и для создания прикладных распределенных программ, нужно запрограммировать соответствующие распределенные алгоритмы на каком-либо языке программирования. Построение программного кода в значительной мере определяется особенностями языка программирования и, в первую очередь, теми базовыми конструкциями, которые представлены в этом языке. В этой книге нас будут интересовать алгоритмы, а не их программные реализации, поэтому при описании базовой модели мы будем говорить о состояниях процессов и о переходах из одних состояний в другие (см. § 2.1.2), а не о выполнении операторов из некоторого заданного множества. Конечно, для описания алгоритмов нужна строгая формальная система обозначений. В гл. А рассказывается о том программном псевдокоде, который используется в этой книге.

В этом параграфе мы обсудим некоторые из основных конструкций, которые присутствуют в реальных языках программирования, предназначенных для

¹⁵Carrier sense multiple access with collision detection. — Прим. перев.

проектирования распределенных систем. Мы ограничимся лишь кратким описанием этих конструкций; их подробное описание, а также примеры реальных языков программирования, в которых задействованы эти конструкции, можно найти, например, в книге Бала [21]. В любом языке, предназначенном для создания распределенных прикладных программ, должны быть средства для представления параллельного выполнения, взаимодействия процессов и недетерминизма. Средства представления параллельных вычислений, безусловно, нужны для того, чтобы программа для различных узлов позволяла работать этим узлам одновременно. В языке программирования должны быть также средства для организации взаимосвязи между узлами. Недетерминизм также необходим, потому что в некоторые узлы могут поступать сообщения сразу из нескольких различных узлов; иногда бывает нужно сделать так, чтобы узел имел альтернативную возможность принять сообщение или отправить сообщение.

Параллелизм. Наиболее подходящая степень параллелизма в распределенных прикладных программах зависит от того, как соотносятся друг с другом расходы на коммуникацию и расходы на вычисление. Высокая степень параллелизма позволяет ускорить выполнение программы, но при этом возрастают коммуникационные издержки, и в том случае, когда связь обходится слишком дорого, выигрыш, достигнутый за счет повышения быстродействия системы, не сможет компенсировать дополнительные расходы на поддержание связи.

Как правило, для того чтобы выразить параллелизм, достаточно определить несколько *процессов*, каждый из которых является последовательной процедурой со своим собственным пространством состояний. В языке могут быть заложены возможности как для статического определения процессов, так и для динамического создания процессов и завершения их работы. Параллелизм также можно выразить посредством специальных параллельных операторов или при помощи языка функционального программирования. Параллелизм не всегда присутствует в языке в явном виде: изолированные компиляторы способны сами разделять программу на параллельные процессы.

Коммуникация. Неотъемлемой чертой распределенных систем является взаимосвязь между процессами: если процессы не взаимодействуют, то каждый процесс работает изолированно от других процессов, и поэтому его нужно рассматривать отдельно от всей распределенной системы. Если процессы в своих вычислениях взаимодействуют друг с другом, то необходимость в связи между ними возникает всякий раз, когда один из процессов испытывает потребность в промежуточных результатах, которые вычисляют другие процессы. Столь же необходима и синхронизация, потому что выполнение такого процесса должно быть приостановлено, до тех пор пока не будут получены ожидаемые результаты. Коммуникация посредством *обмена сообщениями* позволяет обеспечить как коммуникацию, так и синхронизацию. А вот *совместно используемая память* годится только для коммуникации: чтобы синхронизовать работу процессов, которые связаны посредством совместно используемой памяти, требуется аккуратно применять дополнительные средства.

В тех языках, в которых коммуникация обеспечивается путем обмена сообщениями, имеются операторы отправления сообщений *send* и приема сообщений *receive*. Связь происходит при выполнении оператора отправления сообщения в одном из процессов (этот процесс называется отправителем) и оператора приема сообщения в другом процессе (этот процесс называется получателем). Аргументами операции отправления являются адрес получателя, а также дополнительные данные, составляющие содержание сообщения. Эти дополнительные данные становятся доступны получателю, когда выполнится оператор приема сообщения. Таким образом осуществляется взаимосвязь процессов. Операция приема сообщений может завершиться только после того, как будет выполнена операция отправления сообщения. Таким образом достигается синхронизация. В некоторых языках нет операции приема сообщения в явном виде; вместо этого в результате приема сообщения происходит автоматическая неявная активизация некоторой операции или процедуры.

В языке может поддерживаться *синхронный* обмен сообщениями; в этом случае оператор отправления сообщения завершает свое выполнение только после того, как начнется выполнение оператора приема сообщения. Иначе говоря, отправитель остается заблокированным, до тех пор пока не будет принято сообщение, и благодаря этому достигается двухсторонняя синхронизация.

Сообщения могут отправляться по двухточечной связи, т. е. одному получателю от одного отправителя, или широковещательно, и в этом случае одно и то же сообщение будет доставлено всем получателям. В последнем случае для сообщения, которое отправляется целому множеству получателей (хотя и не обязательно всем процессам), используется термин «шировещательное сообщение». Несколько более структурированным базовым средством связи является *вызов удаленной процедуры* (RPC¹⁶⁾). Чтобы установить соединение с процессом *b*, процесс *a* вызывает процедуру, входящую в состав процесса *b*, путем отправления сообщения, содержащего параметры этой процедуры; при этом процесс *a* приостанавливает свое выполнение, до тех пор пока результат вычисления этой процедуры не будет возвращен ему с другим сообщением.

Альтернативу обмену сообщениями как средству связи составляет *совместно используемая память*; один из процессов присваивает значение некоторой переменной, а другой процесс считывает это значение. В этом случае труднее добиться синхронизации процессов, потому что считывание нужного значения может произойти до того, как оно будет присвоено переменной. Применяя такие средства синхронизации, как *семафоры* [60] или *мониторы* [107], можно реализовать передачу сообщений через совместно используемое окружение. И наоборот, при помощи механизма обмена сообщениями можно реализовать виртуальную модель совместно используемой памяти, хотя это крайне неэффективный способ.

Недетерминизм. При выполнении процесса есть много точек, в которых это выполнение может быть продолжено по разным направлениям. Операция приема сообщения часто оказывается недетерминированной, потому что сообщение

¹⁶⁾ Remote Procedure Call. — Прим. перев.

может быть получено от многих разных отправителей. Недетерминизм можно выразить, например, при помощи *охраняемых команд*, или *команд с предохранителями*. В самом общем виде охраняемая команда представляет собой список операторов, каждому из которых предшествует булево выражение (его предохранитель). Процесс может продолжить свое выполнение, выбрав любой из тех операторов, предохранитель которого принимает значение true. В состав предохранителя может входить операция приема сообщения; в этом случае предохранитель принимает истинное значение, если имеется сообщение, которое может быть получено.

1.3. Распределенные алгоритмы

В предшествующих пара параграфах этой главы мы уже говорили о том, зачем нужны распределенные вычислительные системы и каковы их характерные особенности. Понятно, что для этих систем нужно создавать программы. Программы для распределенных систем должны основываться на правильных, гибких и эффективных алгоритмах. В этом параграфе мы покажем, что искусство разрабатывать распределенные алгоритмы очень сильно отличается от умения создавать централизованные алгоритмы. Распределенные и централизованные системы отличаются в целом ряде аспектов, которые мы обсудим в § 1.3.1; в качестве иллюстрации в § 1.3.2 мы приведем несколько примеров. Поэтому изучение распределенных алгоритмов — это отдельная область научных исследований (см. § 1.3.3), введением в которую может служить эта книга. В § 1.4 говорится о том, какие цели ставил перед собой автор книги, и приводится перечень результатов, включенных в эту книгу.

1.3.1. Сравнение распределенных и централизованных алгоритмов

Распределенные системы отличаются от централизованных (однопроцессорных) вычислительных систем тремя наиболее существенными качествами, которые мы здесь обсудим.

1. *Отсутствие сведений о глобальном состоянии.* В централизованном алгоритме управление вычислением осуществляется путем отслеживания состояния всей системы. И хотя состояние всей системы обычно нельзя оценить, выполнив одну-единственную машинную операцию, программа может проверить одну за другой все переменные и выработать решение, как только будет собрана вся необходимая информация. Пока проводится проверка и принимается решение, никакие данные не изменяются, и этим обеспечивается целостность принятого решения.

Узлам распределенной системы открыт доступ только к их собственным состояниям, а не к состоянию всей системы. Поэтому нельзя управлять вычислением, основываясь на информации о глобальном состоянии системы. Конечно, узел может получить информацию о состоянии других узлов и принять решение, учитывая эту информацию. Но, в отличие от централизованных систем, за время,

прошедшее между моментом отправления этой информации и моментом принятия решения, состояние других узлов может измениться, и это обстоятельство делает устаревшую информацию неактуальной.

Узлы системы не способны оценить состояние коммуникационной подсистемы (в частности, они не могут знать о том, какие сообщения пребывают на этапе пересылки в тот или иной момент времени), поскольку содержимое каналов связи не поддается прямому наблюдению со стороны узлов. Эти сведения можно получить косвенным образом, проведя дедуктивный анализ информации о тех сообщениях, которые были отправлены и получены всеми узлами системы.

2. *Отсутствие глобальной шкалы времени.* Все события, которые возникают при выполнении централизованного алгоритма, можно вполне упорядочить естественным образом в зависимости от времени их осуществления; для каждой пары событий одно из них происходит ранее или позднее другого. Подобное отношение предшествования по времени, введенное на множестве событий, которые возникают при выполнении распределенного алгоритма, не является тотальным: для некоторых пар событий есть основания считать, что одно из них происходит ранее или позднее другого, но есть и такие пары событий, ни одно из которых нельзя считать произошедшим ранее или позднее другого [124].

В централизованной системе очень просто добиться взаимно исключенного доступа к ресурсу: если процесс p обращается к ресурсу позже процесса q , то и доступ к ресурсу будет предоставлен процессу p только после того, как процесс q освободит этот ресурс. И в самом деле, все события, о которых идет речь (начало и окончание доступа к ресурсу процессами p и q), вполне упорядочены отношением предшествования по времени. В распределенных системах такого может не быть, и этой стратегии уже недостаточно. Процессы p и q могут обратиться к ресурсу, но ни для одного из этих процессов начало его обращения к ресурсу не предшествует по времени началу обращения другого.

3. *Недетерминизм.* Для заданных входных данных централизованная программа строит одно-единственное вычисление, оно разворачивается совершенно однозначно и недвусмысленно. А вот выполнение распределенной системы, напротив, обычно бывает недетерминированным, ввиду того что быстроедействие различных компонентов системы оказывается разным.

Рассмотрим ситуацию, когда процесс-сервер ожидает поступления заранее неизвестного числа запросов от клиентских процессов. Сервер не может отложить обработку запросов, пока они все не будут ему доставлены, поскольку он не знает, как много сообщений ему поступит. Значит, каждый запрос должен быть обработан немедленно, и обрабатываться запросы должны в порядке их поступления. Даже если известен порядок отправления этих запросов клиентами, задержки, возникающие при их передаче, никогда точно не известны, и поэтому запросы могут поступать совсем в другом порядке.

Все это вместе взятое — отсутствие сведений о глобальном состоянии, отсутствие глобальной шкалы времени и недетерминизм — приводит к тому, что задача проектирования распределенных алгоритмов становится весьма замысловатой, из-за того что эти три особенности могут встречаться в самых разнообразных сочетаниях.

Понятия *времени* и *состояния* очень тесно связаны друг с другом: в централизованных системах понятие времени вводится на основе *последовательности* состояний, которые достигаются по ходу выполнения программы. И хотя для распределенных систем можно определить понятие глобального состояния и выполнение программы можно мыслить как последовательность глобальных состояний (см. определение 2.2), подобная точка зрения представляется довольно ограниченной, поскольку то же самое выполнение может быть образовано другой последовательностью глобальных состояний (теорема 2.21). Такие альтернативные последовательности обычно состоят из разных глобальных состояний, и поэтому нет особого смысла говорить о том, что то или иное состояние достигается по ходу выполнения распределенной программы.

Отсутствие сведений о глобальном состоянии можно было бы возместить возможностью предвидеть достижимость этого состояния по ходу выполнения исходя из устройства алгоритма. К сожалению, и этого нельзя сделать из-за недетерминизма, неизбежно возникающего при выполнении распределенных систем.

1.3.2. Пример связи с передачей одного сообщения

Мы проиллюстрируем те затруднения, которые обусловлены отсутствием сведений о глобальном состоянии и отсутствием глобальной шкалы времени, на примере одной задачи надежного обмена информацией по ненадежному каналу связи, которая обсуждается в работе Бельснеса [26]. Рассмотрим два процесса *a* и *b*, связанных друг с другом в какой-либо сети, которая позволяет передавать данные от одного процесса к другому. Сообщение может быть доставлено сколько угодно поздно после своего отправления, а может и вообще потеряться в сети. Надежность коммуникации можно повысить, если воспользоваться процедурами управления сетью (NCPs¹⁷⁾), при помощи которых процессы *a* и *b* получают доступ к сети. Процесс *a* начинает устанавливать соединение, передавая одну единицу информации *m* процедуре *A*. Взаимодействие между процедурами управления сетью (через сеть передачи данных DN¹⁸⁾) должно быть организовано так, чтобы информация *m* была доставлена процессу *b* (при посредничестве NCP *B*), и после этого процесс должен быть оповещен об этой доставке (при посредничестве NCP *A*). Структура коммуникаций изображена на рис. 1.5.

Даже в том случае, когда процесс *a* должен передать процессу *b* только одну единицу информации, ненадежность сети вынуждает NCP *A* и NCP *B* вступить в диалог, который сопровождается обменом целым рядом сообщений. Эти процедуры сохраняют информацию о том, в каком состоянии находится этот диалог, но ввиду того что число партнеров, с которыми каждый процесс вступает в диалог, может быть очень велико, после завершения обмена сообщениями вся информация о состоянии диалога стирается. Инициализация информации о состоянии диалога называется *открытием* диалога, а сброс информации о его состоянии называется *закрытием* диалога. Нужно иметь в виду, что после закрытия диа-

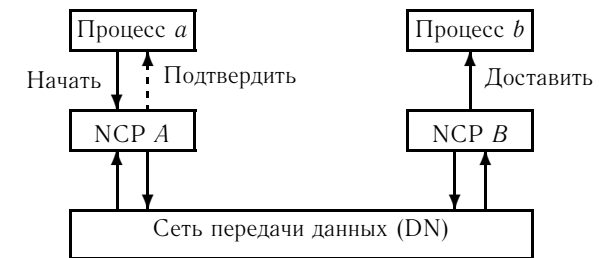


Рис. 1.5. Упрощенная архитектура сети

лога NCP перейдет как раз в то состояние, в котором эта процедура пребывала перед открытием этого диалога; это состояние мы будем называть *закрытым* состоянием.

Говорят, что блок информации *m* был *потерян*, если процесс *a* был оповещен о приеме этого блока процессом *b*, но сам блок при этом так и не был никогда доставлен процессу *b*. Говорят, что блок информации *m* был *дублирован*, если он была доставлен дважды. Надежное средство связи должно предотвращать как потери, так и дублирования. Предполагается также, что процедуры управления сетью могут выходить из строя (ломаться) и после каждой поломки они перезапускаются из закрытого состояния (при этом вся информация о текущем открытом диалоге будет потеряна).

Надежная связь невозможна. Прежде всего убедимся в том, что независимо от того, насколько изошренно устроены процедуры управления сетью, абсолютно надежную связь обеспечить невозможно. К этому выводу можно прийти независимо от устройства сети передачи данных и процедур управления этой сетью, основываясь лишь на предположении о том, что NCP может утрачивать информацию о ведущемся диалоге.

Чтобы убедиться в этом, предположим, что, после того как процесс *a* инициировал соединение, процедуры *A* и *B* вступают в диалог, в ходе которого NCP *B* должна доставить информацию *m* процессу *b* после получения сообщения *M* от NCP *A*. Рассмотрим случай, когда NCP *B* выходит из строя и перезапускается в закрытом состоянии, после того как NCP *A* уже отправила сообщение *M*. В этой ситуации ни NCP *A*, ни NCP *B* не могут сказать, была ли информация *m* уже доставлена в тот момент, когда NCP *B* вышла из строя. Процедура *A* не может этого сделать, потому что она не может судить о том, какие события произошли в NCP *B* (из-за отсутствия сведений о глобальном состоянии), а процедура *B* не может этого сделать, потому что она вышла из строя и была перезапущена в закрытом состоянии. Ошибка может возникнуть независимо от того, как процедуры управления сетью продолжают свой диалог. Если NCP *A* вновь отправит сообщение NCP *B* и NCP *B* доставит это сообщение, то произойдет дублирование информации. А если подтверждение *a* будет отправлено, тогда как доставка информации *m* не состоялась, то случится потеря информации.

¹⁷⁾ Network Control Procedures. — Прим. перев.

¹⁸⁾ Data Network. — Прим. перев.

А теперь мы проведем анализ некоторых вариантов устройства процедур управления и оценим, в какой мере эти процедуры способны терять и дублировать сообщения. Мы будем стараться построить протоколы так, чтобы они в любом случае позволяли избежать потери сообщений.

Диалог с одним сообщением. В самом простом случае NCP *A* после инициализации выполняет только одно действие, которое состоит в том, что *A* отправляет данные без всяких изменений, уведомляет об этом процесс *a* и закрывает сеанс связи. В свою очередь, NCP *B* доставляет полученное сообщение процессу *b* и после этого закрывает сеанс связи. При работе этого протокола потеря сообщения происходит всякий раз, когда сеть отказывается доставить сообщение по назначению, однако здесь нет никакой возможности для дублирования сообщения.

Диалог с двумя сообщениями. Чтобы обезопаситься хотя бы в какой-то мере от потери сообщений, в протокол вводят подтверждения. В ходе нормального развития события NCP *A* отправляет сообщение с данными $\langle \text{data}, m \rangle$ и ожидает получения подтверждающего сообщения $\langle \text{ack} \rangle$ от NCP *B*. Как только подтверждение будет получено, NCP *A* закрывает сеанс ¹⁹⁾. В свою очередь, NCP *B*, получив сообщение $\langle \text{data}, m \rangle$, доставляет информацию *m* процессу *b*, отправляет подтверждающее сообщение $\langle \text{ack} \rangle$ и закрывает сеанс связи. Таким образом, в этом диалоге, обнаруживающем ошибки, происходят три события.

1. NCP *A* send $\langle \text{data}, m \rangle$
2. NCP *B* receive $\langle \text{data}, m \rangle$, deliver *m*, send $\langle \text{ack} \rangle$, close
3. NCP *A* receive $\langle \text{ack} \rangle$, notify, close

В том случае, если по истечении определенного срока подтверждающее сообщение не будет получено, NCP *A*, принимая во внимание возможность потери сообщения, отправит сообщение $\langle \text{data}, m \rangle$ повторно. В связи с отсутствием осведомленности о глобальном состоянии NCP *A* не имеет сведений о том, было ли потеряно сообщение $\langle \text{data}, m \rangle$, или сообщение $\langle \text{ack} \rangle$, или, быть может, NCP *B* вышла из строя после получения сообщения $\langle \text{data}, m \rangle$ и не успела отправить подтверждение $\langle \text{ack} \rangle$. Для этого NCP *A* какое-то время ожидает поступления подтверждающего сообщения, и если такое сообщение не будет получено, то таймер обнаружит истечение срока ожидания (события такого рода называют *тайм-аутом*). Легко видеть, что вместе со способностью процедуры к повторной передаче сообщения в протокол привносится и возможность дублирования сообщения: это происходит в том случае, когда теряется не исходное сообщение, а подтверждение о его получении.

¹⁹⁾Естественно, уведомив об этом процесс *a*. — Прим. перев.

1. NCP *A* send $\langle \text{data}, m \rangle$
2. NCP *B* receive $\langle \text{data}, m \rangle$, deliver *m*, send $\langle \text{ack} \rangle$, close
3. DN $\langle \text{ack} \rangle$ is lost
4. NCP *A* timeout, send $\langle \text{data}, m \rangle$
5. NCP *B* receive $\langle \text{data}, m \rangle$, deliver *m*, send $\langle \text{ack} \rangle$, close
6. NCP *A* receive $\langle \text{ack} \rangle$, notify, close

Но введенные таким образом подтверждения не только приводят к дублированию сообщений; на самом деле они не способны предохранить от потери сообщения, о чем свидетельствует следующий сценарий. Процесс *a* намеревается передать две единицы информации *m*₁ и *m*₂.

1. NCP *A* send $\langle \text{data}, m_1 \rangle$
2. NCP *B* receive $\langle \text{data}, m_1 \rangle$, deliver *m*₁, send $\langle \text{ack} \rangle$, close
3. NCP *A* timeout, send $\langle \text{data}, m_1 \rangle$
4. NCP *B* receive $\langle \text{data}, m_1 \rangle$, deliver *m*₁, send $\langle \text{ack} \rangle$, close
5. NCP *A* receive $\langle \text{ack} \rangle$ (отправленный на шаге 4) notify, close
6. NCP *A* send $\langle \text{data}, m_2 \rangle$
7. DN $\langle \text{data}, m_2 \rangle$ is lost
8. NCP *A* receive $\langle \text{ack} \rangle$ (отправленный на шаге 2), notify, close

Так же как и в предыдущем сценарии выполнения протокола, сообщение *m*₁ дублируется, но первое подтверждающее сообщение здесь было не потеряно, а всего лишь доставлено с опозданием, и в результате этого второй блок информации *m*₂ был потерян. Замедление передачи сообщения обнаружить невозможно ввиду отсутствия глобальной шкалы времени.

Задачу надежной взаимосвязи процессов можно было бы решить гораздо проще, если бы в нашем распоряжении имелся ослабленный вариант шкалы глобального времени, а именно если бы существовала верхняя оценка *T* задержки передачи всякого сообщения по сети связи. Условия такого рода называются *глобальными* допущениями о времени, потому что эти допущения приводят к появлению определенных отношений по времени между событиями, происходящими в разных узлах сети (в данном случае между событием отправления сообщения NCP *A* и событием получения сообщения NCP *B*). В рассмотренном нами протоколе получение сообщений, относящихся к более ранним сеансам связи, можно предотвратить, если NCP *A* будет закрывать сеанс связи *2T* единиц времени спустя после получения последнего сообщения.

Диалог с тремя сообщениями. Коль скоро протокол с двумя сообщениями не может предотвратить потерю и дублирование информации в случае потери или задержки подтверждения, можно ввести в диалог третье сообщение, посредством которого NCP *B* уведомляется о том, что NCP *A* уже получила подтверждение. В нормальном случае диалог должен протекать по следующему сценарию.

1. NCP *A* send $\langle \text{data}, m \rangle$
2. NCP *B* receive $\langle \text{data}, m \rangle$, deliver *m*, send $\langle \text{ack} \rangle$
3. NCP *A* receive $\langle \text{ack} \rangle$, notify, send $\langle \text{close} \rangle$, close
4. NCP *B* receive $\langle \text{close} \rangle$, close

Потеря сообщения $\langle \text{data}, m \rangle$ приводит к тайм-ауту в NCP A, и в этом случае NCP A отправляет сообщение повторно. Потеря сообщения $\langle \text{ack} \rangle$ также приводит к повторному отправлению сообщения $\langle \text{data}, m \rangle$, но это не приводит к дублированию, потому что NCP B все еще имеет открытый сеанс связи, и поэтому процедура B способна распознать то сообщение, которое уже было получено ранее.

К сожалению, и этот протокол может терять и дублировать информацию. Так как NCP B обязательно должна обладать способностью закрывать сеанс связи, даже если будет потеряно сообщение $\langle \text{close} \rangle$, NCP B вынуждена передавать повторно сообщение $\langle \text{ack} \rangle$, если она не получила сообщение $\langle \text{close} \rangle$. Тогда NCP A откликается сообщением о том, что она не участвует в диалоге (сообщение $\langle \text{nocon} \rangle$), и после этого NCP B закрывает сеанс связи. Повторно переданное сообщение $\langle \text{ack} \rangle$ может, однако, возникнуть в следующем сеансе связи NCP A, и оно будет истолковано как подтверждающее сообщение, относящееся к текущему сеансу. В результате та информация, которая передается в этом сеансе связи, может быть потеряна, о чем свидетельствует следующий сценарий диалога процессов.

1. NCP A send $\langle \text{data}, m_1 \rangle$
2. NCP B receive $\langle \text{data}, m_1 \rangle$, deliver m_1 , send $\langle \text{ack} \rangle$
3. NCP A receive $\langle \text{ack} \rangle$, notify, send $\langle \text{close} \rangle$, close
4. DN $\langle \text{close} \rangle$ is lost
5. NCP A send $\langle \text{data}, m_2 \rangle$
6. DN $\langle \text{data}, m_2 \rangle$ is lost
7. NCP B retransmit $\langle \text{ack} \rangle$ (для шага 2)
8. NCP A receive $\langle \text{ack} \rangle$, notify, send $\langle \text{close} \rangle$, close
9. NCP B receive $\langle \text{close} \rangle$, close

И вновь мы сталкиваемся с проблемой, поскольку сообщение, относящееся к одному сеансу связи, может перемешиваться с сообщениями, относящимися к другому сеансу. Это можно предотвратить путем выбора пары новых сеансовых идентификационных номеров для каждого нового сеанса связи, один номер для NCP A, а другой — для NCP B. Выбранные номера включаются в состав всех сообщений, относящихся к этому сеансу связи, и служат для проверки того, что полученное сообщение относится к текущему сеансу. Нормальная работа протокола с тремя сообщениями протекает по следующему сценарию.

1. NCP A send $\langle \text{data}, m, x \rangle$
2. NCP B receive $\langle \text{data}, m, x \rangle$, deliver m , send $\langle \text{ack}, x, y \rangle$
3. NCP A receive $\langle \text{ack}, x, y \rangle$, notify, send $\langle \text{close}, x, y \rangle$, close
4. NCP B receive $\langle \text{close}, x, y \rangle$, close

При такой модификации протокола с тремя сообщениями ошибочный сеанс связи, описанный нами ранее, будет исключен, поскольку то сообщение, которое NCP A получает на шаге 8, не будет считаться подтверждением для сообщения с данными, отправленного на шаге 5. Однако NCP B не проверяет правильность сообщения $\langle \text{data}, m, x \rangle$, прежде чем доставляет информацию m (на шаге

2) процессу b , а это может легко привести к дублированию информации. Если сообщение, отправленное на шаге 1, было задержано и отправлено повторно, то сообщение $\langle \text{data}, m, x \rangle$, пришедшее позднее, вынуждает NCP B доставить информацию m процессу b еще раз.

Конечно, NCP B также должна проверять правильность сообщений, прежде чем доставлять данные процессу. Мы рассмотрим улучшенный вариант диалога с тремя сообщениями, в ходе которого NCP B доставляет данные на шаге 4, а не на шаге 2. Теперь NCP A отправляет уведомление, *прежде чем* NCP B доставит сообщение, но ввиду того, что NCP B уже получила переданную информацию, это вполне оправдано. Тем не менее, нужно убедиться, что NCP B будет доставлять данные в любом случае, и даже тогда, когда сообщение $\langle \text{close}, x, y \rangle$ может быть потеряно. NCP B повторно отправляет сообщение $\langle \text{ack}, x, y \rangle$, и на это NCP A откликается сообщением $\langle \text{nocon}, x, y \rangle$, в результате чего NCP B доставляет информацию процессу b и закрывает сеанс связи, как это видно из следующего сценария.

1. NCP A send $\langle \text{data}, m, x \rangle$
2. NCP B receive $\langle \text{data}, m, x \rangle$, send $\langle \text{ack}, x, y \rangle$
3. NCP A receive $\langle \text{ack}, x, y \rangle$, notify, send $\langle \text{close}, x, y \rangle$, close
4. DN $\langle \text{close}, x, y \rangle$ is lost
5. NCP B timeout, retransmit $\langle \text{ack}, x, y \rangle$
6. NCP A receive $\langle \text{ack}, x, y \rangle$, reply $\langle \text{nocon}, x, y \rangle$
7. NCP B receive $\langle \text{nocon}, x, y \rangle$, deliver m , close

Оказывается, для того чтобы избежать потери информации, NCP B должна доставлять данные даже тогда, когда NCP A не подтверждает своего участия в сеансе связи с идентификационными номерами x и y . Проверка правильности сообщений процедурой NCP B становится бесполезной, и это приводит к возможности дублирования информации, как видно из следующего сценария.

1. NCP A send $\langle \text{data}, m, x \rangle$
2. NCP A timeout, retransmit $\langle \text{data}, m, x \rangle$
3. NCP B receive $\langle \text{data}, m, x \rangle$ (отправленное на шаге 2), send $\langle \text{ack}, x, y_1 \rangle$
4. NCP A receive $\langle \text{ack}, x, y_1 \rangle$, notify, send $\langle \text{close}, x, y_1 \rangle$, close
5. NCP B receive $\langle \text{close}, x, y_1 \rangle$, deliver m , close
6. NCP B receive $\langle \text{data}, m, x \rangle$ (отправленное на шаге 1), send $\langle \text{ack}, x, y_2 \rangle$
7. NCP A receive $\langle \text{ack}, x, y_2 \rangle$, reply $\langle \text{nocon}, x, y_2 \rangle$
8. NCP B receive $\langle \text{nocon}, x, y_2 \rangle$ in reply to $\langle \text{ack}, x, y_2 \rangle$, deliver m , close

Диалог с четырьмя сообщениями. Доставку той информации, которая относилась к предшествующим сеансам связи, можно предотвратить, если процедуры управления сетью смогут прийти к договоренности относительно их сеансовых идентификационных номеров еще до того, как доставлены какие-нибудь данные. Рассмотрим следующий сценарий.

1. NCP A send $\langle \text{data}, m, x \rangle$
2. NCP B receive $\langle \text{data}, m, x \rangle$, send $\langle \text{open}, m, x, y \rangle$
3. NCP A receive $\langle \text{open}, m, x, y \rangle$, send $\langle \text{agree}, m, x, y \rangle$
4. NCP B receive $\langle \text{agree}, m, x, y \rangle$, deliver m , send $\langle \text{ack}, x, y \rangle$, close
5. NCP A receive $\langle \text{ack}, x, y \rangle$, notify, close

В связи с тем, что приходится учитывать возможность выхода из строя NCP B, исправление ошибок неизбежно будет проводиться так, что дублирование сообщений возникнет даже тогда, когда ни одна из NCP в действительности не выйдет из строя. NCP B отправляет сообщение об ошибке $\langle \text{nocon}, x, y \rangle$, когда получено сообщение $\langle \text{agree}, m, x, y \rangle$, но никакой сеанс связи не открыт. Предположим, что NCP A не получает сообщения $\langle \text{ack}, x, y \rangle$ даже после нескольких повторных отправок сообщения $\langle \text{agree}, m, x, y \rangle$; в ответ приходят только сообщения $\langle \text{nocon}, x, y \rangle$. Поскольку не исключена возможность выхода из строя NCP B перед тем, как она получила сообщение $\langle \text{agree}, m, x, y \rangle$, NCP A вынуждена открыть новый сеанс связи (отправив сообщение $\langle \text{data}, m, x \rangle$), чтобы предотвратить потерю информации m ! Но ведь NCP B могла с равным успехом уже доставить информацию m , и просто было потеряно сообщение $\langle \text{ack}, x, y \rangle$. В этом случае произойдет дублирование информации.

Можно модифицировать наш протокол таким образом, чтобы NCP A отправляла уведомление и закрывала сеанс связи после получения сообщения $\langle \text{nocon}, x, y \rangle$; это позволит избежать дублирования, но зато становится возможной потеря информации, что еще менее желательно.

Диалог с пятью сообщениями и сравнительный анализ. Бельснес в своей работе [26] описал протокол с пятью сообщениями, который предотвращает потерю информации и допускает дублирование только тогда, когда одна из NCP действительно выходит из строя. В свете того, что в этом параграфе нами была установлена невозможность абсолютно надежной коммуникации, этот протокол является самым лучшим из всех возможных. Но из-за слишком больших накладных расходов (NCP совершают пять обменов сообщениями, чтобы передать одну единицу информации) можно усомниться в том, что протокол с пятью сообщениями более предпочтителен, чем гораздо более простой протокол с двумя сообщениями. Действительно, ввиду того что даже протокол с пятью сообщениями может дублировать информацию (в случае выхода из строя одной из NCP), с этим типом ошибок нужно уметь как-то справляться на уровне процессов. Таким образом, вполне пригодным будет и наш протокол с двумя сообщениями, который может приводить к дублированию, но не допускает потери информации, если к сообщениям добавлять сеансовые идентификационные номера, как это было сделано для протокола с тремя сообщениями.

1.3.3. Область исследований

Интенсивное изучение распределенных алгоритмов продолжается уже более трех десятилетий. Наиболее значительный прогресс в исследованиях, определивший становление этого раздела информатики, был достигнут на рубеже вось-

мидесятых и девяностых годов. В предыдущих параграфах мы обратили внимание читателей на некоторые технические разработки (проектирование глобальных и локальных сетей, создание многопроцессорных компьютеров), которые стимулировали исследования в области распределенных алгоритмов. Первоначально эти исследования были в значительной мере сосредоточены на алгоритмах, предназначенных для глобальных сетей, но в настоящее время в этой области разработаны четкие математические модели, позволяющие применять полученные методы и результаты к более широкому классу распределенных систем. Тем не менее, исследователи в области распределенных алгоритмов пристально следят за техническими разработками в области телекоммуникаций, поскольку многие алгоритмические результаты зависят от изменений, которым подвергается сетевая модель. Например, с появлением общедоступных дешевых микропроцессоров открылась возможность построения систем, состоящих из большого числа совершенно идентичных процессоров, и это, в свою очередь, послужило стимулом к изучению «анонимных сетей» (см. гл. 9).

Существует несколько журналов и ежегодных конференций, тематика которых ограничена исключительно результатами в области распределенных алгоритмов и распределенных вычислений. В целом ряде других журналов и конференций, имеющих более широкую тематику, также представлено немало публикаций в этой области. Ежегодный симпозиум «Principles of Distributed Computing (PoDC)» проводится в Северной Америке начиная с 1982 г. и до настоящего времени; труды симпозиума публикуются Ассоциацией по вычислительной технике (ACM²⁰). Международная конференция «Workshops on Distributed Algorithms (WDAG)» проводилась впервые в Оттаве (1985), затем в Амстердаме (1987), в Ницце (1989), и, начиная с 1990 г. она стала ежегодным научным форумом, труды которого публикуются издательством Springer-Verlag в серии *Lecture Notes on Computer Science*. В 1998 г. эта конференция изменила название на «Distributed Computing» (DISC). Ежегодные симпозиумы по теории вычислений SToC²¹ и FoCS²² охватывают все фундаментальные аспекты теории вычислений, и нередко в трудах этих конференций появляются работы, посвященные распределенным вычислениям. Труды SToC публикуются Ассоциацией по вычислительной технике, а труды FoCS — Институтом инженеров по электротехнике и электронике. В журналах «Journal of Parallel and Distributed Computing (JPDC)», «Distributed Computing», а также «Information Processing Letters (IPL)» регулярно публикуются статьи, посвященные распределенным алгоритмам.

1.4. Обзор содержания книги

При написании этой книги автор преследовал три цели.

1. Ознакомить читателей с теми методами, которые применяются для изучения свойств заданных распределенных алгоритмов, для анализа и решения задач,

²⁰ Association for Computing Machinery — *Прим. перев.*

²¹ Symposium on the Theory of Computing. — *Прим. перев.*

²² Conference on the Foundations of Computer Science. — *Прим. перев.*

возникающих в связи с разработкой и использованием распределенных систем, или для оценки достоинств той или иной сетевой модели.

2. Дать читателю ясное представление о том, каковы возможности тех или иных моделей распределенных систем. В § 3.3.2, а также в гл. 12 и 15 рассказывается о том, какую роль играет глобальная шкала времени. В гл. 9 речь пойдет о том, насколько существенной оказывается осведомленность процессов об их отличительных признаках. В гл. 8 изучается вопрос о том, насколько существенным является требование завершаемости вычислений. Третья часть книги посвящена изучению проблем, возникающих в связи с возможностью выхода из строя отдельных процессов системы.

3. Рассказать о разнообразии современных распределенных алгоритмов, проведя при этом их верификацию, а также анализ их сложности.

Если какой-то вопрос нельзя осветить во всех подробностях, в книге приводятся ссылки на наиболее важные публикации по этому вопросу. Книга состоит из трех частей: «Протоколы», «Фундаментальные алгоритмы», «Отказоустойчивость».

Первая часть. Протоколы. Эта часть книги посвящена коммуникационным протоколам, которые используются при построении коммуникационных сетей для взаимосвязи компьютеров; здесь также описываются некоторые методы, которые затем используются в других частях книги.

В гл. 2 вводится модель, которая будет использоваться в большинстве последующих глав. Эта модель является, с одной стороны, достаточно общей для того, чтобы ее можно было с успехом применять при разработке и верификации алгоритмов, а с другой стороны, достаточно строгой для того, чтобы ею можно было воспользоваться для доказательства невозможности построения некоторых алгоритмов. В основу этой модели положено понятие системы переходов, которое сопровождается простыми правилами для доказательства свойств безопасности и живости. Также определяется отношение причинно-следственной зависимости, задающее частичный порядок на множестве событий вычисления, и вводятся логические часы.

В гл. 3 изучается проблема передачи сообщений между двумя узлами. Вначале мы ознакомим читателей с семейством протоколов обмена пакетами информации по единственному каналу связи и приведем, следуя работе Щуна, доказательство корректности этих протоколов. Затем рассматривается протокол Флетчера и Ватсона, корректность которого зависит от правильного использования таймеров. Изучение этого протокола позволяет узнать, как можно применять методы верификации к протоколам, в которых задействованы таймеры.

Глава 4 посвящена проблеме маршрутизации в вычислительных сетях. В первую очередь, в ней приведены некоторые теоретические предпосылки для решения задачи маршрутизации, а также описан алгоритм Туэга вычисления таблиц маршрутизации. Далее рассмотрен алгоритм Netchange, предложенный Таджикибаписом, и обоснование его корректности, предложенное Лампортом. В заключение описаны алгоритмы компактной маршрутизации, в том числе алгоритмы интервальной и префиксной маршрутизации. Эти алгоритмы маршрутизации называются

компактными алгоритмами маршрутизации, потому что в каждом узле сети им требуется совсем немного памяти.

Глава 5 завершает обсуждение протоколов для вычислительных сетей. В ней дается описание стратегий предотвращения блокировки хранения-продвижения в сетях с пакетной коммутацией пакетов. В основе этих стратегий лежит построение ациклических ориентированных графов на множестве буферов, принадлежащих узлам сети. Читатели узнают, как можно построить такие графы, используя сравнительно небольшое число буферов в каждом узле.

Вторая часть. Фундаментальные алгоритмы. В этой части книги представлен широкий набор алгоритмических «строительных блоков», которые используются в качестве процедур во многих распределенных прикладных программах, и приведены теоретические результаты, устанавливающие ту роль, которая отводится в вычислениях различным допущениям об устройстве сети.

В гл. 6 определяется понятие «волнового алгоритма», представляющего собой обобщенную схему посещения всех узлов сети. При помощи волновых алгоритмов можно распространять информацию по всей сети, проводить синхронизацию ее узлов, вычислять функцию, значение которой зависит от информации, распределенной между всеми узлами сети. Как станет ясно из последующих глав, многие задачи распределенного управления можно решать при помощи алгоритмических схем достаточно общего вида, использующих волновой алгоритм в качестве одного из своих компонентов. В этой главе также дается определение сложности по времени для распределенных алгоритмов и исследуется сложность по времени и по числу обменов сообщениями целого ряда распределенных алгоритмов поиска в глубину.

Одной из фундаментальных проблем, возникающих в распределенных системах, является *задача избрания лидера*: требуется выбрать один-единственный процесс, которому отводится особая роль в последующем вычислении. Эта задача изучается в гл. 7. Вначале мы ограничимся случаем кольцевых сетей и покажем, что для этой задачи сложность по числу обменов сообщениями составляет величину $\Theta(N \log N)$ (для колец с N процессорами). Далее задача избрания лидера рассматривается для сетей общего вида. Мы опишем ряд конструкций, позволяющих построить алгоритмы избрания лидера на основе волновых алгоритмов и алгоритмов обхода сети. В этой главе также обсуждается алгоритм построения остова дерева, предложенный Галаджером и др.

Другой фундаментальной проблемой является *задача обнаружения завершения вычисления*: процессам системы требуется установить, завершилось ли распределенное вычисление. Эта задача изучается в гл. 8. Мы установим нижнюю оценку сложности решения этой задачи и подробно обсудим различные алгоритмы ее решения. В этой главе рассматриваются несколько классических алгоритмов (в том числе алгоритм Дейкстры, Фейджена и Ван Гастерена, а также алгоритм Дейкстры и Шолтена) и, как и в предыдущей главе, описывается способ получения алгоритмов решения этой задачи на основе волновых алгоритмов.

Глава 9 посвящена изучению вычислительных возможностей систем, процессы которых не имеют индивидуальных отличительных признаков. Как было уста-

новлено Англуином, в этом случае многие вычислительные задачи нельзя решить при помощи детерминированных алгоритмов. В этой главе мы введем вероятностные алгоритмы и исследуем, какие задачи можно решить при помощи этих алгоритмов.

В главе 10 рассказывается о том, как процессы системы могут сформировать глобальную «картину вычисления», т. е. так называемое моментальное состояние системы. Моментальное состояние системы полезно иметь для определения различных свойств вычисления. С его помощью можно выяснить, не возникла ли блокировка, насколько далеко продвинулось вычисление и др. Моментальные состояния применяются также для перезапуска вычисления в случае возникновения ошибки.

В гл. 11 мы рассмотрим некоторые эффекты, которые возникают в том случае, когда процессам становится доступна информация об их положении в сети. Мы опишем также алгоритмы, позволяющие вычислять сведения такого рода.

В гл. 12 рассматриваются эффекты, которые возникают в том случае, когда процессам системы становится доступна шкала отсчета глобального времени. Мы определим несколько степеней синхронизации и покажем, что вполне асинхронные системы способны воспроизводить поведение синхронных систем при помощи очень простых алгоритмов. Это свидетельствует о том, что допущение о синхронизации никоим образом не влияет на класс тех функций, которые могут быть вычислены распределенными системами. Однако вслед за этим мы покажем, что указанное допущение оказывает существенное влияние на коммуникационную сложность решения многих задач: чем более синхронизованной является сеть, тем меньшую сложность имеют алгоритмы решения этих задач.

Третья часть. Отказоустойчивость. При использовании распределенных систем на практике нельзя обойти вниманием возможность того, что некоторые компоненты системы могут выходить из строя. Поэтому важно знать, как будут вести себя алгоритмы при возникновении сбоев в работе отдельных компонентов системы. Это как раз и является предметом изучения последней части нашей книги. В гл. 13 предложено краткое введение в изучаемую предметную область.

В гл. 14 изучаются вопросы отказоустойчивости асинхронных распределенных систем. Приводится результат, полученный Фишером и др., который свидетельствует о том, что детерминированные асинхронные системы не могут справиться даже со сравнительно простыми неисправностями, как то: выход из строя одного-единственного процесса. Но также будет показано, что последствия неисправностей более слабого типа можно устранить и некоторые задачи разрешимы, несмотря на выход из строя некоторых компонентов системы. Мы представим алгоритмы Брачи и Туэга, которые показывают, что рандомизованные асинхронные системы способны справиться с достаточно большим числом неисправностей. Таким образом, мы видим, что, как и в случае с надежными системами, описанными в гл. 9, вычислительные возможности рандомизованных алгоритмов превосходят возможности детерминированных алгоритмов.

В гл. 15 изучаются вопросы отказоустойчивости синхронных систем. Алгоритмы, разработанные Лампортом и др., демонстрируют устойчивость детерми-

нированных синхронных алгоритмов по отношению к сложным неисправностям. Таким образом, мы видим, что, в отличие от надежных систем, рассмотренных в гл. 12, в случае возникновения неисправностей вычислительные возможности синхронных систем превосходят возможности асинхронных систем. Системы можно сделать устойчивыми по отношению к еще большему числу неисправностей, если разрешить процессам «подписывать» те сообщения, которые они отправляют другим процессам. Это означает, что реализация механизма синхронизации в ненадежных системах является еще более сложной задачей, нежели осуществление этого в надежных системах. Этой задаче посвящена гл. 15.

В гл. 16 изучаются свойства общих приемов (их называют *детекторами неисправностей*), которые позволяют процессам системы оценить, какие процессы вышли из строя, а какие остались исправными. Мы расскажем о том, как можно реализовать эти приемы, а также о том, как их можно использовать для реализации спецификаций в неисправной вычислительной среде.

В гл. 17 рассматривается другой подход к повышению надежности систем за счет использования самостабилизирующихся алгоритмов. Алгоритм называется стабилизирующимся, если независимо от начальной конфигурации, с которой начинается его вычисление, оно рано или поздно сходится к требуемому поведению. Мы разработаем некоторые элементы теории стабилизирующихся алгоритмов и приведем ряд примеров стабилизирующихся алгоритмов. К числу этих алгоритмов относятся протоколы для некоторых алгоритмов на графах (вычисления деревьев поиска в глубину, описанные в § 6.6.4, вычисления таблиц маршрутизации описанные в гл. 4). Кроме того, были предложены стабилизирующиеся алгоритмы для задачи передачи данных, которая рассматривается в гл. 3, и это означает, что всякую вычислительную сеть можно реализовать при помощи стабилизирующихся алгоритмов.

Приложения. В гл. А объясняется та система обозначений, которая используется в этой книге для описания распределенных алгоритмов. В гл. Б содержится ряд общих сведений из теории графов, включая используемую в этой книге терминологию из теории графов. В конце книги приводится список литературы и предметный указатель.

ГЛАВА 2

МОДЕЛЬ

При изучении распределенных алгоритмов наиболее часто используются несколько различных моделей распределенной обработки информации. Конкретная модель обычно выбирается в зависимости от исследуемой задачи из области распределенных вычислений, а также от представленной разновидности алгоритма (или доказательства отсутствия такового). Хотя в этой книге рассматривается широкий спектр распределенных алгоритмов и относящихся к ним теоретических результатов, автор предпринял попытку представить весь материал, насколько это возможно, в рамках одной-единственной общей модели, описанной в этой главе.

Для того чтобы позволить получать отрицательные результаты (доказательство невозможности построения алгоритмов решения той или иной задачи), модель должна быть очень точной. Отрицательный результат — это утверждение, относящееся ко *всем* возможным алгоритмам системы; поэтому модель должна быть настолько точна, чтобы в ее рамках можно было описать существенные свойства всех допустимых алгоритмов. И все же вычислительная модель — это нечто большее, чем подробное описание конкретной вычислительной машины или языка программирования. Существует много разных вычислительных систем, и нам хотелось бы, чтобы выбранная модель была применима к целому *классу* родственных систем, наделенных теми специальными важными качествами, благодаря которым эти системы становятся «распределенными». Наконец, выбранная модель должна быть достаточно компактной, поскольку мы вправе ожидать, что ее можно будет всесторонне рассмотреть при проведении доказательств. Таким образом, модель должна *точно* и *кратко* описывать наиболее существенные аспекты некоторого *класса* вычислительных систем.

Распределенное вычисление обычно рассматривается в виде совокупности дискретных событий, каждое из которых представляет собой небольшое изменение конфигурации (состояния всей системы). В §2.5.1 мы определим распределенные вычисления на основе понятия *системы переходов*, которое приведет нас к понятию достижимой конфигурации и к конструктивному определению множества выполнений, порождаемых алгоритмом. Система становится «распределенной» благодаря тому обстоятельству, что при выполнении каждого перехода приходится иметь дело только с *частью* конфигурации, а именно с (локальным) состоянием одного процесса (или с локальными состояниями некоторого подмножества взаимодействующих процессов.)

Сама модель определена в §2.5.1; в §2.5.2 и 2.5.3 рассматриваются ее основные свойства. Параграф 2.5.2 посвящен вопросам построения доказательств для

обоснования требуемых свойств заданных распределенных алгоритмов. В §2.5.3 обсуждается такое важное понятие, как отношение причинно-следственной связи между событиями при выполнении алгоритмов. Это отношение дает возможность ввести отношение эквивалентности на всем множестве выполнений; по определению вычислением в этом случае называется класс эквивалентности выполнений. Определяется понятие часов, и в качестве первого распределенного алгоритма рассматривается алгоритм логических часов. В заключительном §2.4 обсуждаются понятия и обозначения, оставшиеся за пределами базовой модели.

2.1. Системы переходов и алгоритмы

Обычно системы, состояния которых изменяются дискретными шагами (эти шаги называются переходами или событиями), удобно описывать с привлечением понятия *системы переходов*. При изучении распределенных алгоритмов это понятие применяется как ко всей распределенной системе в целом, так и к отдельным процессам, которые взаимодействуют в алгоритме. Поэтому концепция системы переходов очень важна при изучении распределенных алгоритмов, и ее определению посвящен §2.1.1.

В распределенных системах переходы оказывают влияние только на часть конфигурации (глобального состояния системы). Каждая конфигурация представляется некоторым кортежем, и в каждом переходе задействованы только некоторые компоненты этого кортежа. К числу компонентов конфигурации относятся состояния каждого процесса. Для точного определения конфигурации необходимо выделить различные типы распределенных систем в зависимости от способа организации взаимосвязи между процессами.

Процессы в распределенной системе взаимодействуют либо за счет доступа к *разделяемым переменным*, либо посредством *обменов сообщениями*. Наше изложение не будет столь всеобъемлющим, и мы ограничимся рассмотрением только таких распределенных систем, в которых процессы взаимодействуют посредством обменов сообщениями. Распределенные системы, в которых взаимодействие достигается с использованием разделяемых переменных, будут рассмотрены в гл. 17. Читатель, интересующийся взаимодействием посредством разделяемых переменных, может обратиться к замечательной работе Дейкстры [60] или к работе Овицкого и Гриса [155]; из недавних работ по этой теме можно выделить книгу Атты и Велча [12].

В распределенных системах сообщения могут передаваться либо *синхронно*, либо *асинхронно*. В этой книге основное внимание уделено алгоритмам для систем, в которых сообщения передаются асинхронно. Как было показано в работе Чаррона-Боста и его коллег [48], во многих случаях синхронный обмен сообщениями можно рассматривать как специальную разновидность асинхронного обмена сообщениями. В §2.1.2 дано строгое описание модели асинхронного обмена сообщениями; в §2.1.3 эта модель адаптирована к системам, использующим синхронный обмен сообщениями. В §2.1.4 мы кратко обсудим такое явление, как справедливость.

2.1.1. Системы переходов

Система переходов состоит из множества всех возможных состояний системы, переходов («действий»), которые система может совершать в этих состояниях, и подмножества состояний, в которых система может пребывать в самом начале. Чтобы не путать состояния отдельных процессов и состояния всего алгоритма (так называемые «глобальные состояния»), условимся в дальнейшем называть глобальные состояния *конфигурациями*.

Определение 2.1. *Системой переходов* называется тройка $S = (\mathcal{C}, \rightarrow, \mathcal{I})$, в которой $\mathcal{C}$ — это множество *конфигураций*, $\rightarrow$ — двуместное *отношение переходов* на $\mathcal{C}$, и $\mathcal{I}$ — некоторое подмножество множества $\mathcal{C}$, элементы которого называются *начальными конфигурациями*.

Отношение переходов — это некоторое подмножество декартова произведения $\mathcal{C} \times \mathcal{C}$. Однако вместо записи $(\gamma, \delta) \in \rightarrow$ мы будем использовать более удобное обозначение $\gamma \rightarrow \delta$.

Определение 2.2. Пусть $S = (\mathcal{C}, \rightarrow, \mathcal{I})$ — это система переходов. *Выполнением* системы S назовем максимальную последовательность $E = (\gamma_0, \gamma_1, \gamma_2, \dots)$, в которой $\gamma_0 \in \mathcal{I}$ и для каждого i , $i \geq 0$, выполняется отношение $\gamma_i \rightarrow \gamma_{i+1}$.

Заключительной конфигурацией назовем такую конфигурацию γ , для которой не существует конфигурации δ , удовлетворяющей отношению $\gamma \rightarrow \delta$. Следует заметить, что последовательность $E = (\gamma_0, \gamma_1, \gamma_2, \dots)$, в которой для каждого i выполняется отношение $\gamma_i \rightarrow \gamma_{i+1}$, будет максимальной, если она либо является бесконечной, либо оканчивается заключительной конфигурацией.

Определение 2.3. Конфигурация δ считается *достижимой* из конфигурации γ (для обозначения этого будем использовать запись $\gamma \rightsquigarrow \delta$), если существует последовательность конфигураций $\gamma_0, \gamma_1, \gamma_2, \dots, \gamma_k$, в которой $\gamma = \gamma_0$, $\gamma_k = \delta$ и для каждого i , $0 \leq i < k$, выполняется отношение $\gamma_i \rightarrow \gamma_{i+1}$. Всякая конфигурация δ , достижимая из некоторой начальной конфигурации, будет называться просто *достижимой*.

2.1.2. Системы с асинхронным обменом сообщениями

Распределенная система состоит из совокупности *процессов* и *коммуникационной подсистемы*. Каждый процесс сам по себе является системой переходов, с тем лишь условием, что он может взаимодействовать с коммуникационной подсистемой. Чтобы избежать терминологической путаницы между свойствами, присущими всей распределенной системе в целом, и атрибутами отдельных процессов, мы будем придерживаться следующего соглашения. Термины «переход» и «конфигурация» будут использоваться применительно ко всей распределенной системе в целом, тогда как термины «событие» и «состояние», обозначающие равносильные понятия, будут использоваться, если речь идет о процессах системы. Чтобы взаимодействовать с коммуникационной подсистемой, в распоряжении процесса помимо обычных событий (которые мы в дальнейшем будем называть

внутренними событиями) имеются также *события отправления* и *события приема*, которые порождают или изымают сообщения. Обозначим символом $\mathcal{M}$ множество всех возможных *сообщений*; для обозначения совокупности всех мультимножеств с элементами из $\mathcal{M}$ будем использовать запись $\mathbb{M}(\mathcal{M})$.

Определение 2.4. *Локальным алгоритмом* процесса называется пятерка $(Z, I, \vdash^i, \vdash^s, \vdash^r)$, в которой Z — множество *состояний*, I — это некоторое подмножество множества Z , элементы которого называются *начальными состояниями*, $\vdash^i$ — отношение на множестве $Z \times Z$, а $\vdash^s$ и $\vdash^r$ — отношения на множестве $Z \times \mathcal{M} \times Z$. Двуместное отношение $\vdash$ на множестве Z определяется соотношением

$$c \vdash d \iff (c, d) \in \vdash^i \vee \exists m \in \mathcal{M} ((c, m, d) \in \vdash^s \cup \vdash^r).$$

Отношения $\vdash^i$, $\vdash^s$ и $\vdash^r$ представляют переходы между состояниями, связанные соответственно с внутренними событиями, с событиями передачи сообщений и с событиями приема сообщений. Далее мы будем обозначать процессы буквами p , q , r , p_1 , p_2 , и т. п., а множество процессов системы будем обозначать буквой $\mathbb{P}$. Определение 2.4 будет служить нам теоретической моделью процессов. В этой книге при описании алгоритмов мы не будем, конечно, заниматься перечислением состояний и событий, а воспользуемся удобным псевдокодом (см. гл. А). Выполнениями процесса будут называться выполнения системы переходов $(Z, \vdash, I)$.

Нас, однако, будут интересовать выполнения всей системы в целом, и в каждом таком выполнении коммуникационная система будет координировать выполнения отдельных процессов. Для того чтобы описать, как осуществляется эта координация, мы будем рассматривать распределенную систему как систему переходов, в которой множество конфигураций, отношение переходов и начальные состояния формируются из соответствующих компонентов процессов.

Определение 2.5. *Распределенным алгоритмом* для семейства процессов $\mathbb{P} = \{p_1, \dots, p_N\}$ будет называться совокупность локальных алгоритмов, каждый из которых соответствует в точности одному процессу из $\mathbb{P}$.

Поведение распределенного алгоритма описывается посредством системы переходов следующего вида. Каждая конфигурация системы определяется состояниями всех процессов, а также совокупностью сообщений, которые еще не достигли адресатов. Переходами являются все события процессов, которые не только изменяют состояния самих процессов, но способны как оказывать влияние на совокупность сообщений, так и претерпевать воздействия этой совокупности сообщений. Начальными конфигурациями считаются все те конфигурации, в которых все процессы пребывают в начальных состояниях, а множество сообщений пусто.

Определение 2.6. Пусть задано семейство процессов $p_1, \dots, p_N$ и локальный алгоритм каждого процесса p_i представлен пятеркой $(Z_{p_i}, I_{p_i}, \vdash_{p_i}^i, \vdash_{p_i}^s, \vdash_{p_i}^r)$. Тогда будем говорить, что система переходов $S = (\mathcal{C}, \rightarrow, \mathcal{I})$ *порождена распределенным алгоритмом для семейства асинхронно взаимосвязанных процессов* $p_1, \dots, p_N$, если выполнены следующие соотношения:

$$1) \mathcal{C} = \{(c_{p_1}, \dots, c_{p_N}, M) : (\forall p \in \mathbb{P} : c_p \in Z_p) \text{ и } M \in \mathbb{M}(\mathcal{M})\};$$

2) $\rightarrow = (\cup_{p \in \mathbb{P}} \rightarrow_p)$, где символом $\rightarrow_p$ обозначены переходы, соответствующие изменению состояния процесса p ; иначе говоря, $\rightarrow_{p_i}$ — это множество всех пар вида

$$(c_{p_1}, \dots, c_{p_i}, \dots, c_{p_N}, M_1), (c_{p_1}, \dots, c'_{p_i}, \dots, c_{p_N}, M_2),$$

для которых выполняется одно из трех условий:

- $(c_{p_i}, c'_{p_i}) \in \vdash_{p_i}^i$ и $M_1 = M_2$;
- для некоторого сообщения $m \in \mathcal{M}$ имеется событие $(c_{p_i}, m, c'_{p_i}) \in \vdash_{p_i}^s$, и при этом $M_2 = M_1 \cup \{m\}$;
- для некоторого сообщения $m \in \mathcal{M}$ имеется событие $(c_{p_i}, m, c'_{p_i}) \in \vdash_{p_i}^r$, и при этом $M_1 = M_2 \cup \{m\}$;

4) $\mathcal{I} = \{(c_{p_1}, \dots, c_{p_N}, M) : (\forall p \in \mathbb{P} : c_p \in I_p) \wedge M = \emptyset\}$.

Выполнением распределенного алгоритма будет называться всякое выполнение системы переходов, порожденной этим алгоритмом. Чтобы иметь возможность явно описывать события того или иного выполнения, мы введем следующие обозначения. Пары $(c, d) \in \vdash_p^i$ будем называть (возможными) *внутренними событиями* процесса p , а тройки, из которых состоят отношения $\vdash_p^s$ и $\vdash_p^r$, будем называть событиями *отправления* и *приема* (сообщения) соответствующим процессом.

- Внутреннее событие e процесса p , представленное парой $e = (c, d)$, считается *допустимым* в конфигурации $\gamma = (c_{p_1}, \dots, c_p, \dots, c_{p_N}, M)$, если $c_p = c$. Тогда запись $e(\gamma)$ будет обозначать конфигурацию $(c_{p_1}, \dots, d, \dots, c_{p_N}, M)$.
- Событие e отправления сообщения процессом p , представленное тройкой $e = (c, m, d)$, считается *допустимым* в конфигурации $\gamma = (c_{p_1}, \dots, c_p, \dots, c_{p_N}, M)$, если $c_p = c$. Тогда запись $e(\gamma)$ будет обозначать конфигурацию $(c_{p_1}, \dots, d, \dots, c_{p_N}, M \cup \{m\})$.
- Событие e приема сообщения процессом p , представленное тройкой $e = (c, m, d)$, считается *допустимым* в конфигурации $\gamma = (c_{p_1}, \dots, c_p, \dots, c_{p_N}, M)$, если $c_p = c$ и $m \in M$. Тогда запись $e(\gamma)$ будет обозначать конфигурацию $(c_{p_1}, \dots, d, \dots, c_{p_N}, M \setminus \{m\})$.

Предполагается, что для каждого сообщения имеется единственный процесс, который может получать это сообщение. Этот процесс будем называть *адресатом* данного сообщения.

2.1.3. Системы с синхронным обменом сообщениями

Обмен сообщениями называется *синхронным*, если событие отправления сообщения и соответствующее ему событие приема сообщения скоординированы так, что образуют единый переход в системе. Иначе говоря, процессу не дозволено отправлять сообщение, до тех пор пока адресат этого сообщения не будет готов к его приему. В соответствии с этим переходы в системе разделяются на два типа: переходы первого типа связаны с изменением внутренних состояний процесса, а переходы второго типа связаны с комбинированным осуществлением событий отправления-приема сообщения двумя процессами.

Определение 2.7. Пусть задано семейство процессов $p_1, \dots, p_N$. Тогда будем говорить, что система переходов $S = (\mathcal{C}, \rightarrow, \mathcal{I})$ порождена *распределенным алгоритмом для семейства процессов $p_1, \dots, p_N$ при их синхронной взаимосвязи*, если выполнены соотношения:

- 1) $\mathcal{C} = \{(c_{p_1}, \dots, c_{p_N}) : \forall p \in \mathbb{P} : c_p \in Z_p\}$;
- 2) $\rightarrow = (\cup_{p \in \mathbb{P}} \rightarrow_p) \cup (\cup_{p, q \in \mathbb{P} : p \neq q} \rightarrow_{pq})$, где

- $\rightarrow_{p_i}$ представляет собой множество пар

$$(c_{p_1}, \dots, c_{p_i}, \dots, c_{p_N}), \dots, (c_{p_1}, \dots, c'_{p_i}, \dots, c_{p_N}),$$

для которых справедливо включение $(c_{p_i}, c'_{p_i}) \in \vdash_{p_i}^i$;

- $\rightarrow_{p_i p_j}$ представляет собой множество пар

$$(\dots, c_{p_i}, \dots, c_{p_j}, \dots), \dots, (\dots, c'_{p_i}, \dots, c'_{p_j}, \dots),$$

для которых существует такое сообщение $m \in \mathcal{M}$, что имеют место включения

$$(c_{p_i}, m, c'_{p_i}) \in \vdash_{p_i}^s \text{ и } (c_{p_j}, m, c'_{p_j}) \in \vdash_{p_j}^r;$$

3) $\mathcal{I} = \{(c_{p_1}, \dots, c_{p_N}) : (\forall p \in \mathbb{P} : c_p \in I_p)\}$.

В некоторых распределенных системах допускается смешанная форма взаимосвязи; процессы в таких системах имеют примитивы для совершения обменов сообщениями как в синхронном, так и в асинхронном режиме. Располагая теми двумя моделями, которые были определены выше, нетрудно построить модель и для этого типа распределенных систем. Конфигурации для таких систем определяются состояниями процессов и совокупностью сообщений, пребывающих на этапе (асинхронной) пересылки. Переходы системы включают в себя все типы переходов, представленные в определениях 2.6 и 2.7.

Синхронизация и ее влияние на алгоритмы. Как уже было отмечено, при решении многих задач синхронный обмен сообщениями можно рассматривать как особый случай асинхронного обмена сообщениями. Множество выполнений в случае синхронного обмена сообщениями ограничивается лишь теми выполнениями, в которых за событием отправления сообщения непосредственно следует соответствующее ему событие приема сообщения (см. [48]). Поэтому мы считаем асинхронный обмен сообщениями более общим видом модели, нежели синхронный обмен сообщениями, и в дальнейшем мы будем заниматься разработкой алгоритмов преимущественно для этого более общего случая.

Тем не менее, нужно проявить определенную аккуратность, если алгоритм, разработанный в расчете на асинхронный обмен сообщениями, выполняется в системе с синхронным обменом сообщениями. Сокращение недетерминизма в коммуникационной подсистеме должно быть уравновешено возрастанием недетерминизма в процессах системы, ибо в противном случае это может привести к возникновению взаимной блокировки.

В качестве иллюстрации мы рассмотрим простой пример, в котором два процесса посылают друг другу некоторую информацию. В асинхронном случае каждый процесс может вначале отправить сообщение, а затем получить сообщение

от другого процесса. На отрезке времени между отправлением и получением сообщения коммуникационная подсистема выступает в роли буфера, в который поступают сообщения. В синхронном случае такая буферизация невозможна, и если оба процесса должны отправить свои сообщения, прежде чем они приступят к их приему, никаких переходов не может произойти. В синхронном случае один из процессов обязан принять сообщение от другого процесса, перед тем как отправить свое сообщение. Очевидно, что если оба процесса должны провести прием сообщения прежде, чем отправить свои собственные сообщения, то никаких переходов произойти не может.

Обмен двумя сообщениями в синхронных системах может совершиться только в том случае, если выполнено одно из следующих двух условий.

1. Заранее определено, который из двух процессов будет отправлять сообщение первым и который из них будет вначале заниматься приемом сообщения. Предопределить такой выбор заранее во многих случаях бывает невозможно, поскольку для этого требуется, чтобы эти два процесса выполняли разные локальные алгоритмы.

2. Процессам предоставлена возможность недетерминированного выбора: либо вначале отправить сообщение, а затем заняться приемом сообщения, либо вначале совершить прием сообщения, а уже затем приступить к отправлению сообщения. При каждом выполнении коммуникационной подсистемой будет выбран один из возможных порядков действия для каждого процесса, т. е. симметрия будет нарушена коммуникационной подсистемой.

Когда мы даем описание какого-либо алгоритма, рассчитанного на асинхронный обмен сообщениями, и утверждаем, что этот алгоритм может быть использован и в системах с синхронным обменом сообщениями, мы тем самым неявно предполагаем, что для этого в алгоритме нужно ввести указанный недетерминизм, и это всегда возможно.

2.1.4. Справедливость

Иногда при исследовании поведения систем возникает необходимость ограничиться рассмотрением только так называемых *справедливых* выполнений. Условия справедливости позволяют исключить из рассмотрения такие выполнения, в которых некоторые события оказываются допустимыми всегда (или бесконечно часто), но при этом ни разу не осуществляются в виде переходов (из-за того что выполнение продолжается всякий раз за счет осуществления других событий).

Определение 2.8. Выполнение считается *слабо справедливым*, если никакое событие не может оказаться допустимым в каждой из бесконечного числа подряд идущих конфигураций, не осуществившись при этом ни разу в этом выполнении. Выполнение считается *сильно справедливым*, если никакое событие, не может оказаться допустимым в бесконечном множестве конфигураций, не осуществившись при этом ни разу в этом выполнении¹⁾.

¹⁾Приведенное здесь определение слабой и сильной справедливости не является достаточно строгим. Более корректное определение этих понятий таково:

Условия справедливости можно включить в явном виде в состав определения формальной модели, как это было сделано в работе Манни и Пнуели [138]. Большинство алгоритмов, с которыми нам предстоит иметь дело в настоящей книге, не зависят от этих условий; поэтому мы предпочли не включать их в состав модели, а вместо этого будем явно оговаривать эти условия всякий раз, когда они используются в том или ином алгоритме или задаче. Вопрос о целесообразности включения допущений справедливости в состав модели распределенной системы вызвал немало споров. Сложилось убеждение, что на допущения справедливости не следует рассчитывать; вместо этого сами алгоритмы должны быть устроены так, чтобы они не зависели от этих допущений. Обсуждение некоторых замысловатых аспектов, связанных с допущениями справедливости, можно найти в работе [89].

2.2. Как обосновывать свойства систем переходов

Если имеется распределенный алгоритм для некоторой задачи, то необходимо убедиться в том, что данный алгоритм правильно решает эту задачу. Исходя из самой задачи можно определить, какими свойствами должен обладать требуемый алгоритм ее решения; после этого нужно показать, что предложенное решение обладает этими свойствами. Проблема верификации распределенных алгоритмов привлекла большое внимание, и существует много работ, в которых обсуждаются формальные методы верификации (см, например, [46, 89, 115, 138]). В этом параграфе мы обсудим ряд простых, но при этом наиболее часто используемых методов обоснования корректности распределенных алгоритмов. Эти методы основываются только на определении понятия системы переходов.

Многие свойства распределенных алгоритмов, нуждающиеся в проверке, относятся к одному из двух типов: *условие безопасности* и *условие живости*. Условие безопасности требует, чтобы *каждая* достижимая конфигурация в любом выполнении системы обладала определенным свойством. Условие живости требует, чтобы *хотя бы одна* достижимая конфигурация в любом выполнении системы обладала определенным свойством. Формальное изучение вопроса о том, как устроены свойства безопасности и живости, было предпринято в работе Альперна и Шнейдера [6]. Они установили, что *каждая возможная* спецификация (свойство множества выполнений) может быть представлена в виде конъюнкции свойств безопасности и живости.

Эти условия могут возникнуть также и в более слабой форме; например, может потребоваться, чтобы они соблюдались с некоторой заданной вероятностью на множестве всех возможных выполнений. Другие условия, которым должны

1). каждое выполнение, оканчивающееся заключительной конфигурацией, является справедливым (как в слабом, так и в сильном смысле);

2). бесконечное выполнение $E = (\gamma_0, \gamma_1, \gamma_2, \dots)$ считается *слабо справедливым*, если не существует такого натурального числа n , $n > 0$, и такого события e , что для любого i , $i \geq n$, событие e допустимо в конфигурации γ_i , но при этом $\gamma_{i+1} \neq e(\gamma_i)$;

3). бесконечное выполнение $E = (\gamma_0, \gamma_1, \gamma_2, \dots)$ считается *сильно справедливым*, если не существует такого натурального числа n , $n > 0$, и такого события e , что для любого i , $i \geq n$, событие e допустимо в некоторой конфигурации γ_j , где $j \geq i$, но при этом $\gamma_{i+1} \neq e(\gamma_i)$. — Прим. перев.

удовлетворять алгоритмы, могут выражать требования о том, чтобы в этих алгоритмах использовались только сведения определенного рода (см. § 2.4.4), чтобы алгоритмы были устойчивы к сбоям в отдельных процессах (см. гл. 13), чтобы процессы были неотличимы (см. гл. 9), и т. п.

Методы верификации, описанные в этом параграфе, основываются на истинности *утверждений* о конфигурациях, достижимых по ходу выполнения. Такого рода методы называются *ассертивными*²⁾ методами верификации. Под утверждением понимается одноместное отношение на множестве конфигураций, т. е. предикат, который является истинным на множестве одних конфигураций и ложным на множестве других.

2.2.1. Свойства безопасности

Свойством безопасности алгоритма является всякое свойство, которое можно сформулировать в виде следующего предложения: «Для любого выполнения алгоритма утверждение P истинно в каждой конфигурации выполнения». Более кратко это свойство можно сформулировать так: «Утверждение P всегда истинно». Основной прием, при помощи которого удается показать, что утверждение P всегда истинно, заключается в доказательстве того, что P является *инвариантом* согласно определению, которое приводится ниже. Запись $P(\gamma)$, где γ — конфигурация, будет обозначать булево выражение, принимающее истинное значение в том случае, если утверждение P справедливо для конфигурации γ , и ложное значение в противном случае.

Пусть задана некоторая система переходов $S = (\mathcal{C}, \rightarrow, \mathcal{I})$. В дальнейшем мы будем использовать запись $\{P\} \rightarrow \{Q\}$ для обозначения того, что для каждого перехода $\gamma \rightarrow \delta$ в системе S истинность утверждения $P(\gamma)$ влечет истинность утверждения $Q(\delta)$. Таким образом, запись $\{P\} \rightarrow \{Q\}$ означает, что если утверждение P выполнялось в начале перехода, то утверждение Q будет выполняться по завершении перехода.

Определение 2.9. Утверждение P называется *инвариантом* системы S , если

- 1) $P(\gamma)$ истинно для каждой начальной конфигурации $\gamma \in \mathcal{I}$, и
- 2) $\{P\} \rightarrow \{P\}$.

В этом определении говорится о том, что инвариант выполняется в каждой начальной конфигурации и его выполнимость сохраняется после каждого перехода. Отсюда следует, что инвариант будет выполняться в каждой достижимой конфигурации; об этом свидетельствует следующая теорема.

Теорема 2.10. Если утверждение P является инвариантом системы переходов S , то для любого выполнения системы S утверждение P будет справедливо в каждой конфигурации выполнения.

²⁾от слова «assertion» (утверждение). — Прим. перев.

Доказательство. Рассмотрим произвольное выполнение $E = (\gamma_0, \gamma_1, \gamma_2, \dots)$ системы S . Воспользовавшись индукцией, покажем, что $P(\gamma_i)$ принимает истинное значение для каждого i . Во-первых, истинность утверждения $P(\gamma_0)$ следует из первого пункта определения 2.9 и того факта, что $\gamma_0 \in \mathcal{I}$. Во-вторых, предположив, что $P(\gamma_i)$ истинно и по ходу выполнения совершен переход $\gamma_i \rightarrow \gamma_{i+1}$, мы заключаем, исходя из второго пункта определения 2.9, что истинным будет также и $P(\gamma_{i+1})$. Этим и завершается доказательство. $\square$

Однако неверно, что утверждение, которое является истинным в каждой конфигурации любого выполнения, обязательно является инвариантом (см. упражнение 2.2). Это означает, что отнюдь не всякое свойство безопасности можно доказать, применяя теорему 2.10. Тем не менее, каждое утверждение, которое всегда истинно, является следствием некоторого инварианта³⁾; поэтому для обоснования того, что некоторое утверждение всегда истинно, можно воспользоваться следующей теоремой. Нужно, однако, заметить, что часто бывает весьма непросто подобрать подходящий инвариант Q , чтобы применить эту теорему.

Теорема 2.11. Если Q является инвариантом системы переходов S и для каждой конфигурации $\gamma \in \mathcal{C}$ выполняется отношение $Q \implies P$, то для любого выполнения системы S утверждение P будет истинно в каждой конфигурации выполнения.

Доказательство. По теореме 2.10, утверждение Q выполняется в каждой конфигурации⁴⁾. Так как $Q \implies P$, утверждение P будет также выполняться в каждой конфигурации. $\square$

Иногда бывает полезно вначале установить какой-нибудь слабый инвариант, а затем, воспользовавшись им, установить и более сильный инвариант. О том, как можно усиливать инварианты, говорится в определении и теореме, которые приводятся ниже.

Определение 2.12. Пусть S — это система переходов, а P, Q — некоторые утверждения. Утверждение P назовем *Q -производным*, если

- 1) для любой начальной конфигурации $\gamma \in \mathcal{I}$ выполняется отношение $Q(\gamma) \implies P(\gamma)$ и
- 2) $\{Q \wedge P\} \rightarrow \{Q \implies P\}$.

Теорема 2.13. Если утверждение Q является инвариантом, а утверждение P является Q -производным, то утверждение $Q \wedge P$ является инвариантом.

Доказательство. Согласно определению 2.9 необходимо показать, что

- 1) для каждой начальной конфигурации $\gamma \in \mathcal{I}$ выполняется отношение $Q(\gamma) \wedge P(\gamma)$ и
- 2) $\{Q \wedge P\} \rightarrow \{Q \wedge P\}$.

³⁾Можно заметить, однако, что в теореме 2.11 доказывается другое, гораздо более слабое утверждение. — Прим. перев.

⁴⁾Достижимой конфигурации. — Прим. перев.

Так как Q является инвариантом, утверждение $Q(\gamma)$ истинно для каждой начальной конфигурации $\gamma \in \mathcal{I}$. Так же для каждой начальной конфигурации $\gamma \in \mathcal{I}$ выполняется отношение $Q(\gamma) \implies P(\gamma)$. Отсюда мы заключаем, что $P(\gamma)$ истинно для всех начальных конфигураций $\gamma \in \mathcal{I}$. Следовательно, $Q(\gamma) \wedge P(\gamma)$ истинно для всех конфигураций γ из $\mathcal{I}$.

Предположим, что $\gamma \rightarrow \delta$ и $Q(\gamma) \wedge P(\gamma)$ имеет истинное значение. Так как Q является инвариантом, $Q(\delta)$ будет также иметь истинное значение. Поскольку имеет место отношение $\{Q \wedge P\} \rightarrow \{Q \implies P\}$, выражение $Q(\delta) \implies P(\delta)$ будет иметь истинное значение. Отсюда следует, что и $P(\delta)$ принимает истинное значение. Значит, $Q(\delta) \wedge P(\delta)$ также истинно. $\square$

В §3.3.1 мы рассмотрим примеры свойств безопасности, для доказательства которых будем использовать описанные здесь методы.

2.2.2. Свойства живости

Свойством живости алгоритма является всякое свойство, которое можно сформулировать в виде следующего предложения: «Для любого выполнения алгоритма утверждение P истинно в некоторой конфигурации выполнения». Более кратко это свойство можно сформулировать так: «Утверждение P когда-то обязательно становится истинным». Основной прием, при помощи которого удастся показать, что утверждение P когда-то обязательно становится истинным, заключается в использовании *функций нормировки* и *отсутствия блокировки*, или *правильного завершения*. Более простой метод можно использовать в том случае, когда все выполнения алгоритмов имеют фиксированную конечную длину.

Рассмотрим систему переходов S и предикат P . Будем считать, что предикат **term** по определению принимает истинное значение в каждой заключительной конфигурации и ложное значение в конфигурациях, которые не являются заключительными. Вначале рассмотрим те случаи, когда выполнение достигает заключительной конфигурации. Обычно бывает нежелательно достичь такой конфигурации, прежде чем выполнится «целевой» предикат P ; в этом случае говорят, что произошла *блокировка*. С другой стороны, если цель достигнута, то выполнение разрешается завершить; в этом случае говорят, что произошло *правильное завершение*.

Определение 2.14. Система S *правильно завершает* выполнения (или, иными словами, *свободна от блокировки*), если предикат (**term** $\implies P$) всегда является истинным для системы S .

В основу функций нормировки положено математическое понятие *фундированного множества*. Это такое множество с отношением порядка $<$, в котором отсутствуют бесконечно убывающие последовательности элементов.

Определение 2.15. Частично упорядоченное множество $(W, <)$ называется *фундированным*, если из его элементов нельзя составить бесконечно убывающую последовательность

$$\omega_1 > \omega_2 > \omega_3 > \dots$$

Примерами фундированных множеств, которые будут использоваться в этой книге, могут служить множество натуральных чисел с обычным отношением порядка, а также множество всех наборов длины n , состоящих из натуральных чисел, с отношением лексикографического порядка (см. §4.6.3). Отсутствие бесконечно убывающих последовательностей, состоящих из элементов фундированного множества, можно использовать для доказательства того, что некоторое утверждение P наверняка станет истинным. Для этого нужно показать, что существует функция f , отображающая $\mathcal{C}$ в фундированное множество W так, что каждый переход приводит к тому, что либо значение f уменьшается, либо P становится истинным.

Определение 2.16. Пусть заданы система переходов S и утверждение P . Функция f , отображающая множество $\mathcal{C}$ в фундированное множество W , называется *функцией нормировки* (по отношению к P), если для каждого перехода $\gamma \rightarrow \delta$ либо выполняется отношение $f(\gamma) > f(\delta)$, либо $P(\delta)$ принимает истинное значение.

Теорема 2.17. Пусть заданы система переходов S и утверждение P . Если S правильно завершает выполнения и существует функция нормировки f (по отношению к P), то для каждого выполнения S утверждение P становится истинным в некоторой конфигурации.

Доказательство. Рассмотрим произвольное выполнение $E = (\gamma_0, \gamma_1, \gamma_2, \dots)$ системы S . Если E — конечная последовательность, то последняя конфигурация является заключительной, и, коль скоро в системе S всегда верно соотношение **term** $\implies P$, утверждение P выполняется в этой конфигурации. Если выполнение E продолжается бесконечно долго, то выделим в нем самый длинный начальный отрезок E' , не содержащий ни одной конфигурации, в которой выполнялось бы утверждение P . Рассмотрим последовательность $s = (f(\gamma_0), f(\gamma_1), \dots)$ для всех конфигураций γ_i , содержащихся в E' . Согласно выбору E' и в соответствии с определением функции нормировки f последовательность s является убывающей. Коль скоро W — фундированное множество, последовательность s является конечной. Это означает также, что E' является конечным префиксом $(\gamma_0, \gamma_1, \dots, \gamma_k)$ выполнения E . Тогда, помятуя о выборе E' , приходим к заключению о том, что $P(\gamma_{k+1})$ имеет истинное значение. $\square$

Если в расчет принять свойства справедливости, то доказать, что P наверняка станет истинным, можно исходя из более слабых предпосылок, нежели те, которые фигурируют в теореме 2.17; значения функции нормировки не обязательно должны убывать после *каждого* перехода. Допущение справедливости можно использовать для того, чтобы показать, что в бесконечных выполнениях переходы определенного вида совершаются бесконечно часто. Тогда достаточно проверить, что значения f никогда не увеличиваются, зато всегда уменьшаются при выполнении переходов определенного вида.

В некоторых случаях мы будем использовать следующий результат, который является специальным вариантом теоремы 2.17.

Теорема 2.18. Если S правильно завершает выполнения и при этом существует такое число K , что в каждом выполнении совершается не более K переходов, то в каждом выполнении существует конфигурация, в которой утверждение P истинно.

2.3. Причинно-следственный порядок событий и логические часы

Представляя выполнение в виде последовательности переходов, мы тем самым естественно привносим в модель понятие времени. Будем говорить, что переход a происходит раньше, чем переход b , если в последовательности переходов событие a предшествует событию b . Сопоставим выполнению $E = (\gamma_0, \gamma_1, \dots)$ связанную с ним последовательность событий $\bar{E} = (e_0, e_1, \dots)$, где e_i обозначает событие преобразования конфигурации γ_i в конфигурацию γ_{i+1} . Будем иметь в виду, что каждому выполнению таким образом сопоставляется единственная последовательность событий. Для визуализации выполнения можно использовать пространственно-временные диаграммы; пример одной из таких диаграмм изображен на рис. 2.1. В такой диаграмме каждому процессу соответствует горизонтальная прямая линия, и каждое событие отмечается точкой на этой прямой в том месте, где происходит это событие. Если отправление сообщения t происходит при осуществлении события s , а его прием — при осуществлении события r , то в диаграмме проводится дуга из точки s в точку r ; в этом случае будем называть события s и r взаимосвязанными.

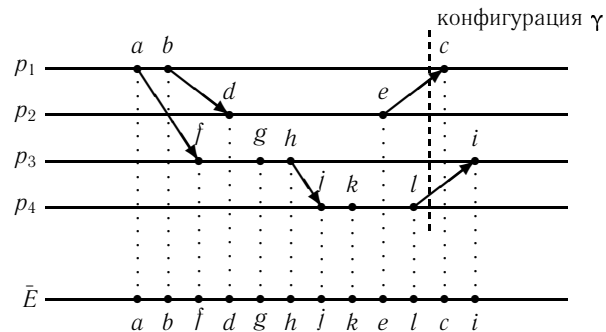


Рис. 2.1. Пример пространственно-временной диаграммы

Как будет показано в § 2.3.1, порядок следования событий в распределенном выполнении может быть изменен так, что это никак не отразится на последующих конфигурациях. Поэтому представление о времени как о линейном порядке на множестве событий выполнения не совсем подходит для распределенных вычислений; вместо этого предлагается воспользоваться отношением причинно-следственной зависимости. Эквивалентность выполнений при изменении порядка следования событий рассматривается в § 2.3.2. В § 2.3.3 мы обсудим, каким

образом можно создать такие часы, по которым можно вести отсчет причинно-следственной зависимости, а не времени, и приведем один важных примеров часов такого рода — логические часы Лампорта.

2.3.1. Зависимые и независимые события

Как уже отмечалось ранее, переходы в распределенных системах зависят и оказывают влияние только на часть конфигураций. Это наводит на мысль о том, что два последовательно происходящих события, оказывающих воздействие на два множества конфигураций, которые не пересекаются друг с другом, являются независимыми и могут осуществляться в другом порядке. Для систем с асинхронным обменом сообщениями это наблюдение приводит к следующей теореме.

Теорема 2.19. Пусть γ — конфигурация распределенной системы (с асинхронным обменом сообщениями), и пусть e_p и e_q — события, которые происходят в разных процессах p и q , и при этом оба события допустимы в конфигурации γ . Тогда событие e_p допустимо в конфигурации $e_q(\gamma)$, а событие e_q — в конфигурации $e_p(\gamma)$ и при этом $e_p(e_q(\gamma)) = e_q(e_p(\gamma))$.

Доказательство. Чтобы избежать поочередного разбора разных случаев в зависимости от того, с событиями какого типа (внутренними, отправления сообщения или приема сообщения) мы имеем дело, введем единообразное обозначение (c, x, y, d) для каждого события. Здесь символы c и d обозначают состояния процесса до осуществления события и после его осуществления, x — это совокупность сообщений, принятых по ходу осуществления события, а y — это совокупность сообщений, отправленных при осуществлении этого события. Таким образом, внутреннее событие (c, d) будет обозначаться записью $(c, \emptyset, \emptyset, d)$, событие отправления сообщения (c, m, d) — записью $(c, \emptyset, \{m\}, d)$, а событие приема сообщения (c, m, d) — записью $(c, \{m\}, \emptyset, d)$. В рамках введенной системы обозначений событие $e = (c, x, y, d)$ процесса p допустимо в конфигурации $\gamma = (c_{p_1}, \dots, c_p, \dots, c_{p_N}, M)$, если $c_p = c$ и $x \subseteq M$. В этом случае

$$e(\gamma) = (c_{p_1}, \dots, d, \dots, (M \setminus x) \cup y).$$

Теперь предположим, что события $e_p = (b_p, x_p, y_p, d_p)$ и $e_q = (b_q, x_q, y_q, d_q)$ допустимы в конфигурации

$$\gamma = (\dots, c_p, \dots, c_q, \dots, M),$$

т.е. $c_p = b_p$, $c_q = b_q$, $x_p \subseteq M$ и $x_q \subseteq M$. Важную роль здесь играет то обстоятельство, что множества x_p и x_q не пересекаются⁵⁾; адресатом сообщения x_p (если таковое имеется) является процесс p , тогда как адресатом сообщения x_q (если таковое имеется) является процесс q .

Рассмотрим конфигурацию $\gamma_p = e_p(\gamma)$ и заметим, что

$$\gamma_p = (\dots, d_p, \dots, c_q, \dots, (M \setminus x_p) \cup y_p).$$

⁵⁾ Это следует из замечания в конце § 2.1.2. — Прим. перев.

Так как $x_q \subseteq M$ и $x_q \cap x_p = \emptyset$, справедливо включение $x_q \subseteq (M \setminus x_p \cup y_p)$, и поэтому событие e_q является допустимым в конфигурации γ_p . Рассмотрим конфигурацию $\gamma_{pq} = e_q(\gamma_p)$ и заметим, что

$$\gamma_{pq} = (\dots, d_p, \dots, d_q, \dots, ((M \setminus x_p \cup y_p) \setminus x_q) \cup y_q).$$

Учитывая, что рассматриваемые процессы равноправны, мы можем, применяя те же рассуждения, показать, что событие e_p является допустимым в конфигурации $\gamma_q = e_q(\gamma_p)$. Рассмотрим конфигурацию $\gamma_{qp} = e_p(\gamma_q)$ и заметим, что

$$\gamma_{qp} = (\dots, d_p, \dots, d_q, \dots, ((M \setminus x_q \cup y_q) \setminus x_p \cup y_p)).$$

Поскольку M — это мультимножество сообщений и при этом $x_p \subseteq M$ и $x_q \subseteq M$, справедливо равенство

$$((M \setminus x_p \cup y_p) \setminus x_q \cup y_q) = ((M \setminus x_q \cup y_q) \setminus x_p \cup y_p),$$

и поэтому $\gamma_{pq} = \gamma_{qp}$. $\square$

Пусть e_p и e_q — это два события, которые происходят последовательно в некотором выполнении, т. е. для некоторой конфигурации γ выполнение содержит подпоследовательность

$$\dots, \gamma, e_p(\gamma), e_q(e_p(\gamma)), \dots$$

Условия применения теоремы 2.19 не выполняются для событий e_p и e_q в следующих двух случаях:

- 1) $p = q$ или
- 2) e_p — это событие отправления сообщения, а e_q — это взаимосвязанное событие приема сообщения.

Действительно, в этой теореме явно оговорено, что речь идет о разных процессах p и q , и, кроме того, если e_q — это событие приема того сообщения, которое было отправлено при осуществлении события e_p , то событие приема сообщения e_q не может быть допустимым в исходной конфигурации γ , как того требуют условия теоремы. Таким образом, если имеет место одно из двух указанных соотношений, то эти события не могут произойти в другом порядке; в остальных случаях порядок, в котором они совершаются, может быть изменен, но в результате будет достигнута та же самая конфигурация. Заметим, что с глобальной точки зрения *переходы* нельзя переставлять, потому что если обратиться к теореме 2.19, то переход из конфигурации γ_p в конфигурацию γ_{pq} отличается от перехода из конфигурации γ в конфигурацию γ_q . Однако с точки зрения поведения отдельных процессов эти два события неразличимы.

То, что какие-то два события нельзя поменять местами, означает, что между этими событиями есть *причинно-следственная зависимость*. Это отношение можно распространить на все множество событий выполнения, введя тем самым *причинно-следственное отношение частичного порядка*.

Определение 2.20. Пусть задано выполнение E . Отношением *причинно-следственного порядка* $\prec$ называется наименьшее отношение, удовлетворяющее следующим условиям:

- 1) если e и f — два разных события одного и того же процесса и при этом событие e происходит прежде события f , то $e \prec f$;
- 2) если s — это событие отправления сообщения, а r — взаимосвязанное с ним событие приема сообщения, то $s \prec r$;
- 3) Отношение $\prec$ обладает свойством транзитивности.

Будем использовать запись $a \preceq b$ для обозначения отношения, которое определяется формулой $(a \prec b \vee a = b)$. Поскольку $\preceq$ является отношением *частичного* порядка, могут найтись события a и b , для которых не выполняется $a \preceq b$ и $b \preceq a$. Такие события будем называть *параллельными* и будем использовать для обозначения этого факта запись $a \parallel b$. Например, для выполнения, изображенного на рис 2.1, имеют место $b \parallel f$, $d \parallel i$, и т. п.

Причинно-следственный порядок был впервые введен в работе Лампорта [124], и он играет значительную роль при проведении логического анализа распределенных алгоритмов. Как следует из определения отношения $\prec$, события, связанные причинно-следственной зависимостью, могут образовывать *причинно-следственные цепочки*. Это означает, что выполнимость отношения $a \prec b$ предполагает существование такой последовательности событий $a = e_0, e_1, \dots, e_k = b$, что для каждой пары соседних ее членов выполняется либо п. 1), либо п. 2) определения 2.20. Можно выбрать такую причинно-следственную цепочку, в которой любые два события, удовлетворяющие п. 1), происходят в том процессе, к которому они относятся, последовательно, т. е. между ними не происходит никаких событий. Для выполнения, изображенного на рис. 2.1, причинно-следственной цепочкой между событием a и событием l может служить последовательность a, f, g, h, j, k, l .

2.3.2. Эквивалентность выполнений: вычисления

В этом параграфе мы покажем, что любая перестановка событий в выполнении, сохраняющая причинно-следственный порядок, не оказывает влияния на результат выполнения. Такая перестановка событий приводит к другой последовательности конфигураций, но полученное при этом выполнение будет считаться эквивалентным исходному выполнению.

Пусть задана последовательность событий $f = (f_0, f_1, f_2, \dots)$. С этой последовательностью событий может быть связано выполнение $F = (\delta_0, \delta_1, \delta_2, \dots)$, если для каждого i событие f_i является допустимым в конфигурации δ_i и при этом $f_i(\delta_i) = \delta_{i+1}$. В таком случае будем говорить, что последовательность f *подразумевает* выполнение F . Мы бы предпочли, чтобы выполнение F однозначно определялось последовательностью f , но это не всегда возможно: если ни одно событие процесса p не содержится в последовательности f , то любое состояние процесса p можно взять в качестве начального. Если, однако, в f содержится хотя бы одно событие процесса p , то первое событие (c, x, y, d) , которое совершается в процессе p , указывает на то, что начальным состоянием процесса p является c . Таким образом, если в f содержится хотя бы одно событие каждого процесса, то

начальная конфигурация δ_0 определяется однозначно, и за счет этого однозначно определяется и все выполнение.

Рассмотрим некоторое выполнение $E = (\gamma_0, \gamma_1, \gamma_2, \dots)$ и связанную с ним последовательность событий $\bar{E} = (e_0, e_1, e_2, \dots)$. Предположим, что задана перестановка f элементов последовательности $\bar{E}$. Это означает, что существует такая перестановка σ на множестве натуральных чисел (или на конечном множестве $\{0, \dots, k-1\}$, если E — конечное выполнение, в котором совершается k событий), что $f_i = e_{\sigma(i)}$. Перестановка $(f_0, f_1, f_2, \dots)$ событий выполнения E *сохраняет* причинно-следственный порядок, если отношение $f_i \preceq f_j$ влечет неравенство $i \leq j$, т. е. ни одно событие не появляется в этой последовательности прежде того события, от которого оно зависит.

Теорема 2.21. Пусть $f = (f_0, f_1, f_2, \dots)$ — перестановка событий выполнения E , сохраняющая причинно-следственный порядок событий выполнения. Тогда f определяет единственное выполнение F , которое начинается с той же конфигурации, что и выполнение E . При этом в F совершается столько же событий, сколько их совершается в E , и в том случае, если E — конечное выполнение, то последняя конфигурация выполнения F будет точно такой же, как последняя конфигурация выполнения E .

Доказательство. Будем формировать конфигурации выполнения F последовательно одну за другой; для построения δ_{i+1} достаточно показать, что событие f_i является допустимым в конфигурации δ_i . Будем полагать, что $\delta_0 = \gamma_0$.

Предположим, что для всех $j < i$ событие f_j является допустимым в конфигурации δ_j и при этом $\delta_{j+1} = f_j(\delta_j)$. Пусть $\delta_i = (c_{p1}, \dots, c_{pN}, M)$, и пусть $f_i = (c, x, y, d)$ — это событие процесса p . Тогда событие f_i будет допустимым в конфигурации δ_i , если $c_p = c$ и $x \subseteq M$.

Чтобы показать, что имеет место равенство $c_p = c$, нужно рассмотреть два случая. В обоих случаях мы принимаем во внимание то обстоятельство, что причинно-следственный порядок в исполнении E линейно упорядочивает события процесса p ; это означает, что порядок расположения событий процесса p в последовательности f будет тем же самым, что и в последовательности $\bar{E}$.

Случай 1: f_i — это первое событие процесса p в последовательности f . Тогда c_p является начальным состоянием процесса p . Значит, f_i также является и первым событием процесса p в последовательности $\bar{E}$, и поэтому c — начальное состояние процесса p . Следовательно, $c = c_p$.

Случай 2: f_i не является первым событием процесса p в последовательности f . Пусть последним событием процесса p в последовательности f , предшествующим f_i , является событие $f_{i'} = (c', x', y', d')$. Тогда $c_p = d'$. Но тогда $f_{i'}$ — это также и последнее событие процесса p , предшествующее f_i в последовательности $\bar{E}$. Поэтому $c = d'$, и отсюда следует равенство $c = c_p$.

Чтобы показать, что имеет место включение $x \subseteq M$, мы заметим, что соответствующие друг другу события отправления и приема сообщений расположены в последовательностях f и $\bar{E}$ в одном и том же порядке. Если f_i не является событием приема сообщения, то $x = \emptyset$, и включение $x \subseteq M$, очевидно, выполняется. Если же f_i — это событие приема сообщения, то рассмотрим соответствующее

ему событие отправления сообщения f_j . Так как $f_j \prec f_i$, справедливо неравенство $j < i$, т. е. событие отправления сообщения предшествует в последовательности f событию f_i . Следовательно, $x \subseteq M$.

Покажем теперь, что для любого i событие f_i является допустимым в конфигурации δ_i и при этом в качестве δ_{i+1} можно взять $f_i(\delta_i)$. В конечном итоге мы должны показать, что последние конфигурации выполнений F и E совпадают, если E — конечная последовательность. Пусть γ_k — это последняя конфигурация выполнения E . Если в $\bar{E}$ не содержится ни одного события процесса p , то состоянием процесса p в конфигурации γ_k является начальное состояние. Коль скоро в f также не содержится ни одного события процесса p , состоянием процесса p в конфигурации δ_k также будет начальное состояние. Следовательно, состояние процесса p в конфигурации δ_k будет тем же самым, что и в конфигурации γ_k . В противном случае состояние процесса p в конфигурации γ_k — это то состояние, в которое переходит процесс p , после того как в этом процессе совершится последнее событие из числа тех, которые содержатся в последовательности $\bar{E}$. Это событие также является последним событием, которое совершается в процессе p по ходу осуществления последовательности событий f . Значит, точно таким же будет и состояние процесса p в конфигурации δ_k .

Сообщения, пребывающие на этапе пересылки в конфигурации γ_k , — это в точности те самые сообщения, для которых события отправления сообщений не имеют парных им соответствующих событий приема сообщений в последовательности $\bar{E}$. Но коль скоро в последовательностях $\bar{E}$ и f содержится одна и та же совокупность сообщений, те же самые сообщения будут пребывать на этапе пересылки и в последней конфигурации последовательности F . $\square$

В обоих выполнениях F и E происходит одна и та же совокупность событий, и причинно-следственный порядок на множестве событий выполнения E тот же самый, что и на множестве событий выполнения F . Поэтому последовательность событий $\bar{E}$ может быть образована в результате перестановки событий выполнения F , сохраняющей частичный порядок на множестве событий этого выполнения. В том случае, если выполняются условия теоремы 2.21, мы будем говорить, что выполнения E и F *эквивалентны* и обозначать это отношение $E \sim F$.

На рис. 2.2 изображена временная диаграмма выполнения, эквивалентного тому выполнению, которое представлено на рис. 2.1. Эквивалентные временные диаграммы можно получить из заданной диаграммы путем ее растяжения и сжатия, как резиновой ленты; см. [142]. Представьте себе, что оси времени отдельных процессов можно сжимать и растягивать как угодно, лишь бы только стрелки на них оставались ориентированными слева направо, и диаграмма на рис. 2.1 превратилась в диаграмму на рис. 2.2.

Хотя изображенные на этих рисунках выполнения эквивалентны и в них происходит одно и то же множество событий, эти выполнения содержат разные совокупности конфигураций. Выполнение, диаграмма которого изображена на рис. 2.1, содержит конфигурацию γ , в которой оба сообщения, отправленные при осуществлении событий e и l , пребывают на этапе пересылки одновременно. А выполнение, диаграмма которого изображена на рис. 2.2, не содержит ни од-

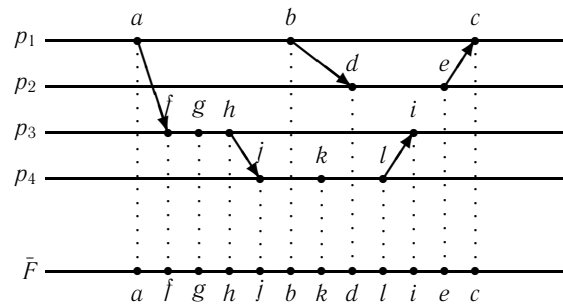


Рис. 2.2. Пространственно-временная диаграмма, эквивалентная диаграмме, изображенной на рис. 2.1.

ной такой конфигурации, поскольку сообщение, отправленное при осуществлении события l , достигает адресата *раньше*, чем происходит событие e .

Сторонний наблюдатель, который имеет возможность видеть подлинную последовательность событий, способен отличить одно эквивалентное выполнение от другого, т. е. он всякий раз видит только одно из возможных выполнений. Однако процессы не могут отличить два эквивалентных выполнения, так как с точки зрения процессов невозможно понять, какое из двух эквивалентных выполнений происходит на самом деле. Это можно объяснить на следующем примере. Предположим, что нужно выяснить, действительно ли сообщения, отправленные при осуществлении событий e и l , пребывают на этапе пересылки одновременно. Заведем в одном из процессов булеву переменную sim , значение которой положим равным $true$, если сообщения пребывают на этапе пересылки одновременно, и равным $false$ в противном случае. Тогда в последней конфигурации выполнения, изображенного на рис. 2.1, значение переменной sim должно быть равно $true$, а в последней конфигурации выполнения, изображенного на рис. 2.2, значение переменной sim должно быть равно $false$. Но по теореме 2.21 эти конфигурации должны быть эквивалентны, и поэтому ожидаемого присваивания значений переменной sim достичь невозможно.

Класс эквивалентности выполнений по отношению эквивалентности $\sim$ однозначно определяется множеством событий и действующим на нем причинно-следственным порядком; эти классы эквивалентности называются вычислениями алгоритма.

Определение 2.22. *Вычислением* распределенного алгоритма называется класс эквивалентности выполнений алгоритма по отношению эквивалентности $\sim$.

Нет смысла говорить о конфигурациях вычисления, потому что различные выполнения одного и того же вычисления могут иметь разные конфигурации. Но при этом имеет смысл говорить о совокупности событий вычисления, так как во всех выполнениях одного и того же вычисления происходят одни и те же события. Причинно-следственный порядок на множестве событий также однозначно опре-

деляется вычислением. Мы будем называть вычисление конечным, если конечны его выполнения. Все выполнения одного и того же вычисления начинаются из одной и той же конфигурации и завершаются, если вычисление конечно, в одной и той же конфигурации (см. теорему 2.21); эти конфигурации называются начальной и заключительной конфигурациями вычисления. Мы будем отождествлять вычисление с частично упорядоченным множеством событий, которые происходят в нем.

Как следует из теории частично упорядоченных множеств, два параллельных события одного и того же вычисления могут происходить в произвольном порядке.

Утверждение 2.23. *Пусть $(X, <)$ — частично упорядоченное множество и для двух элементов $a, b \in X$ выполняется соотношение $b \not< a$. Тогда существует такая линейзация $<_1$ частичного порядка $<$, для которой выполняется соотношение $a <_1 b$.*

Это означает, что если a и b — параллельные события вычисления S , то существуют два таких выполнения E_a и E_b этого вычисления, в одном из которых событие a происходит раньше события b , а в другом, наоборот, событие b происходит раньше события a . Процессам же по ходу выполнения неизвестно, какое из этих двух событий происходит первым.

Синхронный обмен сообщениями. Модифицированный вариант теоремы 2.19 можно сформулировать и для систем с синхронным обменом сообщениями. Как утверждается в этой теореме, в таких системах два последовательно произошедших события будут независимыми, если они затрагивают разные процессы.

Теорема 2.24. *Предположим, что γ — конфигурация распределенной системы с синхронным обменом сообщениями, и пусть e_1 — синхронный переход в процессах p и q , а e_2 — синхронный переход в процессах r и s , отличных от p и q , и при этом оба события e_1 и e_2 допустимы в конфигурации γ . Тогда переход e_1 допустим в конфигурации $e_2(\gamma)$, переход e_2 допустим в конфигурации $e_1(\gamma)$ и при этом $e_1(e_2(\gamma)) = e_2(e_1(\gamma))$.*

Доказательство этой теоремы, в котором можно использовать те же рассуждения, что в доказательстве теоремы 2.19, оставлено читателям в качестве упражнения (см. упражнение 2.9). Понятие причинно-следственной зависимости в синхронных системах можно определить подобно тому, как это было сделано в определении 2.20; заинтересованные читатели могут обратиться к работе [48]. Теорема, аналогичная теореме 2.21, будет также справедлива для синхронных систем.

Выполнения и вычисления. По ходу изложения мы вначале ввели понятие выполнения как пути в системе переходов и лишь затем ввели понятие вычисления как класса эквивалентности на множестве выполнений. Такой порядок расположения материала соответствует операционной точке зрения на функционирование распределенных систем. Во многих других работах предпочтение отдается более формальному подходу, при котором вначале определяется поня-

тие вычисления как частично упорядоченного множества событий и лишь затем вводится понятие выполнения как линеаризации этого частично упорядоченного множества. Такому подходу следуют, например, Чаррон-Бост и др. в работе [48], а также разработчики стандартного языка спецификаций Message Sequence Charts в работе [164]. Эти два подхода совершенно равноправны.

2.3.3. Логические часы

Мы можем снабдить распределенные системы часами, при помощи которых можно «отсчитывать» причинно-следственную зависимость, подобно тому как физические часы отсчитывают реальное время. В этом параграфе букву Θ будем использовать для обозначения функции, отображающей множество событий в некоторое упорядоченное множество $(X, <)$.

Определение 2.25. Часами назовем всякую функцию Θ , отображающую множество событий в некоторое упорядоченное множество так, что при этом выполняется соотношение

$$a \prec b \implies \Theta(a) < \Theta(b).$$

Далее мы рассмотрим и обсудим некоторые примеры таких часов.

1. *Последовательное упорядочение.* Для выполнения E , представленного в виде последовательности событий $(e_0, e_1, e_2, \dots)$, положим $\Theta_g(e_i) = i$. Таким образом, каждое событие помечается порядковым номером, которое оно имеет в этой последовательности.

Такой функцией может воспользоваться сторонний наблюдатель системы, который может наблюдать последовательность происходящих событий. Однако эту последовательность нельзя наблюдать, пребывая *внутри* системы; иначе говоря, распределенный алгоритм не способен вычислять функцию Θ_g . Это следует из теоремы 2.19. Предположим, что какой-то распределенный алгоритм записывает значение $\Theta_g(e_i) = i$ для одного из тех событий e_i , которые фигурируют в предпосылке теоремы. Рассмотрим теперь эквивалентное выполнение, в котором изменен порядок осуществления двух событий; хотя значение функции Θ_g изменилось, процесс будет записывать то же самое значение i . Иными словами, функция Θ_g определена на множестве выполнений, а не вычислений.

2. *Часы реального времени.* Можно *расширить* модель, рассматриваемую нами в этой главе, снабдив каждый ее процесс встроенным часовым механизмом. Итак, для каждого события можно зафиксировать время его осуществления, и вычисляемые таким образом значения удовлетворяют определению часов.

Распределенные системы с часами реального времени не подпадают под определение 2.6, потому что физические свойства часов позволяют синхронизировать изменения состояний в разных процессах. Течение времени происходит во всех процессах, и это приводит к тому, что возникают переходы, которые изменяют состояния (а именно показания часов) во всех процессах системы. Оказывается, эти «глобальные переходы» решительным образом изменяют всю модель: если предполагать наличие часов реального времени, то теорема 2.19 перестает быть

справедливой. Однако распределенные системы с часами реального времени имеют практическое применение, и мы исследуем их в нашей книге в § 3.3.2 и в гл. 12 и 15.

3. *Логические часы Лампорта.* Лампорт в работе [124] предложил использовать часовую функцию, которая приписывает событию a число k , равное длине самой протяженной последовательности событий $(e_1, \dots, e_k)$, удовлетворяющей условию

$$e_1 \prec e_2 \prec \dots \prec e_k = a.$$

Действительно, если $a \prec b$, то эту последовательность можно продолжить так, чтобы выполнялось соотношение $\Theta_L(a) < \Theta_L(b)$. Значение Θ_L для каждого события может быть вычислено распределенным алгоритмом по следующим правилам:

- а) если a — это внутреннее событие процесса или событие отправления сообщения, и a' — это предшествующее событие в том же самом процессе, то $\Theta_L(a) = \Theta_L(a') + 1$;
- б) если a — событие приема сообщения, a' — предшествующее событие в том же самом процессе, и b — событие отправления сообщения, соответствующее a , то $\Theta_L(a) = \max(\Theta_L(a'), \Theta_L(b)) + 1$.

Значение $\Theta_L(a')$ полагается равным 0, если a — первое событие в процессе.

Чтобы распределенный алгоритм мог вычислять показания часов, часовая отметка последнего события, произошедшего в процессе p , записывается в переменную θ_p (ее начальное значение равно 0). Чтобы вычислить часовую отметку события приема, каждое сообщение m снабжается часовой отметкой θ_m события отправления этого сообщения e . Показания логических часов Лампорта вычисляются алгоритмом 2.3. Для события e в процессе p значение $\Theta_L(e)$ равно величине θ_p сразу после осуществления события e , т. е. в тот момент, когда происходит изменение состояния в процессе. В качестве упражнения читателям предлагается доказать, что введенная таким образом функция Θ_L удовлетворяет определению часов.

```
var  $\theta_p$  : integer    init 0 ;
```

(* Внутреннее событие *)

```
 $\theta_p := \theta_p + 1$  ;  
Изменение состояния
```

(* Событие отправления сообщения *)

```
 $\theta_p := \theta_p + 1$  ;  
send (messg,  $\theta_p$ ) ; Изменение состояния
```

(* Событие приема сообщения *)

```
receive (messg,  $\theta$ ) ;  $\theta_p := \max(\theta_p, \theta) + 1$  ;  
Изменение состояния
```

Алгоритм 2.3. Логические часы Лампорта

В алгоритме не оговорены условия отправления сообщения или изменения состояния процесса. Эти часы играют роль вспомогательного механизма, которым может быть снабжен любой распределенный алгоритм, для того чтобы оценивать порядок осуществления событий.

4. *Векторные часы.* Иногда бывает удобно иметь часы, при помощи которых можно отмечать не только причинно-следственный порядок (как того требует определение 2.25), но также и параллелизм. Параллелизм можно зафиксировать, если часовые пометки параллельных событий оказываются несравнимыми; это происходит, если в определении 2.25 место импликации занимает эквивалентность и возникает соотношение

$$a \prec b \iff \Theta(a) < \Theta(b). \quad (2.1)$$

Существование параллельных событий возможно только тогда, когда областью значений X функции часов является множество, которое упорядоченно частично, но не линейно.

В *векторных часах* Маттерна [144] $X = \mathbb{N}^N$, т. е. $\Theta_v(a)$ — это *вектор* длины⁶⁾ N . Векторы длины n можно естественным образом упорядочить, воспользовавшись следующим отношением покомпонентного сравнения:

$$(a_1, \dots, a_n) \leq_v (b_1, \dots, b_n) \iff \forall i (1 \leq i \leq n) : a_i \leq b_i. \quad (2.2)$$

Введенный таким образом векторный порядок отличен от лексикографического порядка, который определен в упражнении 2.5 и является линейным порядком. Значением функции часов является вектор $\Theta_v(a) = (a_1, \dots, a_N)$, в котором a_i — это число событий e , произошедших в процессе p_i и удовлетворяющих соотношению $e \preceq a$. Подобно часам Лампорта, эта функция может быть вычислена распределенным алгоритмом.

Чаррон-Бост в своей работе [47] показал, что векторами меньшей длины для этой цели обойтись невозможно (если порядок на векторах определен соотношением (2.2)). Если события произвольного выполнения системы, состоящей из N процессов, отобразить в пространство векторов длины n так, чтобы выполнялось соотношение (2.1), то мы неизбежно будем иметь $n \geq N$.

2.4. Дополнительные допущения. Сложность

Для изучения последующих глав книги вполне достаточно тех понятий, определения которых были даны в этой главе; на основе разработанной здесь модели будет проводиться описание алгоритмов, их верификация, а также доказательства невозможности решения распределенных задач. В последующих главах по мере необходимости будут использоваться дополнительные обозначения и допущения. В этом параграфе мы обсудим ряд понятий, наиболее широко используемых в литературе, посвященной распределенным алгоритмам.

⁶⁾Здесь N обозначает количество процессов в рассматриваемой распределенной системе $\mathbb{P}$ (см. определение 2.5) — *Прим. перев.*

2.4.1. Топология сети

До сих пор модель коммуникационной подсистемы была представлена у нас совокупностью сообщений, пребывающих на этапе пересылки. Кроме того, было отмечено, что каждое сообщение имеет в качестве адресата в точности один из процессов системы. Однако в общем случае совсем не обязательно, чтобы каждый процесс мог отправлять сообщения любому другому процессу. Напротив, для каждого процесса определяется подмножество других процессов (они называются *соседями* этого процесса), которым он может отправлять сообщения. Если процесс p может отправлять сообщения процессу q , то мы будем говорить, что между процессом p и процессом q имеется *канал*. Если не оговорено противное, мы будем подразумевать, что каналы являются *двунаправленными*, т. е. тот же самый канал позволяет процессу q отправлять сообщения в адрес процесса p . Канал, который способен поддерживать лишь одностороннюю передачу сообщений от процесса p в адрес процесса q , называется *однонаправленным* (или *ориентированным*) каналом от p к q .

Совокупность, которую образуют процессы и подсистема коммуникаций, называют также *сетью*. Структуру коммуникационной подсистемы часто представляют в виде *графа* $G = (V, E)$, вершины которого соответствуют процессам; при этом ребро между двумя процессами существует в том и только том случае, когда имеется канал связи между этими двумя процессами. Подобным же образом система с однонаправленными каналами представляется в виде ориентированного графа. Указанный граф распределенной системы также называется *топологией сети*.

Использование графов позволяет нам обсуждать устройство коммуникационных систем в терминах теории графов (для ознакомления с соответствующей терминологией см. гл. Б). Поскольку от топологии сети существенным образом зависят вопросы, связанные с существованием, обликом и сложностью распределенных алгоритмов, предназначенных для решения многих задач, мы бегло ознакомимся с некоторыми наиболее распространенными топологиями (их более подробное описание приведено в гл. Б). Далее, если не оговорено противное, мы будем подразумевать, что топология является *связной*, т. е. между любыми двумя вершинами существует путь.

1. *Кольца.* Кольцом размера N называется такой граф, у которого имеется N вершин $v_0, v_1, \dots, v_{N-1}$ и при этом каждая пара вершин v_i, v_{i+1} (индексация проводится по модулю N) соединена ребром. Ввиду простоты топологии кольца часто используются в распределенных системах управления. Во многих информационных сетях, наподобие сетей с маркером (Token Ring) [182, § 4.3.3], узлы сети также образуют кольцо.

2. *Деревья.* Деревом с N вершинами называется связный граф, имеющий $N - 1$ ребро; в этом графе отсутствуют циклы. Применение деревьев в распределенных вычислениях обусловлено тем, что они позволяют проводить вычисления с небольшими затратами на коммуникацию, и, кроме того, в каждом связном графе можно выделить подсеть, представляющую собой остоное дерево.

3. *Звезды*. В *звезде* размера N имеется одна выделенная вершина (она называется *центром звезды*) и $N-1$ ребер, соединяющих центр звезды с каждой из $N-1$ оставшихся вершин. Звезды используются при проведении централизованных вычислений, в которых один из процессоров выступает в роли контроллера и все остальные процессы могут взаимодействовать только с этим выделенным процессом. К недостаткам звездной топологии можно отнести эффект «бутылочного горлышка», возникающий в центре, а также уязвимость таких систем по отношению к неисправностям в центре.

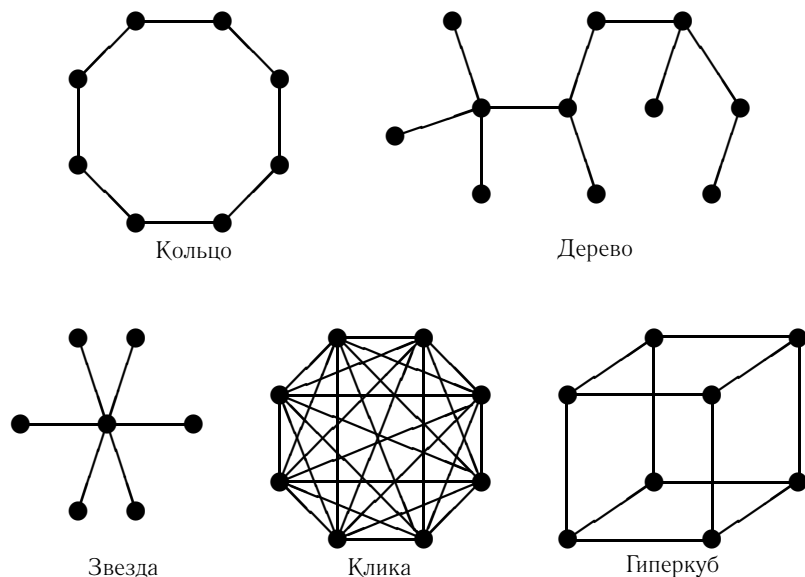


Рис. 2.4. Примеры наиболее употребительных топологий

4. *Клики*. *Кликой* называется сеть, в которой любые две вершины соединены ребром. Это тот самый граф, который неявно подразумевается в случае, когда каждый процесс может непосредственно взаимодействовать со всеми другими процессами; об этом будет говориться в гл. 13—16.

5. *Гиперкубы*. *Гиперкубом* называется граф $HC_N = (V, E)$, имеющий $N = 2^n$ вершин. Множество вершин V состоит из всех двоичных строк длины n :

$$V = \{(b_0, \dots, b_{n-1}) : b_i \in \{0, 1\}\},$$

и две вершины b и c соединены ребром в том и только том случае, когда двоичные строки b и c отличаются всего лишь в одном разряде. Своим названием (гиперкуб) эта разновидность графов обязана графическому представлению сети в виде n -мерного единичного куба, в вершинах которого размещаются узлы сети. Примеры сети каждого из перечисленных типов изображены на рис. 2.4. Топология может быть *статичной* или *динамичной*. Статичная топология остается

неизменной на протяжении всего распределенного вычисления. Динамическая топология допускает возможность добавления или изъятия каналов связи между процессами (а иногда и самих процессов) по ходу вычисления. Эти изменения топологии можно промоделировать посредством переходов между конфигурациями; для этого достаточно, чтобы в процессах хранилась информация об их соседях (см. гл. 4).

2.4.2. Свойства каналов

Модель, описанную в § 2.1.2, можно уточнить, записывая в конфигурации содержимое каждого канала по отдельности, т. е. заменяя множество M совокупностью множеств M_{pq} для каждого (однонаправленного) канала pq . Коль скоро мы постулировали, что адресат каждого сообщения определен заранее явным образом, такая модификация не влечет никаких существенных изменений модели. Зато теперь мы можем обсудить некоторые общепринятые допущения относительно взаимосвязи событий отправления и приема сообщений.

1. *Надежность*. Канал считается *надежным*, если каждое сообщение отправленное по этому каналу, непременно доставляется адресату и при этом в единственном экземпляре (при условии, что адресат способен осуществить прием этого сообщения). Если не оговорено противное, в этой книге все каналы будут считаться надежными. На самом деле это допущение приводит к появлению дополнительного условия (слабой) справедливости: начиная с того момента, когда сообщение было отправлено, прием этого сообщения (адресатом, который пребывает в подходящем для приема этого сообщения состоянии) становится допустимым событием.

Канал, не являющийся надежным, может быть подвержен *коммуникационным неисправностям*; существует несколько типов таких неисправностей: потеря, искажение, дублирование, спонтанное порождение. Эти неисправности могут быть представлены переходами в модели согласно определению 2.6, но переходы такого рода не приводят к изменению состояний процессов.

Потеря сообщения происходит в том случае, когда сообщение отправлено, но никогда не может быть получено; этот эффект можно промоделировать переходом, при котором сообщение изымается из мультимножества M . *Искажение* сообщения возникает, когда полученное сообщение оказывается отличным от того сообщения, которое было отправлено; это можно промоделировать при помощи перехода, изменяющего одно сообщение из мультимножества M . *Дублирование* сообщения приводит к тому, что сообщения принимаются чаще, чем отправляются; это можно промоделировать посредством перехода, добавляющего в M одну или несколько копий одного и того же сообщения. *Спонтанное порождение* сообщения происходит в том случае, когда адресат получает сообщение, которое вообще никогда не было отправлено; это моделируется переходом, который вставляет некоторое сообщение в мультимножество M .

2. *Свойство очередности*. Канал, сохраняющий порядок передаваемых по нему сообщений, называется *очередью*. Это означает, что если процесс p отправляет два сообщения m_1 и m_2 в адрес процесса q и при этом отправление

сообщения m_1 происходит раньше чем отправление сообщения m_2 , то получение сообщения m_1 также происходит раньше, чем получение сообщения m_2 . Если не оговорено противное, мы будем полагать, что рассматриваемые в этой книге каналы *не являются* очередями.

Очереди хорошо согласуются с определением 2.6; их можно легко представить, заменив совокупность сообщений M множеством *очереди*, по одной для каждого канала связи. Отправление сообщения приводит к тому, что это сообщение помещается в конец очереди, а в результате события приема сообщение извлекается из начала очереди. Когда возникают очереди, появляется и новый тип коммуникационных неисправностей, а именно нарушение очередности сообщений в канале; его можно промоделировать с помощью перехода, изменяющего порядок расположения сообщений в очереди.

Иногда случается так, что распределенный алгоритм извлекает определенную выгоду из свойства очередности событий в канале; примером тому может служить коммуникационный протокол, приведенный в § 3.3.1. Воспользовавшись тем, что при приеме сообщений соблюдается очередность, можно уменьшить объем информации, который должен передаваться в каждом сообщении. Однако во многих случаях алгоритм должен быть устроен так, чтобы он функционировал правильно (и эффективно) даже тогда, когда изменяется порядок расположения событий в канале. В общем случае реализация свойства очередности может уменьшить присущий вычислению параллелизм, поскольку принимающий процесс должен обеспечить буферизацию сообщений перед их обработкой. По этой причине мы предпочли не использовать в этой книге неявного предположения об очередности сообщений.

Более слабое предположение было предложено в работе Ахуджи [5]; *выровненным каналом* называется канал, в котором сохраняется очередность только тех сообщений, которые были особо отмечены при их отправлении. Можно рассмотреть и более сильные допущения. Щипер и др. в работе [172] дали следующее определение *передачи сообщений, сохраняющей причинно-следственный порядок*. Если в процессах p_1 и p_2 происходят события e_1 и e_2 отправления сообщений m_1 и m_2 процессу q и при этом справедливо соотношение $e_1 \prec e_2$, то q получает событие m_1 раньше, чем событие m_2 . Иерархия различных предположений о порядке доставки сообщений, включающая полную асинхронность, передачу с сохранением причинно-следственного порядка, доставку по очереди и синхронную коммуникацию, была исследована в работе Чаррон-Боста и др. [48].

3. Пропускная способность каналов. *Пропускной способностью канала* называется количество сообщений, которые могут одновременно пребывать в канале на этапе пересылки. Канал считается *переполненным* в тех конфигурациях, в которых число содержащихся в нем сообщений в точности равно его пропускной способности. Событие отправления допустимо только в тех случаях, когда канал не является переполненным.

В определении 2.6 рассматривается модель, каналы которой имеют неограниченную пропускную способность и никогда не переполняются. В нашей книге

мы всегда будем считать, что пропускная способность каналов является неограниченной.

2.4.3. Допущения реального времени

Важнейшее свойство рассматриваемых моделей — это, безусловно, их распределенность, т. е. абсолютная независимость событий, происходящих в разных процессах, о чем свидетельствует теорема 2.19. Это свойство утрачивается, как только мы вводим модель в рамки реального времени и придаем процессам способность отсчитывать физическое время (встраиваем в процессы таймер). В самом деле, если истекает реальное время, то оно истекает во всех процессах, и течение времени отражается на часах каждого процесса.

В нашу модель можно ввести часы реального времени, снабдив каждый процесс вещественной переменной (часами); течение реального времени моделируется посредством перехода, который продвигает вперед показания часов каждого процесса (см. § 3.3.2). Наряду с наличием часов реального времени обычно предполагается, что время доставки сообщения (отрезок времени между осуществлением события отправления и приема сообщения) ограничено. Этот срок доставки сообщения также может быть включен в общую модель системы переходов.

В этой книге, за исключением особо оговоренных случаев, мы будем иметь дело с допущениями реального времени, т. е. мы рассматриваем *вполне асинхронные* системы и алгоритмы. Допущения о времени будут использоваться в § 3.3.2, в гл. 12, и в гл. 15.

2.4.4. Осведомленность процессов

Мы будем использовать термин «осведомленность процессов» для обозначения той информации о распределенной системе, которая представлена в начальном состоянии процессов. В тех случаях, когда мы говорим, что алгоритм опирается на такую информацию, это означает, что соответствующие данные заранее заложены в память процесса еще до того, как система начинает выполнение. Примером сведений такого рода может быть следующая информация.

1. Топологическая информация. Информация о топологии включает сведения о числе процессов, о диаметре графа сети, о топологии сети. Говорят, что сеть наделена *представлением об ориентации*, если процессам известна согласованная ориентированная разметка ребер сети (см. гл. Б).

2. Отличительные признаки процессов. Во многих алгоритмах требуется, чтобы процессы имели уникальные имена (идентификаторы), и чтобы каждому процессу было заранее известно его собственное имя. Предполагается, что в процессах имеется переменная, начальным значением которой является это имя (разные процессы имеют разные имена). Далее, можно уточнить, из какого множества выбираются имена процессов, упорядочены ли эти имена, являются ли эти имена натуральными числами.

Если не оговорено противное, при изложении материала мы будем исходить из предположения о том, что процессам известны их собственные имена; в та-

ком случае говорят, что сеть является *именованной*. Случаи *анонимных сетей* будут исследованы в гл. 9.

3. *Отличительные признаки соседей*. Если процессы различаются по их уникальным именам, то можно предполагать, что каждому процессу заранее известны имена его соседей. Если не оговорено противное, то допущение о сведениях такого рода, которые называются *сведениями о соседях*, нами использоваться не будут. Имена процессов могут быть использованы в качестве адресов сообщений при отправлении сообщений по *прямому адресу*. Более сильное допущение предполагает, что каждому процессу известны имена всех процессов системы.

При более слабом допущении предполагается, что процессу известно о существовании соседей, но их имена ему неизвестны. Прямой адресацией в этом случае воспользоваться невозможно, и поэтому процессы вынуждены использовать для адресации локальные имена каналов; этот способ адресации называется *косвенной адресацией*; см. [176, с. 54]. Прямая и косвенная адресация проиллюстрированы на рис. 2.5. В случае прямой адресации имена процессов играют роль адресов, а при косвенной адресации процессы p , r и s используют различные имена (a , b и c соответственно), для того чтобы доставить сообщение одному и тому же процессу q .

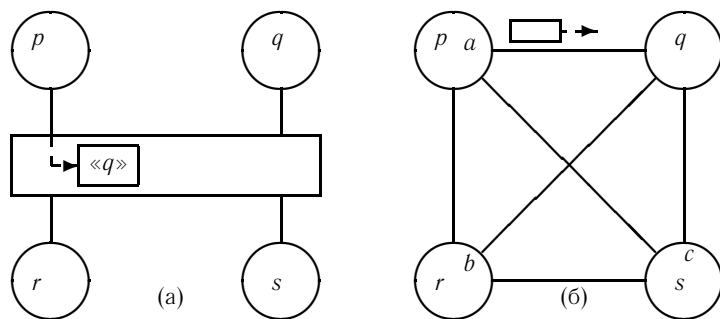


Рис. 2.5. Прямая (а) и косвенная (б) адресация

2.4.5. Сложность распределенных алгоритмов

Наиболее важным качеством распределенных алгоритмов является их корректность: алгоритм должен удовлетворять всем требованиям, которые налагаются на него той задачей, для решения которой предназначен этот алгоритм. Чтобы сравнивать друг с другом различные алгоритмы решения одной и той же задачи, можно оценить объем ресурсов, которые потребляются этими алгоритмами. Чем ниже уровень потребления, тем «лучше» алгоритм. Измерять потребление ресурсов можно по-разному.

1. *Сложность по числу обменов сообщениями*. Это общее число сообщений, которые были отправлены по ходу выполнения алгоритма.

2. *Битовая сложность*. Обычно множество M содержит несколько разных типов сообщений, и поэтому, чтобы различать эти сообщения, каждое из них должно нести определенное количество информации. Если множество типов сообщений M , используемых алгоритмом, невелико, то для идентификации этих сообщений достаточно небольшого числа битов, а вот алгоритмам, в которых задействовано много разных типов сообщений, приходится использовать большее число битов для обозначения каждого сообщения. Так как передача «длинных» сообщений требует больших затрат, нежели передача «коротких» сообщений, нужно оценивать также и суммарное количество битов, содержащихся в сообщениях.

В большинстве алгоритмов, которые описаны в этой книге, сообщения состоят из $O(\log N)$ битов, где N — общее число процессов системы, поэтому их битовая сложность отличается от сложности по числу сообщений на логарифмический множитель. Чаще всего мы будем ограничиваться анализом сложности по числу сообщений, а битовая сложность будет вычисляться только для таких алгоритмов, в которых используются очень длинные или очень короткие сообщения.

3. *Сложность по времени*. Так как в нашей модели распределенных алгоритмов не задействовано понятие времени, неясно, каким образом нам определить сложность распределенных алгоритмов по времени. В математической литературе имеется немало различных определений этого понятия (их сравнительный анализ можно найти в §6.6.4). В основу того определения, которым мы будем пользоваться в этой книге, положено идеализированное хронометрирование событий вычисления согласно следующим допущениям:

- обработка событий не занимает ни одной единицы времени;
- для передачи сообщения требуется не более одной единицы времени, т. е. протяженность отрезка времени между событием отправления и событием приема сообщения не превосходит одной единицы;

Сложностью алгоритма по времени будет считаться время, которое требуется для проведения вычисления в рамках указанных выше допущений. Следует обратить внимание, что эти допущения были выдвинуты исключительно для того, чтобы ввести определение сложности по времени для распределенных алгоритмов. Корректность асинхронных алгоритмов должна быть доказана независимо от этих допущений

4. *Сложность по объему памяти*. Сложность алгоритма по объему памяти равна количеству единиц памяти, необходимой отдельному процессу для выполнения этого алгоритма. В нашем случае этот объем равен логарифму от числа состояний процесса.

Так как функционирование распределенных алгоритмов является недетерминированным, не исключены случаи, когда допустимыми оказываются сразу несколько вычислений, имеющих различные меры сложности. Поэтому мы будем проводить различие между сложностью в среднем и сложностью в наихудшем случае. Сложность алгоритма в наихудшем случае определяется наибольшей сложностью среди всех возможных вычислений алгоритма. Сложность алгоритма в среднем, как следует из названия, представляет собой среднюю сложность по

всем возможным вычислениям, но, чтобы провести ее оценку, нужно определить распределение вероятностей по всем возможным вычислениям.

2.5. Упражнения к главе 2

2.5.1

Упражнение 2.1. Предложите определение модели распределенной системы, в которой возможен как синхронный, так и асинхронный обмен сообщениями.

2.5.2

Вниманию читателей предлагаются три упражнения, в которых речь идет о том, в чем состоит различие между инвариантом и всюду истинным предикатом, дизъюнктивными и конъюнктивными свойствами инвариантов и производных, а также о том, как себя ведут инварианты при параллельной композиции программ.

Упражнение 2.2. Приведите пример такой системы переходов S и такого утверждения P , что P всегда истинно в системе S , но при этом не является инвариантом этой системы.

(Подсказка: пример системы с тремя конфигурациями приведен в работе [99]. Можно ли отыскать пример системы с двумя конфигурациями?)

Упражнение 2.3. Предположим, что P_1 и P_2 — это инварианты системы S . Докажите, что $(P_1 \vee P_2)$ и $(P_1 \wedge P_2)$ также являются инвариантами.

Предположим, что P_1 и P_2 — это Q -производные системы S . Докажите, что $(P_1 \vee P_2)$ и $(P_1 \wedge P_2)$ также являются Q -производными системы S .

Для систем переходов с одним и тем же множеством конфигураций и одинаковыми начальными конфигурациями можно определить композицию их отношений переходов. Так, например, *параллельной композицией* систем $S_1 = (\mathcal{C}, \rightarrow_1, \mathcal{I})$ и $S_2 = (\mathcal{C}, \rightarrow_2, \mathcal{I})$ называется такая система $S = (\mathcal{C}, \rightarrow, \mathcal{I})$, у которой $\rightarrow = (\rightarrow_1 \cup \rightarrow_2)$.

Упражнение 2.4. Пусть S — параллельная композиция систем S_1 и S_2 .

Докажите, что всякий инвариант P систем S_1 и S_2 также является инвариантом системы S .

Докажите, что всякая Q -производная P систем S_1 и S_2 также является Q -производной системы S .

Приведите пример утверждения P , которое является всегда истинным в обеих системах S_1 и S_2 , но не является таковым в системе S .

В следующем упражнении речь идет о фундированных множествах. Множество может быть фундированным даже в том случае, если для некоторого элемента в множестве содержится бесконечно много элементов, меньших этого элемента. *Лексикографический порядок* для векторов $a = (a_1, a_2, \dots, a_n)$ и $b = (b_1, b_2, \dots, b_n)$ определяется следующим соотношением:

$$a <_l b \iff a_1 < b_1 \vee (a_1 = b_1 \wedge (a_2, \dots, a_n) <_l (b_2, \dots, b_n)).$$

Упражнение 2.5. Докажите, что в множестве $(\mathbb{N}^n, \leq_l)$, где $n \geq 2$, имеется такой элемент, который превосходит бесконечно много элементов этого множества.

Докажите, что множество $(\mathbb{N}^n, \leq_l)$ является фундированным.

2.5.3

Упражнение 2.6. Определите, каковы будут показания логических часов Лампорта для всех событий выполнения, представленного на рис. 2.1.

Определите, каковы будут показания часов Маттерна для этих событий.

Выделите некоторые пары параллельных событий и проверьте, будут ли упорядочены показания часов для этих событий.

Упражнение 2.7. Предложите распределенный алгоритм (подобный алгоритму 2.3), который вычисляет для каждого события показания векторных часов Маттерна.

Упражнение 2.8. Можно ли доказать теорему 2.21 путем последовательного применения теоремы 2.19 к событиям выполнения E ?

Упражнение 2.9. Докажите теорему 2.24.

Упражнение 2.10. Подберите подходящее определение для причинно-следственного порядка переходов системы с синхронным обменом сообщениями. Предложите определение часов для систем такого рода и постройте распределенные алгоритмы для вычисления показаний этих часов.

Г Л А В А 3

КОММУНИКАЦИОННЫЕ ПРОТОКОЛЫ

В этой главе мы рассмотрим два протокола, которые используются для обеспечения надежного обмена данными между двумя вычислительными станциями. В идеальном случае обмен данными можно было бы осуществлять просто путем отправления и приема сообщений. Но, к сожалению, мы не можем игнорировать возможность возникновения ошибок при передаче данных; передача сообщений должна проводиться в реальной физической среде, и вследствие этого может происходить потеря или дублирование передаваемых сообщений, изменение порядка их следования, появление фиктивных сообщений. Для обнаружения и устранения этих ошибок вычислительные станции используют дополнительные средства, которые принято называть протоколами.

Основным назначением этих протоколов является *передача данных*, т. е. получение информации от одной станции и доставка ее по назначению другой станции. Надежная передача данных предполагает повторную передачу тех сообщений, которые были потеряны, избавление от дублирующих сообщений, исправление поврежденных сообщений и устранение фиктивных сообщений. Для этого в протоколе ведется учет состояния информации; в нем отмечается, какие данные уже были отправлены, какие данные были зарегистрированы как полученные и пр. В связи с необходимостью использования состояния информации возникает вопрос об *управлении соединением*, т. е. инициализации и аннулировании состояния информации. Инициализация называется *установлением* соединения, а аннулирование — *завершением* соединения. Трудность управления соединением вызвана тем обстоятельством, что при завершении соединения не исключена вероятность того, что в каналах связи могут еще оставаться сообщения. Прием таких сообщений может произойти вне всякого соединения связи или в рамках последующих соединений, и получение этих сообщений не должно нарушать правильность проведения очередных соединений.

Протоколы, которые будут рассмотрены в этой главе, относятся к разным уровням иерархии протоколов согласно классификации на основе базовой модели OSI (см. § 1.2.2). Эти протоколы включены в книгу по разным причинам; первый из этих протоколов вполне асинхронный, в то время как во втором предусматривается правильное использование таймеров. Во обоих случаях при верификации этих протоколов внимание будет сосредоточено на проверке свойства *безопасности*, которое выражает требование доставки получателю всех правильных данных.

Первый протокол, описанный в § 3.3.1, предназначен для обмена данными между двумя станциями, которые имеют прямое физическое соединение (напри-

мер, по телефонной линии); этот протокол относится к уровню управления передачей данных — второму уровню модели OSI. Второй протокол, описанный в § 3.3.2, предназначен для организации связи между двумя станциями в том случае, когда эта связь осуществляется через промежуточную сеть, которая включает в себя другие станции и допускает соединение конечных станций по различным маршрутам; этот протокол относится к транспортному уровню модели OSI. Это различие между протоколами оказывает двойное влияние на их функциональные возможности.

1. *Обрабатываемые ошибки.* Для этих двух протоколов будут рассмотрены различные классы ошибок передачи данных. Сообщения не могут обгонять друг друга при физическом соединении, и они также не могут дублироваться; поэтому в § 3.3.1 рассматриваются только ошибки потери сообщения (об искажении сообщений говорится чуть ниже). В сетях сообщения могут перемещаться по разным маршрутам, и поэтому они могут обгонять друг друга; кроме того, вследствие неисправностей в работе промежуточных станций сообщения могут как дублироваться, так и утрачиваться. Поэтому в § 3.3.2 будут рассматриваться ошибки утраты, дублирования и переупорядочения сообщений.

2. *Управление соединением.* Мы не будем рассматривать управление соединением для первого протокола, этот вопрос существен для второго протокола. Предполагается, что физическое соединение обычно функционирует непрерывно в течение очень долгого времени, а не устанавливается и завершается периодически. Совсем по-другому происходит соединение с удаленными станциями. Потребность в таком соединении возникает на короткий срок для обмена некоторыми данными, но обычно поддерживать неопределенно долгое время соединение с каждой удаленной станцией — это слишком накладно. Поэтому второй протокол должен обладать способностью устанавливать и завершать соединения.

Изучение первого протокола показывает, что добиться требуемых свойств безопасности протоколов передачи данных можно без привлечения таймеров. В § 3.3.1 представлен первый развернутый пример обоснования свойств безопасности, опирающегося на те методы доказательства, которые были описаны в § 2.5.2. Принято считать (см. [199]), что для безопасного управления соединением необходимо использовать таймеры и устанавливать срок пребывания сообщения на этапе пересылки. Поэтому при обосновании свойств безопасности протоколов управления соединением нужно принимать в расчет ту роль, которую в них играют таймеры. В § 3.3.2 мы покажем, как можно ввести таймеры в модель распределенных систем (определение 2.6), и приведем пример одной из таких расширенных моделей.

Искажение сообщений. Вполне естественно, что во внимание должна быть принята вероятность того, что при передаче сообщения подвергаются искажениям. Содержание сообщения, передаваемого по физическому каналу связи, может быть повреждено вследствие атмосферных помех, нарушений работы устройств памяти и т. п. Тем не менее можно предполагать, что процесс-получатель способен обнаруживать искажения сообщений, например, при помощи счетчиков четности или более общих методов обнаружения ошибок (об этом рассказано в ра-

боте [182, гл. 3]). В таком случае к получению искаженного сообщения можно относиться так, как будто никакого сообщения не было получено, и это означает, что искажение сообщения может быть приравнено к потере сообщения. Поэтому мы не уделяем особого внимания искажениям сообщений, но всегда учитываем вероятность того, что сообщение может быть потеряно.

3.1. Симметричный протокол раздвижного окна

В этом параграфе изучается симметричный протокол, надежный двусторонний обмен информацией. Этот протокол позаимствован из работы [173, гл. 2]. Поскольку он служит для обмена информацией между станциями, соединенными прямой линией связи, можно предполагать, что в каналах связи поддерживается очередность сообщений. Однако это допущение не будет приниматься в расчет вплоть до § 3.1.3, в котором будет показано, что используемая в протоколе нумерация элементов последовательности может быть ограничена. Описание протокола приведено в § 3.1.1, а его корректность доказана в § 3.1.2.

Два взаимодействующих процесса будут обозначаться буквами p и q . Описание протокола, а также все допущения и требования относятся к процессу p в той же мере, в какой они относятся к процессу q . Входными данными процесса p является та информация, которую он должен отправить процессу q , эта информация моделируется бесконечным массивом слов in_p . Выходными данными процесса p является та информация, которую он получает от процесса q , и она также моделируется бесконечным вектором слов out_p . Считается, что p имеет произвольный доступ для чтения элементов массива in_p и произвольный доступ для записи в массив out_p . Первоначально элементы $out_p[i]$ для каждого i имеют неопределенное значение $undef$. Вход и выход процесса q соответственно моделируются массивами in_q и out_q . Для индексации элементов массивов используются натуральные числа, и нумерация элементов начинается с нуля. Как будет показано в § 3.1.3, вместо произвольного доступа к массивам можно ограничиться доступом к «окну» конечного размера, которое перемещается по массиву. Вот поэтому этот протокол часто называют «протоколом раздвижного окна».

В процессе p имеется переменная s_p , служащая для обозначения наименьшего номера того слова, которое процесс p все еще ожидает получить от процесса q . Таким образом, в каждый момент времени участок массива, начинающийся элементом $out_p[0]$ и оканчивающийся элементом $out_p[s_p - 1]$, уже заполнен процессом p . Значение переменной s_p не убывает. Совершенно аналогичная переменная s_q имеется в процессе q . Теперь мы можем сформулировать те требования, которые предъявляются к протоколу. Свойство безопасности предписывает каждому процессу формировать на выходе только правильные данные, а свойство живости предусматривает, что все данные рано или поздно достигнут назначения.

1. *Безопасная доставка данных.* В каждой достижимой конфигурации протокола выполняются соотношения

$$out_p[0..s_p - 1] = in_q[0..s_p - 1] \quad \text{и} \quad out_q[0..s_q - 1] = in_p[0..s_q - 1].$$

2. *Неизбежная доставка сообщений.* Для каждого целого числа $k \geq 0$ в ходе выполнения протокола будет достигнута конфигурация, в которой $s_p \geq k$ и $s_q \geq k$.

3.1.1. Описание протокола

Обычно в протоколах передачи данных значительную роль играют *подтверждающие сообщения*. Подтверждающее сообщение посылается процессом-получателем, чтобы оповестить процесс-отправитель о том, что данные были успешно доставлены по назначению. Если отправитель данных не получает подтверждение, он передает те же самые данные повторно исходя из предположения о том, что эти данные (или подтверждение об их получении) были потеряны. Однако в протоколе, который описан в этом параграфе, подтверждающие сообщения в явном виде не фигурируют. В рассматриваемом протоколе на обеих станциях есть сообщения, которые они должны отправить друг другу; для каждой станции ее входные данные используются для подтверждения получения сообщений от другой станции.

Сообщения, которыми обмениваются процессы, называются *пакетами*; пакет представляет собой набор вида $\langle \text{pack}, w, i \rangle$, где w — информационное слово, а i — натуральное число, которое называется *порядковым номером* пакета. Этот пакет, будучи отправленным процессом p (по назначению q), не только передает слово $w = in_p[i]$ процессу q , но также, как было отмечено выше, служит подтверждением тому, что ряд пакетов, отправленных процессом q , был успешно получен. Процесс p может «опережать» процесс q на некоторое заданное число пакетов l_p , если мы постановим, что отправление пакета $\langle \text{pack}, w, i \rangle$ процессом p подтверждает получение слов с номерами $0, \dots, i - l_p$ от процесса q . (То же самое значение придается пакетам, которые отправляются процессом q .) Константы l_p и l_q — это неотрицательные целые числа, известные процессам p и q . Использование пакетов данных в качестве подтверждающих сообщений имеет двойные последствия для переходов рассматриваемого протокола.

1. Процесс p может отправить слово $in_p[i]$ (в пакете $\langle \text{pack}, in_p[i], i \rangle$) только после того, как будут занесены в память все слова, начиная с $out_p[0]$ и заканчивая $out_p[i - l_p]$, т. е. когда будет выполняться неравенство $i < s_p + l_p$.

2. Как только p получает пакет $\langle \text{pack}, w, i \rangle$, отпадает необходимость в повторной передаче слов, начиная с $in_p[0]$ и заканчивая $in_p[i - l_q]$.

Пояснения к псевдокоду. После того как мы сформулировали общие принципы устройства протокола, написать псевдокод протокола уже нетрудно; он представлен в алгоритме 3.1. Переменная a_p в процессе p (равно как и переменная a_q в процессе q) используется для обозначения наименьшего порядкового номера того входного слова, для которого процесс p (или соответственно процесс q) еще не получил подтверждающего сообщения. В алгоритме 3.1 действие S_p осуществляет отправку i -го входного слова процесса p , действие R_p осуществляет прием слова процессом p , а действие L_p моделирует потерю пакета, адресатом которого является процесс p . Процесс p может отправить любое

слово, порядковый номер i которого находится в тех пределах, которые указаны в действии S_p . Когда процесс получает некоторое сообщение, он вначале проверяет, не было ли такое же сообщение получено ранее (в этом случае процесс имеет дело с повторным получением сообщения). Если это не так, то слово, содержащееся в сообщении, записывается в выходной массив и при этом значения переменных a_p и s_p изменяются. В процессе q также имеются аналогичные действия S_q , R_q и L_q .

Инвариант протокола. Коммуникационная подсистема представлена двумя очередями: очередью пакетов Q_p , адресованных процессу p , и очередью пакетов Q_q , адресованных процессу q . Заметим, что при обновлении значения переменной s_p в действии R_p новое значение никогда не может быть меньшим, чем прежнее значение, т. е. значения переменной s_p не убывают. Чтобы показать, что алгоритм удовлетворяет тем требованиям, которые были сформулированы выше, покажем сначала, что утверждение P является инвариантом. И в этом, и в последующих утверждениях i может принимать только натуральные значения.

$$\begin{aligned}
P \equiv & \quad \forall i < s_p : out_p[i] \neq undef & (0p) \\
& \wedge \forall i < s_q : out_q[i] \neq undef & (0q) \\
& \wedge \langle \text{pack}, w, i \rangle \in Q_p \implies w = in_q[i] \wedge (i < s_q + l_q) & (1p) \\
& \wedge \langle \text{pack}, w, i \rangle \in Q_q \implies w = in_p[i] \wedge (i < s_p + l_p) & (1q) \\
& \wedge out_p[i] \neq undef \implies out_p[i] = in_q[i] \wedge (a_p > i - l_q) & (2p) \\
& \wedge out_q[i] \neq undef \implies out_q[i] = in_p[i] \wedge (a_q > i - l_p) & (2q) \\
& \wedge a_p \leq s_q & (3p) \\
& \wedge a_q \leq s_p & (3q)
\end{aligned}$$

Лемма 3.1. Утверждение P является инвариантом алгоритма 3.1.

Доказательство. В каждой начальной конфигурации очереди Q_p и Q_q пусты, для всякого i значения $out_p[i]$ и $out_q[i]$ равны $undef$, а значения переменных a_p , a_q , s_q и s_p равны 0; отсюда следует истинность утверждения P . Рассмотрим поочередно все переходы протокола и покажем, что при каждом из них утверждение P сохраняет истинность. Прежде всего отметим, что значения массивов in_p и in_q никогда не изменяются.

S_p : Чтобы убедиться, что действие S_p сохраняет выполнимость условия (0p), заметим, что это действие не изменяет значения переменной s_p и не записывает в массив $out_p[i]$ неопределенного значения $undef$.

Чтобы убедиться, что действие S_p сохраняет выполнимость условия (0q), заметим, что S_p не увеличивает значения переменной s_q и не записывает в массив $out_q[i]$ неопределенного значения $undef$.

Чтобы убедиться, что действие S_p сохраняет выполнимость условия (1p), заметим, что S_p не добавляет новых пакетов в очередь Q_p и не уменьшает значения переменной s_q .

Чтобы убедиться, что действие S_p сохраняет выполнимость условия (1q), заметим, что S_p добавляет пакет $\langle \text{pack}, w, i \rangle$ с номером $i < s_p + l_p$, содержащий слово $w = in_p[i]$, в очередь Q_q и при этом значение переменной s_p не изменяется.

```

var  $s_p, a_p$  : integer           init 0, 0 ;
     $in_p$        : array of word    (* Data to be sent *) ;
     $out_p$       : array of word    init undef, undef, ... ;

```

```

 $S_p$ : {  $a_p \leq i < s_p + l_p$  }
      begin send  $\langle \text{pack}, in_p[i], i \rangle$  to  $q$  end

```

```

 $R_p$ : {  $\langle \text{pack}, w, i \rangle \in Q_p$  }
      begin receive  $\langle \text{pack}, w, i \rangle$  ;
        if  $out_p[i] = undef$  then
          begin  $out_p[i] := w$  ;
                 $a_p := \max(a_p, i - l_q + 1)$  ;
                 $s_p := \min \{j \mid out_p[j] = undef\}$ 
          end
        (* else игнорировать повторное получение пакета *)
      end

```

```

 $L_p$ : {  $\langle \text{pack}, w, i \rangle \in Q_p$  }
      begin  $Q_p := Q_p \setminus \{\langle \text{pack}, w, i \rangle\}$  end

```

Алгоритм 3.1. Симметричный протокол раздвижного окна (для процесса p).

Чтобы убедиться, что действие S_p сохраняет выполнимость условий (2p) и (2q), заметим, что S_p не изменяет значений переменных out_p , out_q , a_p и a_q .

Чтобы убедиться, что действие S_p сохраняет выполнимость условий (3p) и (3q), заметим, что S_p не изменяет значений переменных a_p , a_q , s_q и s_p .

R_p : Чтобы убедиться, что действие R_p сохраняет выполнимость условия (0p), заметим, что R_p не записывает в массив $out_p[i]$ неопределенного значения $undef$, и если это действие изменяет значение переменной s_p , то условие (0p) сохраняет при этом выполнимость.

Чтобы убедиться, что действие R_p сохраняет выполнимость условия (0q), заметим, что R_p не изменяет значений переменных out_q и s_q .

Чтобы убедиться, что действие R_p сохраняет выполнимость условия (1p), заметим, что R_p не добавляет новых пакетов в очередь Q_p и не уменьшает значения переменной s_q .

Чтобы убедиться, что действие R_p сохраняет выполнимость условия (1q), заметим, что R_p не добавляет новых пакетов в очередь Q_q и не уменьшает значения переменной s_p .

Чтобы убедиться, что действие R_p сохраняет выполнимость условия (2p), заметим, что R_p записывает слово w в элемент $out_p[i]$ выходного массива при получении пакета $\langle \text{pack}, w, i \rangle$. Так как в очереди Q_p уже содержался этот пакет, до того как выполнилось действие R_p , из условия (1p) следует, что $w = in_q[i]$. Присваивание $a_p := \max(a_p, i - l_q + 1)$ гарантирует, что неравенство $a_p > i - l_q$ будет справедливо и после выполнения этого действия.

Чтобы убедиться, что действие R_p сохраняет выполнимость условия (2q), заметим, что R_p не изменяет значений переменных out_q и a_q .

Чтобы убедиться, что действие $\mathbf{R}_p$ сохраняет выполнимость условия (3p), заметим, что, когда $\mathbf{R}_p$ присваивает переменной a_p значение $\max(a_p, i - l_q + 1)$ (при получении пакета $\langle \text{pack}, \omega, i \rangle$), из условия (1p) следует, что неравенство $i < s_q + l_q$ останется верным, и, значит, неравенство $a_p \leq s_q$ будет соблюдаться после этого присваивания. Действие $\mathbf{R}_p$ не изменяет значения переменной s_q .

Чтобы убедиться, что действие $\mathbf{R}_p$ сохраняет выполнимость условия (3q), заметим, что значение переменной s_p может лишь увеличиться после выполнения действия $\mathbf{R}_p$.

$\mathbf{L}_p$: Чтобы убедиться, что действие $\mathbf{L}_p$ сохраняет выполнимость условий (0p), (0q), (2p), (2q), (3p) и (3q), достаточно заметить, что $\mathbf{L}_p$ не изменяет состояний обоих процессов.

Выполнимость условий (1p) и (1q) сохраняется, поскольку действие $\mathbf{L}_p$ лишь удаляет пакеты из очереди, но не вставляет и не искажает никаких пакетов.

Аналогично можно проверить, что действия $\mathbf{S}_q$, $\mathbf{R}_q$ и $\mathbf{L}_q$ также сохраняют истинность утверждения P . $\square$

3.1.2. Доказательство корректности протокола

Теперь мы покажем, что алгоритм 3.1 обеспечивает надежную (свойство безопасности) и неизбежную (свойство живости) доставку сообщений. Как будет установлено в теореме 3.2, свойство безопасности следует из инварианта, а вот с доказательством свойства живости дело обстоит сложнее.

Теорема 3.2. *Алгоритм 3.1 удовлетворяет требованию безопасной доставки сообщений.*

Доказательство. Легко видеть, что из условий (0p) и (2p) следует справедливость равенства $out_p[0..s_p - 1] = in_q[0..s_p - 1]$, а из условий (0q) и (2q) следует справедливость равенства $out_q[0..s_q - 1] = in_p[0..s_q - 1]$. $\square$

Чтобы убедиться в выполнимости свойства живости протокола, необходимо ввести в рассмотрение допущения справедливости, а также ввести ограничения на значения l_p и l_q . Без этих допущений протокол не будет обладать свойством живости, и вот почему. В качестве l_p и l_q можно взять любые неотрицательные константы; но если обе константы будут равны 0, то протокол будет заблокирован уже в своей конфигурации (т. е. каждая начальная конфигурация будет также и заключительной). Поэтому мы будем считать, что соблюдается неравенство $l_p + l_q > 0$.

Условимся, что запись $\gamma = (c_p, c_q, Q_p, Q_q)$, где c_p и c_q — состояния процессов p и q , будет обозначать конфигурацию протокола. Рассмотрим произвольную конфигурацию γ , в которой допустимо действие $\mathbf{S}_p$ (при некотором значении i). Пусть

$$\delta = \mathbf{S}_p(\gamma) = (c_p, c_q, Q_p, (Q_q \cup \{m\})).$$

Заметим, что действие $\mathbf{L}_q$ является допустимым в конфигурации δ . Как только действие $\mathbf{L}_q$ промоделирует потерю сообщения m , будет выполняться соотношение $\mathbf{L}_q(\delta) = \gamma$. Равенство $\mathbf{L}_q(\mathbf{S}_p(\gamma)) = \gamma$ открывает возможность для осуществле-

ния бесконечного вычисления, в котором значения обеих переменных s_p и s_q не будут увеличиваться.

Протокол будет удовлетворять требованию неизбежной доставки сообщений, если принять во внимание следующие два допущения справедливости.

F1. Если бесконечно часто возникает возможность отправки пакета, то этот пакет будет отправляться бесконечно часто.

F2. Если один и тот же пакет отправляется бесконечно часто, то и принимается он также бесконечно часто.

Допущение F1 гарантирует, что всякий пакет будет отправляться снова и снова, если не будет получено подтверждение его получения; допущение F2 исключает возможность описанных выше вычислений, в которых ни одно из повторных отправлений одного и того же пакета не приводит к его доставке.

Ни один из двух процессов не может уйти вперед от другого слишком далеко: разность величин s_p и s_q остается ограниченной. Вот поэтому рассматриваемый протокол называется симметричным; если требование неизбежной доставки сообщений выполняется для переменной s_p , то оно также выполняется и для s_q , и наоборот. Это означает также, что данный протокол нельзя применять в тех случаях, когда одному из процессов нужно отправить существенно больше слов, чем другому.

Лемма 3.3. *Из инварианта P следует неравенство $s_p - l_q \leq a_p \leq s_q \leq a_q + l_p \leq s_p + l_p$.*

Доказательство. Из свойств (0p) и (2p) вытекает неравенство $s_p - l_q \leq a_p$. Свойство (3p) обеспечивает выполнимость неравенства $a_p \leq s_q$. Из свойств (0q) и (2q) следует неравенство $s_q \leq a_q + l_p$. И, наконец, из соотношения (3q) следует неравенство $a_q + l_p \leq s_p + l_p$. $\square$

Теорема 3.4. *Алгоритм 3.1 удовлетворяет требованию неизбежной доставки сообщений.*

Доказательство. Прежде всего убедимся в том, что протокол никогда не может быть заблокирован. Как следует из инварианта, один из двух процессов может отправлять пакет, содержащий слово с наименьшим порядковым номером, из тех, которые еще не достигли другого процесса.

Утверждение 3.5. *Из инварианта P следует, что допустимыми действиями являются отправление пакета $\langle \text{pack}, in_p[s_q], s_q \rangle$ процессом p или отправление пакета $\langle \text{pack}, in_q[s_p], s_p \rangle$ процессом q .*

Доказательство. В рамках допущения $l_p + l_q > 0$ по крайней мере одно из неравенств леммы 3.3 является строгим, т. е.

$$s_q < s_p + l_p \quad \vee \quad s_p < s_q + l_q.$$

Из инварианта P также следуют неравенства $a_p \leq s_q$ (3p) и $a_q \leq s_p$ (3q), и поэтому справедливо соотношение

$$(a_p \leq s_q < s_p + l_p) \quad \vee \quad (a_q \leq s_p < s_q + l_q),$$

которое означает, что либо действие S_p допустимо для $i = s_q$, либо действие S_q допустимо для $i = s_p$. $\square$

Теперь мы можем показать, что в любом вычислении значения переменных s_p и s_q возрастают бесконечно часто. Согласно утверждению 3.5 протокол не достигает заключительных конфигураций, и поэтому каждое вычисление продолжается бесконечно долго. Допустим, что существует вычисление C , в котором значения переменных s_p и s_q увеличиваются лишь конечное число раз, и пусть σ_p и σ_q — наибольшие значения, которые могут принимать эти переменные по ходу вычисления C . Согласно доказанному утверждению либо отправление пакета $\langle \text{pack}, in_p[\sigma_q], \sigma_q \rangle$ процессом p , либо отправление пакета $\langle \text{pack}, in_q[\sigma_p], \sigma_p \rangle$ процессом q допустимо бесконечно долго, после того как переменные s_p, s_q, a_p и a_q примут свои окончательные значения. Таким образом, согласно допущению F1 один из этих пакетов отправляется бесконечно часто, а согласно допущению F2 он должен приниматься также бесконечно часто. Но поскольку получение процессом p пакета с порядковым номером s_p приводит к тому, что значение s_p увеличивается (и то же самое справедливо для процесса q), возникает противоречие с предположением о том, что значения переменных s_p и s_q больше не будут увеличиваться. Этим завершается доказательство теоремы 3.4¹⁾. $\square$

В заключение этого параграфа мы обсудим коротко допущения F1 и F2. Допущение F2 представляет собой минимальное требование, которое предъявляется к качеству канала связи, соединяющего процессы p и q , для того чтобы по этому каналу мог осуществляться обмен данными. Очевидно, что если какое-нибудь слово $in_p[i]$ нельзя передать по этому каналу, то обеспечить его неизбежную доставку невозможно. Допущение F1 обычно реализуется в протоколе посредством условия тайм-аута: если значение переменной a_p не увеличивается на протяжении определенного отрезка времени, то сообщение $in_p[a_p]$ передается повторно. Как уже было упомянуто во введении в настоящую главу, обосновать свойство безопасности для рассматриваемого протокола можно без привлечения понятия времени.

3.1.3. Обсуждение протокола

Ограничение памяти процессов. Алгоритм 3.1 в том виде, в котором он был представлен, непригоден для реализации в компьютерных сетях, ввиду того что в каждом процессе хранится бесконечный объем информации (массивы in и out), а так же потому, что в нем используются неограниченные порядковые номера.

¹⁾Доказательство проведено не вполне корректно. Из приведенных рассуждений всего лишь следует, что в каждом вычислении значение хотя бы одной из переменных s_p или s_q будет увеличиваться бесконечно часто, в то время как нам нужно показать, что это будет иметь место сразу для *обеих* переменных. Все дело в том, что в предложенном доказательстве «от противного» было сделано неверное допущение. Правильнее было бы предположить, что существует вычисление C , в котором значения *хотя бы одной* из переменных s_p и s_q увеличиваются лишь конечное число раз. Тогда, согласно лемме 3.3 значение другой переменной также не может увеличиваться бесконечно часто. Далее по тексту. — Прим. перев.

На самом деле, как будет показано далее, в каждый момент времени процессам достаточно хранить лишь ограниченное количество слов. Пусть $L = l_p + l_q$.

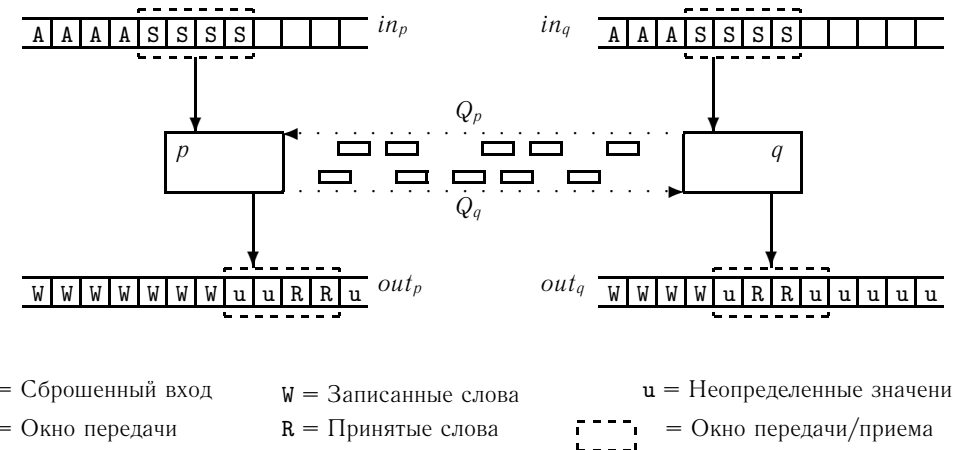
Лемма 3.6. Из инварианта P следует, что отправление пакета $\langle \text{pack}, w, i \rangle$ процессом p допустимо только тогда, когда $i < a_p + L$.

Доказательство. Действие S_p сопровождается логическим условием $i < s_p + l_p$; выполнимость этого условия, как следует из леммы 3.3, влечет за собой выполнимость неравенства $i < a_p + L$. $\square$

Лемма 3.7. Из инварианта P следует, что в том случае, когда $out_p[i] \neq \text{undef}$, выполняется неравенство $i < s_p + L$.

Доказательство. Согласно свойству (2p) справедливо соотношение $a_p > i - l_q$. Выполнимость соотношения $i < a_p + l_q$, как следует из леммы 3.3, влечет выполнимость неравенства $i < a_p + L$. $\square$

Эти две леммы приводят нас к следующей схеме, изображенной на рис. 3.2. Процесс p должен хранить в памяти только слова из отрезка $in_p[a_p..s_p + l_p - 1]$, поскольку он может отправлять только эти слова. Указанный отрезок назовем *окном передачи* процесса p (на рис. 3.2 оно обозначено буквами **S**). Как только значение переменной a_p увеличивается, p избавляется от тех слов, которые выходят за рамки окна передачи (на рис. 3.2 оно обозначено буквами **A**). Как только значение переменной s_p увеличивается, p прочитывает следующие слова в окне отправления из того источника, который генерирует эти слова. Согласно лемме 3.6 окно отправления процесса p содержит не более L слов.



A = Сброшенный вход **W** = Записанные слова **u** = Неопределенные значения
S = Окно передачи **R** = Принятые слова **[]** = Окно передачи/приема

Рис. 3.2. Протокол раздвижного окна

Подобного же рода доводы позволяют ограничить объем памяти, необходимый процессу p для хранения массива out_p . Коль скоро значения тех элементов $out_p[i]$, для которых выполняется неравенство $i < s_p$, остаются неизменными,

можно предположить, что процесс p проводит окончательную выдачу этих слов и перестает хранить их в памяти (на рис. 3.2 эти слова обозначены буквами W). Коль скоро $out_p[i] = undef$ для всех тех i , для которых выполняется неравенство $i \geq s_p + L$, эти элементы массива $out_p[i]$ также не нужно хранить в памяти процесса. Отрезок массива $out_p[s_p..s_p + L - 1]$ назовем *окном приема* процесса p . Окно приема обозначено на рис. 3.2 буквами u для тех элементов, значение которых еще не определено, и буквами R для тех слов, которые уже были получены. В памяти процесса хранятся только те слова, которые находятся в рамках окна приема. Леммы 3.6 и 3.7 свидетельствуют о том, что каждый процесс в любой момент времени хранит не более $2L$ слов.

Ограничение порядковых номеров. Наконец, мы покажем, что в том случае, если каналы поддерживают очередность передачи сообщений, порядковые номера слов также можно ограничить. Воспользовавшись гипотезой об очередности передачи сообщений, можно показать, что порядковый номер пакета, который получает процесс p , отстоит от значения переменной s_p не более чем на $2L$. Мы обращаем внимание читателей на то, что это первый случай, когда нам пришлось обратиться к допущению об очередности передачи сообщений.

Лемма 3.8. Утверждение P' , определяемое следующей формулой:

$P' \equiv P$

$\wedge \langle \mathbf{pack}, w, i \rangle \text{ следует за } \langle \mathbf{pack}, w', i' \rangle \text{ в очереди } Q_p \implies i > i' - L \quad (4p)$

$\wedge \langle \mathbf{pack}, w, i \rangle \text{ следует за } \langle \mathbf{pack}, w', i' \rangle \text{ в очереди } Q_q \implies i > i' - L \quad (4q)$

$\wedge \langle \mathbf{pack}, w, i \rangle \in Q_p \implies i \geq a_p - l_p \quad (5p)$

$\wedge \langle \mathbf{pack}, w, i \rangle \in Q_q \implies i \geq a_q - l_q, \quad (5q)$

является инвариантом алгоритма 3.1.

Доказательство. Так как ранее нам удалось показать, что утверждение P является инвариантом, можно ограничиться обоснованием того, что свойства (4p), (4q), (5p) и (5q) выполняются в начальных конфигурациях и сохраняются при каждом переходе. Заметим, что в каждой начальной конфигурации очереди сообщений пусты, и поэтому свойства (4p), (4q), (5p) и (5q), очевидно, выполняются. Остается показать, что они сохраняются при переходах системы.

S_p : Чтобы убедиться, что действие S_p сохраняет выполнимость условий (4p) и (5p), заметим, что S_p не добавляет новых пакетов в очередь Q_p и не изменяет значения переменной a_p .

Чтобы убедиться, что действие S_p сохраняет выполнимость условия (5q), заметим, что если S_p добавляет пакет $\langle \mathbf{pack}, w, i \rangle$ в очередь Q_q , то $i \geq a_p$. Согласно лемме 3.3 отсюда следует, что $i \geq a_q - l_q$.

Чтобы убедиться, что действие S_p сохраняет выполнимость условия (4q), заметим, что если пакет $\langle \mathbf{pack}, w', i' \rangle$ уже находится в очереди Q_q , то согласно свойству (1q) имеет место неравенство $i' < s_p + l_p$. Поэтому если S_p добавляет в очередь пакет $\langle \mathbf{pack}, w, i \rangle$ с порядковым номером $i \geq a_p$, то согласно лемме 3.3 будет выполняться неравенство $i' < a_p + L \leq i + L$.

R_p : Чтобы убедиться, что действие R_p сохраняет выполнимость условий (4p) и (4q), заметим, что R_p не добавляет никаких пакетов ни в очередь Q_p , ни в очередь Q_q .

Чтобы убедиться, что действие R_p сохраняет выполнимость условия (5p), заметим, что при увеличении значения переменной a_p (после приема пакета $\langle \mathbf{pack}, w', i' \rangle$) до величины $i' - l_q + 1$ для всех оставшихся пакетов $\langle \mathbf{pack}, w, i \rangle$ из очереди Q_p будет выполняться неравенство $i > i' - L$ (в силу свойства (4p)). Следовательно, неравенство $i \geq a_p - l_p$ сохраняет свою выполнимость и после увеличения значения переменной a_p .

Чтобы убедиться, что действие R_p сохраняет выполнимость условия (5q), заметим, что R_p не изменяет состава очереди Q_q и значения переменной a_q .

L_p : Действие L_p не добавляет никаких пакетов в очереди Q_p и Q_q и не изменяет значений переменных a_p и a_q . Поэтому оно сохраняет выполнимость условий (4p), (4q), (5p) и (5q).

Как следует из симметричности рассматриваемого протокола, действия S_q , R_q и L_q также сохраняют справедливость утверждения P' . $\square$

Лемма 3.9. Из утверждения P' вытекают следующие соотношения:

$$\langle \mathbf{pack}, w, i \rangle \in Q_p \implies s_p - L \leq i < s_p + L$$

и

$$\langle \mathbf{pack}, w, i \rangle \in Q_q \implies s_q - L \leq i < s_q + L.$$

Доказательство. Рассмотрим пакет $\langle \mathbf{pack}, w, i \rangle \in Q_p$. Согласно свойству (1p) имеем неравенство $i < s_q + l_q$, из которого по лемме 3.3 следует неравенство $i < s_p + L$. Согласно свойству (5p) имеем неравенство $i \geq a_p - l_p$, из которого по лемме 3.3 следует неравенство $i \geq s_p - L$. Аналогичным образом доказывается соотношение о пакетах из очереди Q_q . $\square$

Как видно из этой леммы, достаточно, чтобы нумерация пакетов проводилась по модулю k , где $k \geq 2L$. Действительно, располагая значением переменной s_p , а также значением переменной i по модулю k , процесс p может вычислить номер i .

Выбор параметров. Эффективность протокола в значительной степени зависит от выбора констант l_p и l_q . Вероятностный анализ зависимости пропускной способности протокола от этих констант был исследован в работе [173, гл. 2]. Оптимальные значения констант зависят от ряда системотехнических параметров, таких как

- 1) *время связи*, т. е. промежуток времени между двумя последовательными операциями процесса,
- 2) *задержка подтверждения приема*, т. е. среднее время, которое требуется процессу p для передачи пакета процессу q и получения от него ответа,
- 3) *вероятность ошибки*, т. е. вероятность потери пакета при передаче.

Протокол чередующихся битов. Особо интересный вариант протокола раздвижного окна возникает в том случае, когда $L = 1$, т. е. когда $l_p = 1$ и $l_q = 0$ (или наоборот). В качестве начальных значений переменных a_p и a_q в этом случае

выбирается не 0, а числа $-l_p$ и $-l_q$. Можно показать, что равенства $a_p + l_q = s_p$ и $a_q + l_p = s_q$ будут всегда сохраняться, и поэтому в протоколе достаточно использовать только одну из двух переменных a_p или s_p (и только одну из переменных a_q или s_q). Если воспользоваться таймерами, которые позволят станциям отправлять сообщения поочередно, то мы получим таким образом широко известный *протокол чередующихся битов*; см. [135].

3.2. Протокол с таймерами

Теперь мы перейдем к изучению той роли, которая отводится таймерам при проектировании и верификации коммуникационных протоколов; для этого рассмотрим упрощенный вариант Δt -протокола сквозной передачи сообщений Флетчера и Ватсона. Этот протокол был описан в работе [85], но та его слегка упрощенная версия, с которой мы будем иметь дело в этом разделе, заимствована из книги [187, § 3.2]. Этот протокол поддерживает не только механизм обмена данными (как это делает симметричный протокол раздвижного окна, рассмотренный нами в § 3.3.1), но также обеспечивает открытие и завершение соединений. Протокол обладает устойчивостью относительно потери и дублирования сообщений, а также относительно нарушения порядка следования сообщений.

Информация о состоянии протокола (вернее, той его части, которая обеспечивает передачу данных) содержится в структуре данных, которая называется *досье соединения*. (В § 3.2.1 будет рассказано о том, какая именно информация содержится в досье соединения). Досье соединения создается и стирается при открытии и закрытии соединения. Таким образом, соединение считается открытым (одной из станций), если существует досье этого соединения.

Желая сосредоточить внимание читателей на наиболее существенных особенностях протокола (а именно, на механизме управления соединением и на той роли, которая в этом механизме отведена таймерам), мы ограничимся рассмотрением упрощенного варианта протокола. В работах [85] и [187, § 3.2] можно найти обсуждение того, каким образом этот протокол должен быть расширен, для того чтобы отвечать практическим потребностям. Упрощения, о которых идет речь, касаются следующих четырех аспектов этого протокола.

1. *Однонаправленность*. Рассматривается передача данных только в одном направлении, а именно от процесса p к процессу q . Иногда мы будем называть процесс p *отправителем*, а процесс q — *получателем*. При этом, однако, нужно иметь в виду, что в протоколе используются подтверждающие сообщения, которые следуют в обратном направлении от процесса q к процессу p .

Но обычно приходится иметь дело с двусторонним обменом данными. Для этого приходится заводить второй протокол, в котором процессы p и q меняются ролями. И в этом случае можно ввести комбинированные сообщения, в которых содержатся как данные (сопровождаемые порядковым номером), так и информация, входящая в состав подтверждающего пакета.

2. *Окно приема шириной в одно слово*. Получатель не проводит запись пакетов данных, имеющих больший порядковый номер, чем тот, который им ожидается. Поступивший пакет принимается в расчет и немедленно доставляется по

назначению только в том случае, когда его порядковый номер совпадает с ожидаемым номером. Более сложные варианты протокола позволяют проводить запись преждевременно поступивших пакетов с большими порядковыми номерами и доставлять их по назначению, после того как будут получены и доставлены пакеты с меньшими порядковыми номерами.

3. *Упрощенные допущения о времени*. В рассматриваемом протоколе используется минимальное количество таймеров. Например, предполагается, что подтверждение может быть отправлено в любое время, до тех пор пока получатель поддерживает соединение открытым. Однако возможны ситуации, при которых подтверждение должно быть отправлено спустя небольшой период времени; учет этого обстоятельства приводит к дополнительному усложнению протокола.

Кроме того, мы исключили из описания протокола специальные таймеры, которые используются для запуска повторной передачи пакетов данных, как это было сделано в § 3.3.1. В протокол включены лишь те таймеры, которые необходимы для обеспечения свойств безопасности протокола.

4. *Пакеты, состоящие из одного слова*. В каждый пакет данных отправитель может поместить одно-единственное слово. Протокол можно сделать более эффективным, если разрешить помещать в пакет целый блок последовательно идущих слов.

В рассматриваемом нами протоколе используются таймеры, т. е. процессы имеют доступ к устройствам, измеряющим физическое время. Мы будем опираться на следующие основные предположения о времени и таймерах.

5. *Глобальное время*. Все процессы системы функционируют в рамках единой глобальной шкалы времени; это означает, что каждое событие происходит в некоторый вполне определенный момент времени. Каждое событие происходит мгновенно, т. е. его длительность равна 0, и при этом процессы *не обладают* способностью регистрировать те моменты времени, в которые происходят события.

6. *Ограниченное время жизни пакета*. Время жизни пакета ограничено некоторой константой μ (*максимальное время жизни пакета*). Поэтому, если какой-то пакет был отправлен в момент времени σ и получен в момент времени τ , то верно неравенство

$$\sigma < \tau < \sigma + \mu.$$

Если в канале произошло дублирование пакета, то каждая копия должна быть получена спустя не более μ единиц времени после отправления исходного пакета (в противном случае копии будут утрачены).

7. *Таймеры*. Сами по себе процессы не обладают способностью регистрировать абсолютное время выполнения своих действий, но они имеют доступ к *таймерам*. В роли таймера выступает вещественная переменная, значение которой со временем постоянно убывает (или присваивается этой переменной явным образом). Точнее говоря, если переменная Xt — это таймер, то мы будем использовать запись $Xt^{(t)}$ для обозначения значения этой переменной в момент времени t , и если в период времени между моментами t_1 и t_2 переменной Xt не было при-

своео какое-либо другое значение, то верно равенство

$$Xt^{(t_1)} - Xt^{(t_2)} = t_2 - t_1.$$

Следует обратить внимание на то, что таймеры отсчитывают время точно, т. е. за период времени длительности δ их значения уменьшаются в точности на δ . В §3.2.3 мы обсудим, как поступать в том случае, когда таймеры подвержены расхождению во времени.

Как и в §3.3.1, входные слова, подлежащие отправлению, моделируются бесконечным массивом in_p . И точно так же предполагается, что этот массив никогда не содержится целиком в процессе-отправителе p ; в каждый момент времени процесс p имеет доступ только к некоторой части массива. Та часть массива in_p , которая доступна процессу p , неуклонно расширяется (номера слов возрастают), по мере того как p получает все новые и новые слова от процесса, порождающего эти слова. Мы условимся называть эту операцию *поступлением* слова к процессу-отправителю.

В этом параграфе мы будем моделировать слова, доставленные процессу-получателю, совсем не так, как это было сделано в §3.3.1. Вместо того, чтобы осуществлять запись слов в (бесконечный) массив, процесс-получатель будет вручать эти слова процессу-потребителю посредством операции, которую условимся называть *вручением* слова. Наш замысел состоит в том, чтобы каждое слово из массива in_p было вручено в точности один раз, и при этом порядок вручения слов должен быть правильным.

Спецификация нашего протокола будет, однако, не столь требовательной, и причина этого состоит в том, что протоколу разрешается обрабатывать каждое слово из массива in_p в течение ограниченного интервала времени. Никакой протокол не может предоставить гарантии того, что слово будет доставлено по назначению за ограниченный срок времени, поскольку не исключена возможность того, что за это время все пакеты будут потеряны. Поэтому в спецификации нашего протокола предусмотрена возможность *зарегистрированной потери*, при которой протокол отправителя выдает сообщение об ошибке, свидетельствующее о том, что слово могло быть потеряно. (Если в связи с этим сообщением протокол высокого уровня предоставит процессу p это слово повторно, то это, естественно, приведет к дублированию; однако здесь мы не будем утруждать себя рассмотрением этой проблемы.) Интересующие нас свойства протокола, которые будут доказаны в §3.2.2 таковы.

1. *Отсутствие потерь.* Каждое слово из массива in_p будет вручено процессом q или зарегистрировано процессом p (как «вероятно потерянное») спустя ограниченный отрезок времени с момента поступления этого слова к процессу p .

2. *Соблюдение порядка.* Слова, которые вручаются процессом q , следуют в порядке строгого возрастания номеров в массиве in_p .

3.2.1. Описание протокола

Протокол открывает сеанс связи (соединение) всякий раз, когда соединение отсутствует, но при этом либо к отправителю поступает некоторое слово, либо

Сетевая константа:

μ : **real**; (* Максимальное время жизни пакета *)

Константы протокола:

U : **real**;

(* Продолжительность периода отправления сообщения *)

R : **real**;

(* Продолжительность перерыва при приеме сообщения: $R \geq U + \mu$ *)

S : **real**;

(* Продолжительность перерыва при передаче сообщения: $S \geq R + 2\mu$ *)

Учетные записи отправителя:

Low : **integer**; (* Подтвержденные слова текущего сеанса связи *)

$High$: **integer**; (* Поступившие слова текущего сеанса связи *)

St : **timer**; (* Таймер отправителя *)

Учетные записи получателя:

Exp : **integer**; (* Очередной ожидаемый порядковый номер *)

Rt : **timer**; (* Таймер получателя *)

Коммуникационная подсистема:

M_q : **channel**; (* Пакеты данных для процесса q *)

M_p : **channel**; (* Пакеты подтверждений для процесса p *)

Вспомогательные переменные:

B : **integer init** 0; (* Слова из предыдущего сеанса связи *)

cr : **bool init** false; (* Участие получателя в сеансе связи *)

cs : **bool init** false; (* Участие отправителя в сеансе связи *)

Рис. 3.3. Переменные протокола с таймерами

получателю передается пакет, который должен быть вручен. Поэтому в нашем протоколе, для того чтобы открыть сеанс связи, нет необходимости обмениваться контрольными сообщениями перед отправлением пакета. За счет этого достигается определенная эффективность протокола в тех случаях, когда в каждом сеансе связи совершается передача всего лишь нескольких слов (в режиме передачи информации мелкими пакетами). Предикат cs (соответственно cr) имеет значение true, когда отправитель (или получатель) открыл сеанс связи. Вообще-то эти предикаты не задаются в явном виде булевыми переменными отправителя (или получателя); вместо этого признаком открытого сеанса связи служит существование его учетной записи. Для того чтобы выяснить, открыт ли сеанс связи, процесс проверяет, существует ли учетная запись соединения в списке записей открытых сеансов связи.

Когда отправитель открывает сеанс связи, он начинает нумерацию поступивших к нему слов с 0. Переменная $High$ предназначена для подсчета количества слов, поступивших в текущем сеансе связи, а переменная Low служит для

A_p: (* Поступление очередного слова *)
begin if not cs then
 begin(* Вначале открывается сеанс связи (соединение) *)
 create (*St*, *High*, *Low*) ; (* cs := true *)
 Low := *High* := 0 ; *St* := *S*
 end;
 Ut[*B* + *High*] := *U* ; *High* := *High* + 1
end

S_p: (* Отправить *i*-е слово текущего сеанса связи *)
{ *cs* ∧ *Low* ≤ *i* < *High* ∧ *Ut*[*B* + *i*] > 0 }
begin send ⟨**data**, (*i* = *Low*), *i*, *in_p*[*B* + *i*], *μ*⟩ ;
St := *S*
end

R_p: (* Получить подтверждение *)
{ *cs* ∧ ⟨**ack**, *i*, *ρ*⟩ ∈ *M_p* }
begin receive ⟨**ack**, *i*, *ρ*⟩ ; *Low* := max(*Low*, *i*) **end**

E_p: (* Сформировать сообщение об ошибке в связи с возможной потерей слова *)
{ *cs* ∧ *Ut*[*B* + *Low*] ≤ −2*μ* − *R* }
begin *error*[*B* + *Low*] := true ; *Low* := *Low* + 1 **end**

C_p: (* Закрыть сеанс связи *)
{ *cs* ∧ *St* < 0 ∧ *Low* = *High* }
begin *B* := *B* + *High* ; delete (*St*, *High*, *Low*) **end**
(* *cs* := false *)

Алгоритм 3.4. Протокол процесса-отправителя

подсчета количества слов, подтвержденных в текущем сеансе связи. Тем самым предполагается (так же как в протоколе, описанном в §3.3.1), что отправитель может пересылать пакеты, порядковые номера которых расположены в интервале между двумя числами *Low* и *High* − 1; однако здесь имеются некоторые особенности. Отправитель может переслать слово только в течение интервала времени длины *U*, начиная с момента поступления этого слова. Для этого с каждым словом *in_p*[*i*] ассоциируется таймер *Ut*[*i*]; в момент поступления слова в таймере устанавливается начальное значение — число *U*, и в момент отправления слова таймер должен иметь положительное значение. Таким образом, окно передачи состоит из всех тех слов, у которых порядковые номера расположены в диапазоне *Low* ... *High* − 1, и соответствующие этим словам таймеры имеют положительные значения.

Пакеты данных, которые отправляются по ходу работы рассматриваемого протокола, состоят из особого бита (так называемого *признака начала последовательности*; его назначение мы обсудим позднее), порядкового номера и слова. Чтобы проводить анализ протокола, в каждом пакете выделяется четвертое поле, которое называется *оставшимся сроком жизни пакета*. В этом поле указывается максимальное время, на протяжении которого пакет может оставаться

R_q: (* Принять пакет данных *)
{ ⟨**data**, *s*, *i*, *w*, *ρ*⟩ ∈ *M_q* }
begin receive ⟨**data**, *s*, *i*, *w*, *ρ*⟩ ;
 if cr then
 if i = Exp then
 begin *Rt* := *R* ; *Exp* := *i* + 1 ; deliver *w* **end**
 else if s = true then
 begin create (*Rt*, *Exp*) ; (* *cr* := true *)
 Rt := *R* ; *Exp* := *i* + 1 ; deliver *w*
 end
 end
 end

S_q: (* Отправить подтверждение *)
{ *cr* }
begin send ⟨**ack**, *Exp*, *μ*⟩ **end**
(* Закрыть сеанс связи, если возник тайм-аут по таймеру *Rt*, см. действие **Time** *)

Алгоритм 3.5. Протокол процесса-получателя

в канале, до того как он будет доставлен по назначению или потерян в силу предположения об ограниченном времени жизни пакета. Когда пакет отправлен, оставшийся срок его жизни всегда равен *μ*. В пакет с подтверждением, который отправляется по ходу работы протокола, вкладывается только следующий порядковый номер сообщения, ожидаемого процессом *q*; но, как и в предыдущем случае, для проведения анализа мы будем считать, что в пакете с подтверждением также указывается оставшееся время жизни этого пакета.

Закрытием сеансов связи управляют таймеры: таймер *St* у отправителя и таймер *Rt* у получателя. Исходя из того, что для каждого слова имеется интервал, ограничивающий время его отправления, и каждый пакет также имеет ограниченный период жизни, можно заключить, что каждое слово может быть обнаружено в канале связи только на протяжении отрезка времени длины *μ* + *U*, начиная с момента поступления этого слова. Это позволяет получателю сбрасывать всю информацию о том или ином слове спустя *μ* + *U* единиц времени после получения этого слова; по истечении указанного срока никакие дубликаты уже не могут поступать и, следовательно, нет никакого риска столкнуться с повторным получением сообщения. Всякий раз, когда принимается некоторое слово, на таймере *Rt* устанавливается время *R*, причем константа *R* выбирается так, чтобы выполнялось неравенство *R* ≥ *U* + *μ*. Если следующее слово будет получено спустя не более *R* единиц времени, показания таймера *Rt* обновляются; в противном случае сеанс связи закрывается. Установка таймера отправителя осуществляется так, чтобы избежать ситуации, при которой подтверждение поступает при закрытом сеансе связи; для этого соединение поддерживается по меньшей мере *S* единиц времени после отправления очередного пакета, причем константа *S* выбирается так, чтобы выполнялось неравенство *S* ≥ *R* + 2*μ*. Всякий раз, когда отправляется некоторый пакет, на таймере *St* устанавливается время *S*, и сеанс связи закрывается только тогда, когда *St* < 0. Если в это время передача неко-

торых слов все еще не завершена (т. е. для некоторых отправленных слов не было получено подтверждений), эти слова заносятся в отчет, до того как сеанс связи будет закрыт.

Признак начала последовательности используется получателем в том случае, когда пакет получен при закрытом сеансе связи, для того чтобы решить, можно ли открыть сеанс связи (и принять содержащееся в пакете слово). Отправитель устанавливает в этом бите значение true, когда все предыдущие слова уже получили подтверждение или были занесены в отчет (как вероятно утраченные). Если процесс q получает пакет при уже открытом сеансе связи, то содержащееся в пакете слово принимается в том и только том случае, когда порядковый номер пакета равен очередному номеру ожидаемого пакета (для этого используется переменная Exp).

Loss: $\{ m \in M \}$ (* M — это либо M_p , либо M_q *)
begin remove m from M **end**

Dupl: $\{ m \in M \}$ (* M — это либо M_p , либо M_q *)
begin insert m in M **end**

Time: (* $\delta > 0$ *)
begin forall i do $Ut[i] := Ut[i] - \delta$;
 $St := St - \delta$;
 $Rt := Rt - \delta$;
 if $Rt \leq 0$ **then** delete (Rt , Exp) ; (* $cr := false$ *)
 forall $\langle \dots, \rho \rangle \in M_p, M_q$ **do**
 begin $\rho := \rho - \delta$;
 if $\rho \leq 0$ **then** remove packet
 end
 end
end

Алгоритм 3.6. Дополнительные переходы протокола

Остается обсудить, зачем нужна в протоколе отправления переменная B . Это вспомогательная переменная, которую мы ввели только для доказательства корректности протокола. Отправитель в каждом сеансе связи проводит нумерацию слов, начиная с 0, но для того, чтобы при анализе протокола проводить различие между словами в разных сеансах связи, мы будем нумеровать подряд все слова. Таким образом, в том случае, когда отправитель дает некоторому слову номер i , «абсолютный» номер этого слова будет равен $B + i$, где B — это суммарное число пакетов, поступивших процессу p в предыдущие сеансы связи. Соответствие между «внутренними» и «абсолютными» номерами слов показано на рис. 3.7. При реализации протокола для переменной B памяти не отводится, и отправитель «забывает» обо всех словах $in_p[0..B - 1]$.

Коммуникационная подсистема представлена двумя мультимножествами: M_p для пакетов, адресованных процессу p , и M_q для пакетов, адресованных процессу q . Протокол отправителя задается алгоритмом 3.4, а протокол получателя

задается алгоритмом 3.5. В системе есть также и дополнительные переходы, для описания которых служит алгоритм 3.6. Эти переходы не связаны с действиями протоколов процессов; с их помощью моделируются ошибки в каналах связи и течение времени. В переходах **Loss** и **Dupl** буква M служит для обозначения любого из двух мультимножеств M_p и M_q . Действие **Time** приводит к уменьшению показателей всех таймеров системы на одну и ту же величину δ , и это как раз соответствует тому, что происходит, когда два дискретных события оказываются разделенными интервалом времени протяженности δ . Как только показание таймера получателя оказывается равным 0, сеанс связи закрывается.

3.2.2. Доказательство корректности протокола

Для обоснования всех необходимых свойств протокола воспользуемся рядом лемм и теорем. Формула P_0 , приведенная ниже, выражает утверждение о том, что соединение отправителя остается открытым, до тех пор пока в системе остается хотя бы один пакет, и порядковые номера этих пакетов в текущем сеансе связи имеют правильное значение.

$$P_0 \equiv cs \implies St \leq S \quad (1)$$

$$\wedge cr \implies 0 < Rt \leq R \quad (2)$$

$$\wedge \forall i < B + High: Ut[i] \leq U \quad (3)$$

$$\wedge \forall \langle \dots, \rho \rangle \in M_p, M_q: 0 < \rho \leq \mu \quad (4)$$

$$\wedge \langle \text{data}, s, i, w, \rho \rangle \in M_q \implies cs \wedge St \geq \rho + \mu + R \quad (5)$$

$$\wedge cr \implies cs \wedge St \geq Rt + \mu \quad (6)$$

$$\wedge \langle \text{ack}, i, \rho \rangle \in M_p \implies cs \wedge St > \rho \quad (7)$$

$$\wedge \langle \text{data}, s, i, w, \rho \rangle \in M_q \implies (w = in_p[B + i] \wedge i < High) \quad (8)$$

Для правильного истолкования соотношения (3), мы будем предполагать, что значение $High$ равно нулю во всех тех конфигурациях, в которых отправитель не имеет соединения.

Лемма 3.10. Формула P_0 является инвариантом протокола с таймерами.

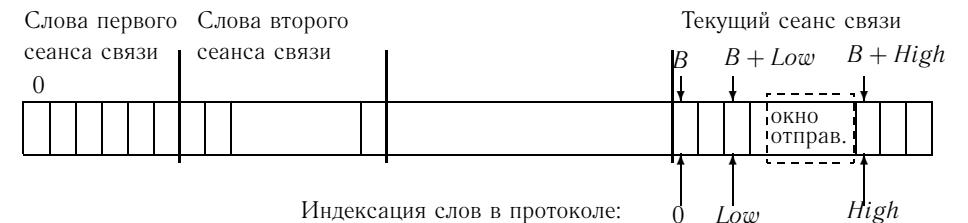


Рис. 3.7. Порядковые номера слов

Доказательство. В самом начале соединения отсутствуют, нет никаких пакетов, $B = 0$, и поэтому утверждение P_0 является истинным.

A_p : Выполнимость соотношения (1) сохраняется, поскольку оператор присваивания, в правой части которого стоит St , всегда приводит к тому, что выполняется равенство $St = S$. Выполнимость соотношения (3) сохраняется, потому что, перед тем как переменная $High$ увеличит свое значение, элементу $Ut[B + High]$ присваивается значение U . Выполнимость соотношений (5), (6) и (7) сохраняется, поскольку значение St может только увеличиться. Выполнимость соотношения (8) сохраняется, поскольку значение $High$ может только увеличиться.

S_p : Выполнимость соотношения (1) сохраняется, поскольку St всегда будет присвоено значение S . Выполнимость соотношения (4) сохраняется, потому что каждый отправляемый пакет имеет время оставшейся жизни, равное μ . Выполнимость соотношения (5) сохраняется, так как пакет $\langle \dots, \mu \rangle$ отправлен, St полагается равным S и $S = R + 2\mu$. Выполнимость соотношений (6) и (7) сохраняется, поскольку рассматриваемое действие может только увеличить значение St . Соотношение (8) сохраняется, так как для нового пакета выполняются условия $w = in_p[B + i]$ и $i < High$.

R_p : Действие R_p не изменяет значений ни одной из переменных, фигурирующих в формуле P_0 , и изъятие пакета сохраняет соотношения (4) и (7).

E_p : Действие E_p не изменяет значений ни одной из переменных, фигурирующих в формуле P_0 .

C_p : Действие C_p может нарушить справедливость заключений в соотношениях (5), (6) и (7), но (согласно соотношениям (2), (5), (6) и (7)) оно применимо только тогда, когда предпосылки этих соотношений не выполняются. Действие C_p также изменяет значение переменной B , но, поскольку согласно соотношениям (5) и (7) ни один пакет не находится на этапе пересылки, выполнимость соотношения (8) сохраняется.

R_q : Выполнимость соотношения (2) сохраняется, так как переменной Rt всегда присваивается значение R (если присваивание имеет место). Выполнимость соотношения (6) сохраняется, поскольку переменной Rt присваивается значение R только при получении пакета $\langle \text{data}, s, i, w, \rho \rangle$, а из соотношений (4) и (5) следует, что в этом случае выполняется формула $cs \wedge St \geq R + \mu$.

S_q : Выполнимость соотношения (4) сохраняется, потому что при отправлении каждого пакета оставшееся время его жизни полагается равным μ . Выполнимость соотношения (7) сохраняется, поскольку при отправлении пакета $\langle \text{ack}, i, \rho \rangle$ в случае, когда cr имеет значение $true$, выполняется равенство $\rho = \mu$, и поэтому из соотношений (2) и (6) следует неравенство $St > \mu$.

Loss: Выполнимость соотношений (4), (5), (7) и (8) сохраняется, потому что изъятие пакета может привести лишь к тому, что нарушатся предпосылки этих соотношений.

Dupl: Выполнимость соотношений (4), (5), (7) и (8) сохраняется, так как дублирующая вставка пакета m возможна только тогда, когда пакет m уже содержится в канале; отсюда вытекает, что следствия соответствующих соотношений были выполнены уже перед самой вставкой.

Time: Выполнимость соотношений (1), (2) и (3) сохраняется, поскольку значения переменных St , Rt и $Ut[i]$ могут лишь уменьшиться и сеанс связи получателя закрывается, когда значение Rt становится равным 0. Выполнимость соотношения (4) сохраняется, потому что значение ρ может лишь уменьшиться и пакет изымается, когда значение его поля ρ становится равным 0. Заметим, что действие **Time** приводит к уменьшению значения всех таймеров (включая те, которые содержатся в поле ρ пакетов) на *одну и ту же величину*; поэтому сохраняется выполнимость всех соотношений вида $Xt \geq Yt + C$, где Xt и Yt — таймеры, а C — константа. Это означает, что соотношения (5), (6) и (7) остаются верными. $\square$

Первое требование, которое предъявляется к рассматриваемому протоколу, состоит в том, что каждое слово должно быть рано или поздно доставлено по назначению или занесено в отчет как утраченное. Определим предикат $Ok(i)$ следующей формулой:

$$Ok(i) \iff error[i] = true \quad \vee \quad \text{процессу } q \text{ было доставлено сообщение } in_p[i].$$

Как будет показано, протокол не теряет ни одного слова, не отметив этого факта. Рассмотрим формулу P_1 следующего вида:

$$P_1 \equiv P_0$$

$$\wedge \neg cs \implies \forall i < B: Ok(i) \quad (9)$$

$$\wedge cs \implies \forall i < B + Low: Ok(i) \quad (10)$$

$$\wedge \langle \text{data}, true, I, w, \rho \rangle \in M_q \implies \forall i < B + I: Ok(i) \quad (11)$$

$$\wedge cr \implies \forall i < B + Exp: Ok(i) \quad (12)$$

$$\wedge \langle \text{ack}, I, \rho \rangle \in M_p \implies \forall i < B + I: Ok(i) \quad (13)$$

Лемма 3.11. Формула P_1 является инвариантом протокола с таймерами.

Доказательство. Прежде всего заметим, что, как только предикат $Ok(i)$ становится истинным для некоторого i , он больше никогда не может стать ложным. Вначале нет никакого соединения, нет пакетов и $B = 0$, вследствие чего выполняется утверждение P_1 .

A_p : Действие A_p может открыть сеанс связи, но выполнимость соотношения (10) при этом сохраняется, потому что при открытии сеанса связи $Low = 0$ и $\forall i < B: Ok(i)$ вследствие выполнимости соотношения (9).

S_p : Действие S_p может отправить пакет $\langle \text{data}, s, I, w, \rho \rangle$, но, ввиду того что s имеет значение $true$ только в том случае, когда $I = Low$, выполнимость соотношения (10) приводит к тому, что выполнимость соотношения (11) сохраняется.

R_p : Значение переменной Low может увеличиться, если будет получен пакет $\langle \text{ack}, I, \rho \rangle$. Тем не менее выполнимость соотношения (10) сохраняется, поскольку в случае получения этого подтверждения будет выполняться соотношению (13) $\forall i < B + I: Ok(i)$.

E_p : Значение переменной Low может увеличиться, когда выполняется действие E_p , но формирование сообщения об ошибке приводит к тому, что выполнимость соотношения (10) сохраняется.

C_p: Действие **C_p** приводит к тому, что *cs* получает значение false, но это действие применимо только тогда, когда $St < 0$ и $Low = High$. Из соотношения (10) следует, что неравенство $\forall i < B + High : Ok(i)$ соблюдается в момент выполнения действия **C_p**, и поэтому выполнимость соотношения (9) сохраняется. В результате выполнения этого действия предпосылка соотношения (10) становится ложной, и поэтому из соотношений (5), (6) и (7) следует, что и предпосылки соотношений (11), (12) и (13) также становятся ложными; отсюда следует, что выполнимость соотношений (10), (11), (12) и (13) сохраняется.

R_q: Вначале рассмотрим случай, когда процесс *q* получает пакет $\langle data, true, I, w, \rho \rangle$, в то время как никакого соединения нет (*cr* имеет значение false). Тогда согласно соотношению (11) имеет место неравенство $\forall i < B + I : Ok(i)$, и в результате этого действия будет доставлено слово *w*. Коль скоро согласно соотношению (8) верно равенство $w = in_p[B + I]$, в результате присваивания $Exp := I + 1$ выполнимость соотношения (12) сохраняется.

Теперь обратимся к случаю, когда значение *Exp* увеличивается в результате получения пакета $\langle data, s, Exp, w, \rho \rangle$ при открытом сеансе связи. Как следует из соотношения (12), перед получением сообщения справедливо неравенство $\forall i < B + Exp : Ok(i)$ и в ходе осуществления рассматриваемого действия будет получено слово $w = in_p[B + Exp]$. Следовательно, при увеличении значения *Exp* выполнимость соотношения (12) сохраняется.

S_q: Как следует из соотношения (12), при отправлении сообщения $\langle ack, Exp, \mu \rangle$ выполнимость соотношения (13) сохраняется.

Loss: Применение действия **Loss** приводит к тому, что предпосылки всех соотношений становятся ложными.

Dupl: Дублирующая вставка пакета *m* может осуществиться только в том случае, когда предпосылки соответствующих соотношений (а значит, и их следствия) были верны перед этой вставкой.

Time: В соотношениях (9)–(13) никакие таймеры не фигурируют. Изъятие пакета или закрытие сеанса связи процессом *q* может лишь сделать ложными предпосылки соотношений (11), (12) или (13). $\square$

Сделав дополнительное допущение, мы можем доказать теперь первую часть спецификации протокола. Без этого допущения отправитель может оказаться чересчур «ленивым» при составлении отчета о потерянных словах; в алгоритме 3.4 всего лишь указано, что сообщение о потере *не может быть* сформировано в течение $2\mu + R$ единиц времени, следующих за окончанием интервала отправления слова, но там не сказано о том, что это сообщение рано или поздно обязано быть сформировано. Поэтому давайте сделаем дополнительное допущение о том, что действие **E_p** рано или поздно будет выполнено процессом *p* в разумные сроки, скажем, *не позднее* $Ut[B + Low] = -2\mu - R - \lambda$.

Теорема 3.12 (отсутствие потерь). *Каждое слово массива in_p либо достигает процесса *q*, либо отмечается процессом *p* как утраченное в течение $U + 2\mu + R + \lambda$ единиц времени после поступления этого слова процессу *p*.*

Доказательство. При поступлении слова $in_p[I]$ выполняется неравенство $B + High > I$, и оно продолжает выполняться далее. Если сеанс связи закрывается спустя предписанный период времени после поступления слова $in_p[I]$, значит, $B > I$, и утверждение теоремы следует из соотношения (9). Если спустя указанный период времени сеанс связи остается открытым и $B + Low \leq I$, все слова, порядковые номера которых располагаются на отрезке $B + Low..I$, будут отмечены в отчете спустя $2\mu + R$ единиц времени по окончании интервала времени, отведенного для отправления слова $in_p[I]$. Отсюда следует, что запись в отчете будет сделана спустя $2\mu + R + \lambda$ единиц времени после окончания срока, отведенного для отправления сообщения, т. е. спустя $U + 2\mu + R + \lambda$ единиц времени после поступления слова. Это также влечет за собой выполнимость неравенства $I < B + Low$, и поэтому рассматриваемое слово будет либо отмечено в отчете, либо доставлено по назначению согласно соотношению (10). $\square$

Чтобы установить второе требование корректности протокола, необходимо показать, что каждое слово, которое было вручено потребителю, имеет больший порядковый номер (в массиве in_p), чем ранее врученное слово. Воспользуемся записью *pr* для обозначения порядкового номера самого последнего из врученных потребителю слов (для удобства будем полагать, что вначале $pr = -1$ и $Ut[-1] = -\infty$). Рассмотрим формулу P_2 , имеющую следующий вид:

$$\begin{aligned} P_2 \equiv & P_1 \\ & \wedge \langle data, s, i, w, \rho \rangle \in M_q \implies Ut[B + i] > \rho - \mu \quad (14) \\ & \wedge i_1 \leq i_2 < B + High \implies Ut[i_1] \leq Ut[i_2] \quad (15) \\ & \wedge cr \implies Rt \geq Ut[pr] + \mu \quad (16) \\ & \wedge pr < B + High \wedge (Ut[pr] > -\mu \implies cr) \quad (17) \\ & \wedge cr \implies B + Exp = pr + 1 \wedge \quad (18) \end{aligned}$$

Лемма 3.13. *Формула P_2 является инвариантом протокола с таймерами.*

Доказательство. Первоначально мультимножество M_q пусто, $B + High$ равно нулю, выполняются соотношения $\neg cr$ и $Ut[pr] < -\mu$; отсюда следует выполнимость соотношений (14)–(18).

A_p: Выполнимость соотношения (15) сохраняется, потому что только что доставленное слово сопровождается таймером *U*, показание которого согласно соотношению (3) по крайней мере равно показанию таймеров, сопровождающих ранее доставленные слова.

S_p: Выполнимость соотношения (14) сохраняется, так как $Ut[B + i] > 0$, и при отправлении пакета соблюдается равенство $\rho = \mu$.

C_p: Выполнимость соотношений (14), (16) и (18) сохраняется, поскольку согласно соотношениям (5) и (6) предпосылки этих соотношений неверны, когда применимо действие **C_p**. Выполнимость соотношения (15) сохраняется, потому что переменной *B* присвоено значение $B + High$ и показания таймеров не изменились. Выполнимость соотношения (17) сохраняется, ввиду того, что переменной *B* присвоено значение $B + High$, а значения *pr* и *cr* не изменяются.

R_q: Выполнимость соотношения (16) сохраняется, так как в том случае, когда переменной Rt присвоено значение R (после того как доставлено слово), неравенство $Ut[pr] \leq U$ соблюдается вследствие соотношения (3) и при этом имеет место неравенство $R \geq 2\mu + U$. Выполнимость соотношения (17) сохраняется, поскольку неравенство $pr < B + High$ следует из (8) и при этом cr принимает значение $true$. Выполнимость соотношения (18) сохраняется, так как значение Exp становится равным $i + 1$, а значение pr становится равным $B + i$; отсюда следует, что соотношение (18) становится истинным.

Time: Выполнимость соотношения (14) сохраняется, ввиду того что значения $Ut[B + i]$ и ρ уменьшаются на одну и ту же величину (а изъятие пакета может лишь сделать ложным предпосылку соотношения). Выполнимость соотношения (15) сохраняется, ввиду того что значения $Ut[i_1]$ и $Ut[i_2]$ уменьшаются на одну и ту же величину. Выполнимость соотношения (16) сохраняется, потому что это действие не обращает cr в истину и значения Rt и $Ut[pr]$ уменьшаются на одну и ту же величину. Выполнимость соотношения (17) сохраняется, поскольку его заключение может стать ложным только тогда, когда значение Rt становится не большим нуля, а это, согласно соотношению (16), означает, что значение $Ut[pr]$ не превосходит $-\mu$. Выполнимость соотношения (18) сохраняется, поскольку в том случае, когда cr не обращается в ложь, значения B , Exp и pr не изменяются.

Действия **R_p**, **E_p** и **S_q** не изменяют значений тех переменных, которые фигурируют в соотношениях (14)–(18). Действия **Loss** и **Dupl** сохраняют выполнимость соотношений (14)–(18); обоснование этого проводится так же, как и в предыдущих доказательствах. $\square$

Лемма 3.14. Из формулы P_2 следует соотношение

$$\langle data, s, i_1, w, \rho \rangle \in M_q \implies (cr \vee B + i_1 > pr).$$

Доказательство. Согласно соотношению (14) из условия $\langle data, s, i_1, w, \rho \rangle \in M_q$ следует, что $Ut[B + i_1] > \rho - \mu > -\mu$. Если $B + i_1 \leq pr$, то, принимая во внимание, что согласно соотношению (15) справедливо неравенство $pr < B + High$, получаем неравенство $Ut[pr] > -\mu$, из которого на основании соотношения (17) следует истинность предиката cr . $\square$

Теорема 3.15 (упорядочение). Слова, доставленные процессом q потребителю, расположены в строго возрастающем порядке в массиве in_p .

Доказательство. Предположим, что процесс q получает пакет $\langle data, s, i_1, w, \rho \rangle$ и вручает потребителю слово w . Если перед получением не было никакого соединения, то $B + i_1 > pr$ (по лемме 3.14), и поэтому слово w в массиве in_p располагалось в позиции, номер которой следует за pr . Если же соединение имелось, то $i_1 = Exp$, и поэтому согласно соотношению (18) имеют место равенства $B + i_1 = B + Exp = pr + 1$, из которых следует, что $w = in_p[pr + 1]$. $\square$

3.2.3. Обсуждение протокола

Некоторые расширенные варианты рассмотренного протокола уже обсуждались во введении к этому параграфу. В заключение мы продолжим обсуждение протокола, а также тех методов, которые были введены и использованы в этом параграфе.

Качественный анализ протокола. Требования *отсутствия потерь* и *упорядочения* являются свойствами безопасности, и для того чтобы удовлетворить эти требования, можно предложить чрезвычайно простое решение, а именно, протокол, который вообще не отправляет и не принимает ни одного пакета, а каждое слово заносит в отчет как утраченное. Нет нужды объяснять, почему такой протокол, который не обеспечивает передачи данных от отправителя к получателю, нельзя считать очень «хорошим» решением.

Хорошим считается такое решение, которое не только удовлетворяет требованиям *отсутствия потерь* и *упорядочения*, но также сообщает об утрате как можно меньшего числа слов. Чтобы достичь этой цели мы должны снабдить протокол, описанный в этом параграфе, некоторым механизмом, который позволял бы отправлять каждое слово повторно (на протяжении интервала отправления слова), до тех пор пока не будет получено подтверждение о его доставке. Интервал отправления должен быть достаточно протяженным, чтобы позволить осуществлять многократную передачу определенных слов и сделать тем самым вероятность утраты слова как можно меньшей.

Со стороны получателя должен срабатывать механизм, который запускает отправление подтверждающих сообщений при доставке пакета или его получении при открытом сеансе связи.

Ограниченные порядковые номера. Порядковые номера, используемые в протоколе, можно ограничить, доказав для этого протокола утверждение, аналогичное лемме 3.9 для симметричного протокола раздвижного окна [187, § 3.2]. Для этого необходимо ввести допущение о том, что скорость поступления слов на вход процесса p ограничена таким образом, чтобы очередное слово поступало спустя не менее $U + 2\mu + R$ единиц времени после поступления L -го по порядку предшествующего слова. Чтобы достичь этого, действие **A_p** нужно снабдить предохранителем

$$\{(High < L) \vee (Ut[B + High - L] < -R - 2\mu)\}.$$

В рамках такого допущения можно показать, что порядковые номера полученных пакетов данных отстоят не более чем на $2L$ от ожидаемого номера Exp , а порядковые номера в подтверждающих сообщениях отстоят не более чем на L от номера $High$. Следовательно, при передаче увеличение порядковых номеров можно проводить по модулю $2L$.

Облик действий и инвариантов. Теоретико-доказательный подход сводит анализ коммуникационного протокола к задаче проверки и преобразования формул (иногда весьма громоздких). Манипуляции с формулами — это очень «на-

дежный» прием, потому что на каждом шаге можно провести очень тщательную проверку и вероятность ошибки чрезвычайно мала. И все же есть определенный риск того, что при этом может быть потеряно общее понимание протокола и его связь с рассматриваемыми формулами. Вопрос об устройстве протокола можно осмыслить как с прагматической, так и с формальной точки зрения. Флетчер и Ватсон в работе [85] настаивают на том, что управляющая информация должна быть «защищена», имея в виду, что ее смысл не должен изменяться при утрате или дублировании пакетов; такова прагматическая точка зрения. При использовании теоретико-доказательного подхода к решению задачи верификации «смысл» управляющей информации проявляется в выборе тех или иных утверждений в качестве инвариантов. Формальный взгляд на устройство протокола определяется выбором этих инвариантов и устройством переходов, сохраняющих эти инварианты. На самом деле, как будет показано далее, замечание Флетчера и Ватсона можно переформулировать в терминах описания структуры формул, которые разрешается или не разрешается использовать в качестве инвариантов протокола, стойкого по отношению к потерям и дублированию пакетов.

Все соотношения (дизъюнкты), входящие в состав формулы P_2 и касающиеся пакетов, имеют следующий общий вид:

$$\forall m \in M : A(m),$$

и, как нетрудно убедиться, выполнимость соотношений такого вида сохраняется при дублировании или потере пакетов. В последующих главах мы будем рассматривать инварианты более общего вида, например

$$\sum_{m \in M} f(m) = K$$

или

$$\text{предусловие} \implies \exists m \in M : A(m).$$

Предложения такого вида могут утратить истинность в случае потери или дублирования пакетов, и поэтому их нельзя использовать при доказательстве корректности алгоритмов, от которых требуется устойчивость по отношению к подобным сбоям.

Аналогичное замечание касается устройства инвариантов, относящихся к действию **Time**. Как уже говорилось, это действие сохраняет выполнимость всех предложений вида

$$Xt \geq Yt + C,$$

где Xt и Yt — таймеры, а C — константа.

Неточные таймеры. Действие **Time** моделирует идеальные таймеры, значения которых уменьшается в точности на δ по истечении δ единиц времени, но на практике таймеры подвержены неточностям, которые называются *расхождением*. Обычно предполагается, что это расхождение является ϵ -ограниченным для некоторой известной константы ϵ . Это означает, что за период времени длины δ показания таймера уменьшаются на величину δ' , которая удовлетворяет неравенству $\delta/(1+\epsilon) \leq \delta' \leq \delta \times (1+\epsilon)$. (Обычно ϵ является величиной порядка 10^{-5}

Time- ϵ : $\{ \delta > 0 \}$

begin (* Значения всех таймеров процесса p уменьшаются на δ' *)

$$\delta' := \dots ; (* \frac{\delta}{(1+\epsilon)} \leq \delta' \leq \delta \times (1+\epsilon) *)$$

forall i **do** $Ut[i] := Ut[i] - \delta'$;

$St := St - \delta'$;

(* Значения всех таймеров процесса q уменьшаются на δ'' *)

$$\delta'' := \dots ; (* \frac{\delta}{(1+\epsilon)} \leq \delta'' \leq \delta \times (1+\epsilon) *)$$

$Rt := Rt - \delta''$;

if $Rt \leq 0$ **then** delete (Rt, Exp) ;

(* Поля ρ вычисляются точно *)

forall $\langle \dots, \rho \rangle \in M_p, M_q$ **do**

begin $\rho := \rho - \delta$;

if $\rho \leq 0$ **then** remove packet

end

end

Алгоритм 3.8. Модифицированное действие **Time**

или 10^{-6} .) Такое поведение таймеров моделируется действием **Time- ϵ** , которое описано алгоритмом 3.8.

Как уже отмечалось ранее, действие **Time** сохраняет выполнимость предложений вида $Xt \geq Yt + C$, потому что значения таймеров в обеих частях неравенства уменьшаются на одну и ту же величину, и неравенство $Xt \geq Yt + C$ влечет соотношение $(Xt - \delta) \geq (Yt - \delta) + C$. Аналогичное замечание относится и к действию **Time- ϵ** . Для действительных чисел $Xt, Yt, \delta, \delta', \delta'', r$ и c , удовлетворяющих неравенствам $\delta > 0$ и $r > 1$, соотношение

$$[Xt \geq r^2 Yt + c] \wedge \left(\frac{\delta}{r} \leq \delta' \leq \delta \times r \right) \wedge \left(\frac{\delta}{r} \leq \delta'' \leq \delta \times r \right)$$

влечет неравенство

$$(Xt - \delta') \geq r^2 (Yt - \delta'') + c.$$

Следовательно, действие **Time- ϵ** сохраняет выполнимость предложений вида

$$Xt \geq (1+\epsilon)^2 Yt + c.$$

Наш протокол теперь можно так модифицировать, чтобы он мог работать с расходящимися таймерами; инварианты при этом преобразуются соответствующим образом. Для того чтобы другие действия также сохраняли выполнимость модифицированных инвариантов, константы R и S , используемые в протоколе, должны удовлетворять соотношениям

$$R \geq (1+\epsilon)((1+\epsilon)U + (1+\epsilon)^2\mu) \text{ и } S \geq (1+\epsilon)(2\mu + (1+\epsilon)R).$$

В остальном, за исключением указанных констант, протокол остается без изменений. Его инвариант представлен на рис. 3.9.

$$\begin{aligned}
P'_2 \equiv & \quad cs \implies St \leq S & (1') \\
& \wedge cr \implies 0 < Rt \leq R & (2') \\
& \wedge \forall i < B + High: Ut[i] \leq U & (3') \\
& \wedge \forall \langle \dots, \rho \rangle \in M_p, M_q: 0 < \rho \leq \mu & (4') \\
& \wedge \langle data, s, i, w, \rho \rangle \in M_q \implies cs \wedge St \geq (1 + \epsilon)(\rho + \mu + (1 + \epsilon)R) & (5') \\
& \wedge cr \implies cs \wedge St \geq (1 + \epsilon)((1 + \epsilon)Rt + \mu) & (6') \\
& \wedge \langle ack, i, \rho \rangle \in M_p \implies cs \wedge St > (1 + \epsilon) \times \rho & (7') \\
& \wedge \langle data, s, i, w, \rho \rangle \in M_q \implies (w = in_p[B + i] \wedge i < High) & (8') \\
& \wedge \neg cs \implies \forall i < B: Ok(i) & (9') \\
& \wedge cs \implies \forall i < B + Low: Ok(i) & (10') \\
& \wedge \langle data, true, l, w, \rho \rangle \in M_q \implies \forall i < B + l: Ok(i) & (11') \\
& \wedge cr \implies \forall i < B + Exp: Ok(i) & (12') \\
& \wedge \langle ack, l, \rho \rangle \in M_p \implies \forall i < B + l: Ok(i) & (13') \\
& \wedge \langle data, s, i, w, \rho \rangle \in M_q \implies Ut[B + i] > (1 + \epsilon)(\rho - \mu) & (14') \\
& \wedge i_1 \leq i_2 < B + High \implies Ut[i_1] \leq Ut[i_2] & (15') \\
& \wedge cr \implies Rt \geq (1 + \epsilon)((1 + \epsilon)Ut[pr] + (1 + \epsilon)^2\mu) & (16') \\
& \wedge pr < B + High \wedge Ut[pr] > -(1 + \epsilon)\mu \implies cr & (17') \\
& \wedge cr \implies B + Exp = pr + 1 \wedge & (18')
\end{aligned}$$

Рис. 3.9. Инвариант протокола с расходящимися таймерами.

Теорема 3.16. Формула P'_2 является инвариантом протокола с таймерами, имеющими ϵ -ограниченное расхождение. Этот протокол удовлетворяет требованиям отсутствия потерь и упорядочения.

3.3. Упражнения к главе 3

3.3.1

Упражнение 3.1. Покажите, что симметричный протокол раздвижного окна не удовлетворяет требованию неперенной доставки сообщения, если из двух допущений справедливости F1 и F2 выполняется только допущение F2.

Упражнение 3.2. Докажите, что если в симметричном протоколе раздвижного окна $L = 1$ и начальные значения переменных a_p и a_q равны $-l_q$ и $-l_p$, то равенства $a_p + l_q = s_p$ и $a_q + l_p = s_q$ всегда выполняются.

3.3.2

Упражнение 3.3. В протоколе с таймерами отправитель может занести в отчет слово как возможно утраченное, в то время как это слово было благополучно доставлено получателю.

1. Опишите выполнение этого протокола, в ходе которого происходит подобный эффект.

2. Можно ли разработать такой протокол, в котором отправитель за ограниченное время составляет отчет об ошибке *в том и только том случае*, когда слово не доставляется получателю?

Упражнение 3.4. Предположим, что в связи с выходом из строя часового механизма получатель не может закрыть сеанс связи вовремя. Опишите вычисление протокола с таймерами, в ходе которого слово будет утрачено, но отправитель не сможет отметить это в отчете.

Упражнение 3.5. Опишите такое вычисление протокола с таймерами, в ходе которого получатель открывает сеанс связи после получения пакета с порядковым номером, большим нуля.

Упражнение 3.6. Действие **Time- ϵ** не моделирует расхождение оставшегося времени жизни пакета. Почему?

Упражнение 3.7. Докажите теорему 3.16.

Упражнение 3.8. Проектировщик сети хотел бы воспользоваться протоколом с таймерами, но при этом желает, чтобы о возможно утраченных словах запись в отчете осуществлялась ранее. Для этого он модифицирует действие E_p следующим образом:

E_p : (* Сформировать отчет об ошибке для возможно утраченного слова *)
 $\{ Ut[B + Low] < 0 \}$
begin $error[B + Low] := true$; $Low := Low + 1$ **end**

Будет ли модифицированный таким образом протокол удовлетворять требованиям отсутствия потерь и упорядочения или для этого необходимо внести также другие изменения? Каковы, по Вашему мнению, преимущества и недостатки указанной модификации?

Г Л А В А 4

АЛГОРИТМЫ МАРШРУТИЗАЦИИ

В общем случае процесс (узел в компьютерной сети) непосредственно не соединен каналами связи с каждым другим процессом. Из каждого узла пакеты информации могут непосредственно передаваться только некоторому подмножеству других узлов, которые называются *соседями* этого узла. Маршрутизация — это термин, который используется для описания процедуры принятия решения о том, какому соседу (иногда не единственному) следует переслать пакет, чтобы он в конце концов был доставлен по назначению. Цель, которая ставится при проектировании алгоритма маршрутизации, состоит в том, чтобы снабдить каждый узел процедурой, которая сможет выполнять эту функцию и гарантировать доставку каждого пакета.

Ясно, что в каждом узле должна храниться некоторая информация о топологии сети, и локальная процедура должна принимать решение на основе этой информации; эту информацию мы будем называть *таблицей маршрутизации*. Как только введены эти таблицы, задачу маршрутизации можно разбить на две алгоритмические составляющие; определение структуры таблицы, конечно, является составной частью разработки алгоритма.

1. *Вычисление таблиц*. Таблицы маршрутизации должны быть вычислены при инициализации сети и должны быть обновлены при изменении топологии сети.

2. *Продвижение пакета*. Когда пакет пересылается по сети, для его продвижения используются таблицы маршрутизации.

Критерии оценки качества методов маршрутизации учитывают следующие показатели.

1. *Корректность*. Алгоритм должен доставлять каждый пакет, поступивший в сеть, в точности по назначению.

2. *Эффективность*. Алгоритм должен отправлять пакеты по «наилучшим» путям; например, ожидается, что выбранные пути приводят к небольшой задержке и обеспечивают высокую пропускную способность всей сети. Алгоритм называется *оптимальным*, если он позволяет задействовать «наилучшие» пути.

3. *Сложность*. Алгоритм вычисления таблиц должен использовать как можно меньше сообщений, расходовать как можно более экономно время и память. Другим аспектам сложности, учитывающим, насколько быстро может быть построен маршрут, насколько быстро пакет может быть подготовлен для пересылки, и т. п., в этой главе будет уделяться меньше внимания.

4. *Устойчивость*. В случае изменения топологии (добавления или удаления канала или вершины) алгоритм вносит изменения в таблицы маршрутизации, для того чтобы задачу маршрутизации можно было решить в модифицированной сети.

5. *Адаптивность*. Для того чтобы выровнять загруженность каналов и вершин, алгоритм приспособливает к этому таблицы, избегая прокладывать пути через те каналы и вершины, которые чрезмерно загружены, и отдавая предпочтение тем каналам и вершинам, на которые в это время возложена наименьшая нагрузка.

6. *Справедливость*. Алгоритм должен обслуживать всех пользователей в равной мере.

Эти требования порой вступают в конфликт друг с другом, и большинство алгоритмов имеют хорошую производительность только относительно некоторого их подмножества.

Сеть, как обычно, представляется графом, вершины которого соответствуют узлам сети, и при этом соседние вершины соединены ребром, если между ними имеется канал связи. Оптимальность алгоритма зависит от того, какой путь в графе считается «наилучшим»; существует несколько вариантов понятия «наилучший путь», и для каждого из них имеется свой класс алгоритмов маршрутизации.

1. *Минимальное количество звеньев*. Стоимость использования пути определяется *количеством звеньев* (пройденных каналов или шагов от одной вершины к другой) в этом пути. Алгоритм маршрутизации с минимальным числом звеньев выбирает пути, имеющих наименьшее возможное число звеньев.

2. *Кратчайший путь*. Каждому каналу связи придается неизменный (неотрицательный) *вес*, и стоимость пути полагается равной суммарному весу всех каналов пути. Алгоритм кратчайшего пути выбирает путь наименьшей стоимости.

3. *Минимальная задержка*. Каждому каналу связи придается динамически изменяемая характеристика, зависящая от трафика через этот канал. Алгоритм минимальной задержки постоянно пересматривает таблицы так, чтобы всегда выбирались пути с (почти) минимальной общей задержкой. Так как задержка, приписанная каналам, зависит от фактического трафика, различные пакеты, переправляемые по сети, влияют друг на друга; степень этого влияния на алгоритмы маршрутизации мы обсудим в конце §4.6.1.

Другие варианты понятия оптимальности пути могут оказаться полезными в специальных приложениях, но мы не будем их обсуждать здесь.

Содержание главы. В этой главе будет рассматриваться следующий материал. В §4.6.1 будет показано, что по крайней мере для задачи маршрутизации с наименьшим числом звеньев и по кратчайшему пути можно проложить оптимальные маршруты для всех пакетов, адресованных одному и тому же адресату d , посредством остоного дерева с корнем в вершине d . Вследствие этого при выборе маршрута источник сообщения может не приниматься во внимание.

В §4.6.2 описан алгоритм вычисления таблиц маршрутизации для статической сети со взвешенными каналами. Алгоритм проводит распределенное вычисление кратчайшего пути между любыми двумя парами вершин и помещает

в память каждой вершины-источника первого соседа на пути к тому или иному узлу-адресату. Недостатком этого алгоритма является то, что все вычисление нужно проводить повторно всякий раз, когда топология сети изменяется, т. е. алгоритм неустойчив.

Алгоритм Netchange, который рассматривается в § 4.6.3, свободен от этого недостатка: он адаптируем к отказам и восстановлениям, возникающим в каналах связи, путем *частичного* перевычисления таблиц маршрутизации. Для того чтобы упростить анализ алгоритма, мы рассмотрим тот его вариант, в котором стоимость пути определяется числом звеньев в этом пути. Алгоритм Netchange можно модифицировать так, чтобы применять его и для случая взвешенных каналов, в которых возможны отказы и восстановления.

В алгоритмах маршрутизации, представленных в § 4.6.2 и 4.6.3, в каждой вершине используются таблицы маршрутизации в зависимости от возможных процессов-адресатов. Для больших сетей, состоящих из компактных вершин, хранение этой информации может быть обременительным. В § 4.6.4 будут рассмотрены некоторые стратегии маршрутизации, которые кодируют топологическую информацию в адресе вершины для сокращения объема таблиц маршрутизации и упрощения поиска в этих таблицах. Эти «компактные» алгоритмы маршрутизации обычно не используют оптимальных путей. Мы проанализируем древесные схемы, схемы интервальной и префиксной маршрутизации.

В § 4.6.5 рассматриваются иерархические методы маршрутизации. Эти методы предусматривают разбиение сети на связанные друг с другом кластеры; маршрутизация внутри кластера и между кластерами проводится по-разному. Такой подход позволяет сократить число решений, которые нужно принять, для того чтобы проложить маршрут или для того чтобы сократить объем памяти, необходимой для хранения таблиц маршрутизации в каждой вершине.

4.1. Маршрутизация на основе узлов-адресатов

Выбор маршрута для продвижения пакета обычно проводится только с учетом *узла-адресата* пакета (и содержимого таблиц маршрутизации) и *независимо* от исходного отправителя (источника) пакета. Как следует из результатов этого параграфа, при выборе маршрута можно не принимать в расчет источник и тем не менее построить оптимальный путь. Эти результаты не зависят от того, какому критерию оптимальности путей отдается предпочтение, при условии, что соблюдается следующее допущение. (Напомним, что путь называется *простым*, если каждая вершина встречается в нем не более одного раза, и путь является циклом, если первая его вершина совпадает с последней.)

1. Стоимость отправления пакета по пути P не зависит от того, как этот путь используется, и, в частности, от того, используются ли ребра пути P для продвижения других сообщений. В свете этого допущения стоимость использования пути P становится функцией, зависящей только от самого пути; в связи с этим условимся обозначать записью $C(P)$ стоимость пути P .

2. При последовательном соединении двух путей образуется новый путь, стоимость которого равна сумме стоимостей двух исходных путей, т. е. для всех $i =$

$= 0, \dots, k$ имеет место равенство

$$C(\langle u_0, u_1, \dots, u_k \rangle) = C(\langle u_0, \dots, u_i \rangle) + C(\langle u_i, \dots, u_k \rangle).$$

Таким образом, стоимость пустого пути $\langle u_0 \rangle$ (ведущего из вершины u_0 в вершину u_0) удовлетворяет условию $C(\langle u_0 \rangle) = 0$.

3. В графе отсутствуют циклы отрицательной стоимости.

(Этим допущениям удовлетворяют критерии стоимости минимального числа звеньев и кратчайшего пути.) Путь из вершины u в вершину v называется *оптимальным*, если не существует пути меньшей стоимости из u в v . Заметим, что оптимальный путь может быть не единственным; могут существовать различные пути, имеющие одну и ту же (минимальную) стоимость.

Лемма 4.1. Пусть u и v — вершины из множества V . Если в графе G есть путь P из u в v , то имеется также и простой путь из u в v , который является при этом оптимальным.

Доказательство. Так как существует конечное число простых путей, существует простой путь S_0 наименьшей стоимости из вершины u в вершину v ; здесь мы имеем в виду, что для каждого *простого* пути P' из u в v выполняется неравенство $C(S_0) \leq C(P')$. Остается показать, что $C(S_0)$ является нижней гранью стоимостей всех (не обязательно простых) путей.

Пусть $V = \{v_1, \dots, v_N\}$. Последовательно удаляя из пути P циклы, содержащие вершины v_1, v_2 , и т. д., можно прийти к заключению о том, что для каждого пути P из u в v существует простой путь P' , для которого выполняется неравенство $C(P') \leq C(P)$. Пусть $P_0 = P$. Построим для каждого $i = 1, \dots, N$ путь P_i следующим образом. Если вершина v_i встречается в пути P_{i-1} не более одного раза, то положим $P_i = P_{i-1}$. В противном случае рассмотрим путь $P_{i-1} = \langle u_0, \dots, u_k \rangle$, и пусть u_{j_1} — это первое вхождение, а u_{j_2} — это последнее вхождение вершины v_i в последовательность P_{i-1} . Тогда положим

$$P_i = \langle u_0, \dots, u_{j_1} (= u_{j_2}), u_{j_2+1}, \dots, u_k \rangle.$$

Согласно построению последовательность вершин P_i является путем из u в v , и при этом каждая из вершин $\{v_1, \dots, v_i\}$ встречается в ней не более чем один раз. Следовательно, P_N — это простой путь из вершины u в вершину v . Путь P_{i-1} состоит из пути ¹⁾ P_i и цикла $Q = \langle u_{j_1}, \dots, u_{j_2} \rangle$, и поэтому $C(P_{i-1}) = C(P_i) + C(Q)$. Так как циклы отрицательного веса отсутствуют, отсюда следует, что $C(P_i) \leq C(P_{i-1})$. Поэтому верно неравенство $C(P_N) \leq C(P)$.

Принимая во внимание особенность выбора пути S_0 , заключаем, что $C(S_0) \leq C(P_N)$; отсюда следует неравенство $C(S_0) \leq C(P)$. $\square$

Если в графе G есть циклы отрицательного веса, то оптимального пути может и не найтись; стоимость каждого пути можно улучшить, стоит только пройти лишний раз по циклу отрицательной стоимости. В приведенной ниже теореме

¹⁾ Точнее говоря, путь P_{i-1} образован в результате последовательного соединения трех путей — пути $P'_{i-1} = \langle u_0, \dots, u_{j_1} \rangle$, цикла $Q = \langle u_{j_1}, \dots, u_{j_2} \rangle$ и пути $P''_{i-1} = \langle u_{j_2}, \dots, u_k \rangle$. При этом соединение путей P'_{i-1} и P''_{i-1} как раз и образует путь P_i . — Прим. перев.

подразумевается, что граф G является связным (в случае несвязного графа теорема применяется к каждой его компоненте связности по отдельности).

Теорема 4.2. Для каждой вершины $d \in V$ существует такое дерево $T_d = (V, E_d)$, $E_d \subseteq E$, в котором для любой вершины $v \in V$ единственный путь из v в d в дереве T_d является оптимальным путем из v в d в графе G .

Доказательство. Пусть $V = \{v_1, \dots, v_N\}$. Мы построим индуктивно ряд деревьев $T_i = (V_i, E_i)$ (для $i = 0, \dots, N$), обладающих следующими свойствами.

1. Каждое дерево T_i является подграфом графа G , т. е. $V_i \subseteq V$, $E_i \subseteq E$.
 2. Каждое дерево T_i (для $i < N$) является поддеревом дерева T_{i+1} .
 3. Для каждого $i > 0$ выполняются включения $v_i \in V_i$ и $d \in V_i$.
 4. Для каждой вершины $w \in V_i$ простой путь из вершины w в вершину d в дереве T_i является оптимальным путем из w в d в графе G .
- Эти свойства гарантируют, что дерево T_N удовлетворяет тем требованиям, которые предъявляются к дереву T_d .

Чтобы построить последовательность деревьев, положим $V_0 = \{d\}$ и $E_0 = \emptyset$. Дерево T_{i+1} строится следующим образом. Выбирается оптимальный простой путь $P = \langle u_0, \dots, u_k \rangle$ из v_{i+1} в d ; обозначим символом l такой наименьший индекс, для которого выполняется включение $u_l \in T_i$ (такой индекс l существует, поскольку $u_k = d \in T_i$, при этом возможно $l = 0$). Положим

$$V_{i+1} = V_i \cup \{u_j : j < l\} \text{ и } E_{i+1} = E_i \cup \{(u_j, u_{j+1}) : j < l\}.$$

(Эта конструкция изображена на Рис. 4.1.) Нетрудно убедиться в том, что T_i является поддеревом графа T_{i+1} и $v_{i+1} \in V_{i+1}$. Граф T_{i+1} является деревом, поскольку по построению T_{i+1} является связным графом, и число вершин в нем превосходит число ребер в точности на единицу. (Граф T_0 обладает последним из упомянутых свойств, и на каждом этапе к графу добавляется столько же вершин, сколько и ребер.)

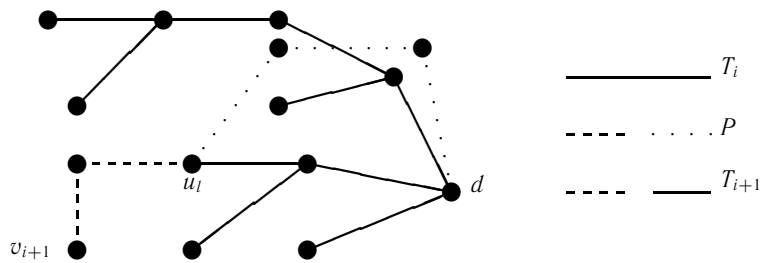


Рис. 4.1. Построение дерева T_{i+1} .

Остается показать, что для всех вершин $w \in V_{i+1}$ единственный путь из w в d в дереве T_{i+1} является оптимальным путем из w в d в графе G . Для вершин $w \in V_i \subset V_{i+1}$ это верно, ввиду того что T_i является поддеревом дерева T_{i+1} ,

и поэтому путь из w в d в дереве T_{i+1} тот же самый, что и путь в дереве T_i , который по индуктивному предположению является оптимальным. Теперь рассмотрим вершину $w = u_j$, $j < l$, принадлежащую множеству $V_{i+1} \setminus V_i$. Обозначим символом Q путь из вершины u_l в вершину d в дереве T_i . Тогда в дереве T_{i+1} из вершины u_j в вершину d ведет путь, полученный в результате последовательного соединения пути $\langle u_j, \dots, u_l \rangle$ и пути Q . Остается показать, что этот путь является оптимальным в графе G . Прежде всего заметим, что суффикс $P' = \langle u_l, \dots, u_k \rangle$ пути P сам по себе является оптимальным путем из вершины u_l в вершину d , и поэтому $C(P') = C(Q)$. Это следует из того, что оптимальность пути Q предполагает неравенство $C(P') \geq C(Q)$, а в случае $C(Q) < C(P')$ мы получаем (учитывая свойство аддитивности стоимости путей), что, присоединив к пути $\langle u_0, \dots, u_l \rangle$ путь Q , мы можем построить путь, имеющий меньшую стоимость, чем путь P , вопреки оптимальности пути P . Предположим теперь, что некоторый путь R из u_j в d имеет меньшую стоимость, нежели путь, полученный присоединением к пути $\langle u_j, \dots, u_l \rangle$ пути Q . Тогда, воспользовавшись ранее сделанным замечанием, мы можем прийти к заключению, что стоимость пути R меньше, чем стоимость суффикса $\langle u_j, \dots, u_k \rangle$ пути P , а это означает (с учетом аддитивности стоимости путей), что путь, образованный присоединением к пути $\langle u_0, \dots, u_j \rangle$ пути R , имеет меньшую стоимость, чем P , вопреки оптимальности пути P . $\square$

Остовное дерево, корнем которого служит вершина d , называется *входным деревом* для d , а дерево, обладающее теми свойствами, которые указаны в теореме 4.2 называется *оптимальным входным деревом*. Существование оптимального входного дерева означает, что алгоритмы маршрутизации, в основу которых положен механизм продвижения, описанный в алгоритме 4.2, не поступаются оптимальностью. В этом алгоритме ведущую роль играет локальная процедура $table\_lookup_u$, имеющая один аргумент и выбирающая одного из соседей вершины u (после обращения к таблице маршрутизации). Действительно, ввиду того что все пакеты, адресованные узлу d , можно оптимально направить по остовному дереву, корнем которого служит вершина d , продвижение пакета будет оптимальным, если для каждой вершины $u \neq d$ процедура $table\_lookup_u(d)$ будет выдавать в качестве значения родительскую вершину узла u в остовном дереве T_d .

Если механизм продвижения пакетов устроен именно так и топология сети не претерпевает (дальнейших) изменений, для доказательства корректности таблиц маршрутизации можно воспользоваться следующим результатом. Будем говорить, что таблицы маршрутизации (для узла-адресата d) *содержат цикл*, если существуют такие вершины $u_1, \dots, u_k$, что

- $u_i \neq d$ для всех i ,
- $table\_lookup_{u_i}(d) = u_{i+1}$ для всех $i < k$ и
- $table\_lookup_{u_k}(d) = u_1$.

Таблицы называются *ациклическими*, если они не содержат цикла ни для одной вершины d .

Лемма 4.3. *Механизм продвижения доставляет каждый пакет по назначению в том и только том случае, когда таблицы маршрутизации являются ациклическими.*

Доказательство. Если таблицы маршрутизации содержат цикл для некоторого узла-адресата d , то пакет, адресованный процессу d , никогда не будет доставлен по назначению, если отправителем является вершина, входящая в цикл.

(* Пакет, адресованный вершине d , был получен или сформирован в вершине u *)
if $d = u$

then доставить пакет локальными средствами

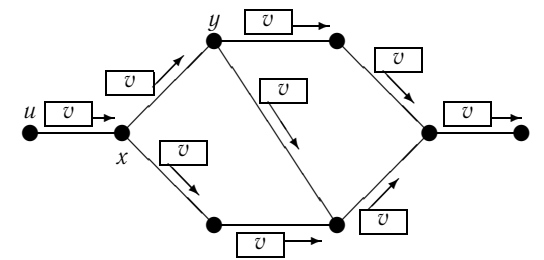
else отправить пакет вершине $table\_lookup_u(d)$

Алгоритм 4.2. Продвижение пакетов по назначению (для узла u).

Предположим, что таблицы являются ациклическими и пакет, предназначенный узлу-адресату d (и отправленный из узла-источника u_0), продвинулся через вершины $u_0, u_1, u_2, \dots$. Если одна и та же вершина встречается в этой последовательности дважды, скажем $u_i = u_j$, то таблицы содержат цикл, в данном случае $\langle u_i, \dots, u_j \rangle$, вопреки предположению об ациклическости таблиц. Значит, каждая вершина встречается в последовательности не более одного раза. Отсюда следует, что эта последовательность является конечной и оканчивается некоторой вершиной u_k ($k < N$). В соответствии с описанием процедуры продвижения пакета указанная последовательность может оканчиваться только вершиной d . Таким образом, $u_k = d$, и пакет достигает вершины адресата, пройдя не более $N - 1$ звеньев. $\square$

В некоторых алгоритмах маршрутизации может случиться так, что таблицы на этапе их вычисления не обладают свойством ациклическости, но только до тех пор, пока не завершится этап вычисления таблиц. При применении такого алгоритма пакет может продвигаться по циклу, пока вычисляются таблицы, но тем не менее доставляется по назначению не более чем за $N - 1$ звеньев, после того как вычисление таблиц завершается при отсутствии дальнейших изменений топологии. Если изменения топологии не прекращаются, т. е. структура сети подвергается постоянным преобразованиям, пакеты могут не достичь своего назначения даже в том случае, если обновляемые таблицы оказываются ациклическими (см. упражнение 4.1).

Разветвляемая маршрутизация с минимальной задержкой. Если требуется провести маршрутизацию по путям с наименьшей задержкой, и при этом задержка в канале связи зависит от его загруженности (и поэтому допущение 1, сделанное в начале этого параграфа является недействительным), то стоимость использования пути нельзя рассматривать как функцию, зависящую только от этого пути. Кроме того необходимо принимать в расчет трафик по каналу связи. Чтобы избежать заторов на пути (из-за которых возникает большая задержка), пакеты, имеющие один и тот же источник и одного того же адресата, продвигают-



Трафик в вершине v расщепления в вершинах x и y .

Рис. 4.3. Пример разветвляемой маршрутизации

ся различными путями; в этом случае трафик расщепляется в одной или нескольких промежуточных вершинах, как это показано на рис. 4.3. Маршрутизация, в которых пакеты, направленные по одному и тому же адресу, продвигаются разными путями, называются *многопутевой маршрутизацией* или *расщепленной маршрутизацией*. Ввиду того что методы расщепленной маршрутизации, как правило, очень сложны, мы не будем рассматривать их в этой главе.

4.2. Задача построения кратчайших путей для всех пар вершин

В этом параграфе мы рассмотрим алгоритм Туэга [194] совместного вычисления таблиц маршрутизации для всех узлов сети. Для каждой пары вершин (u, v) алгоритм вычисляет длину кратчайшего пути из вершины u в вершину v и заносит в память процесса u первый канал этого пути. Это хорошо известная задача построения кратчайших путей для всех пар вершин. В основу распределенного алгоритма Туэга решения этой задачи положен централизованный алгоритм Флойда—Уоршалла [55, § 26.4]. Мы рассмотрим алгоритм Флойда—Уоршалла в § 4.2.1, а затем в § 4.2.2 обратимся к алгоритму Туэга. В § 4.2.3 мы обсудим и другие алгоритмы решения задачи построения кратчайших путей для всех пар вершин.

4.2.1. Алгоритм Флойда—Уоршалла

Предположим, что имеется взвешенный граф $G = (V, E)$, в котором каждому ребру uv приписан вес ω_{uv} . Ясно, что $\omega_{uv} = \omega_{vu}$, и, кроме того, мы будем считать, что в графе отсутствуют циклы отрицательного веса. Весом пути $\langle u_0, \dots, u_k \rangle$ называется число, равное $\sum_{i=0}^{k-1} \omega_{u_i u_{i+1}}$. *Расстоянием* между вершинами u и v называется наименьший вес $d(u, v)$ пути, соединяющего вершины u и v (если таких путей нет, то расстояние считается бесконечным, $d(u, v) = \infty$). Задача построения кратчайших путей для всех пар вершин состоит в том, чтобы вычислить расстояние $d(u, v)$ для каждой пары вершин u и v . (В § 4.2.2 мы расширим возможности алгоритма, и он будет заносить в память процесса первое ребро пути наименьшего веса.)

Для вычисления всех расстояний в алгоритме Флойда—Уоршалла используется понятие S -путей; это пути, в которых все *промежуточные* вершины принадлежат подмножеству S множества вершин V .

Определение 4.4. Пусть S — некоторое подмножество множества вершин V графа G . Путь $\langle u_0, \dots, u_k \rangle$ называется S -путем, если $u_i \in S$ для каждого i , $0 < i < k$. S -расстоянием между вершинами u и v , которое обозначается $d^S(u, v)$, называется наименьший вес S -пути между u и v (если таких путей нет, то расстояние считается бесконечным, $d^S(u, v) = \infty$).

Работа алгоритма начинается с построения всех $\emptyset$ -путей, а затем множество S -путей наращивается для все более обширных подмножеств S , до тех пор пока не будут рассмотрены все V -пути. Уместно принять во внимание следующее замечание.

Предложение 4.5. Для любой вершины u и всякого подмножества S выполняется равенство $d^S(u, u) = 0$. Кроме того, если $u \neq v$, то S -пути подчиняются следующим правилам:

- 1) $\emptyset$ -путь из вершины u в вершину v существует в том и только том случае, когда $uv \in E$;
- 2) если $uv \in E$, то $d^{\emptyset}(u, v) = \omega_{uv}$; в противном случае $d^{\emptyset}(u, v) = \infty$;
- 3) если $S' = S \cup \{w\}$, то простой S' -путь, ведущий из u в v , представляет собой либо S -путь, ведущий из u в v , либо последовательное соединение двух S -путей, один из которых ведет из u в w , а другой — из w в v ;
- 4) если $S' = S \cup \{w\}$, то $d^{S'}(u, v) = \min(d^S(u, v), d^S(u, w) + d^S(w, v))$;
- 5) вершины u и v соединены путем в том и только том случае, когда между вершинами u и v существует V -путь;
- 6) $d(u, v) = d^V(u, v)$,

Доказательство. Для всех вершин u и подмножеств S верно неравенство $d^S(u, u) \leq 0$, потому что пустой путь (не содержащий ни одного ребра) является S -путем из u в u веса 0. Никакой другой путь не имеет меньшего веса, поскольку в графе G нет циклов отрицательного веса. Значит, $d^S(u, u) = 0$.

Правило (1). Так как $\emptyset$ -путь не имеет промежуточных вершин, $\emptyset$ -путь из u в v состоит только из канала uv .

Правило (2). Следует непосредственно из правила 1).

Правило (3). В простом S' -пути из u в v вершина w либо встречается ровно один раз, либо вообще не встречается. Если не содержит промежуточной вершины w , то это S -путь; в противном случае этот путь представляет собой последовательное соединение двух S -путей, один из которых ведет в вершину w , а другой исходит из вершины w .

Правило (4). Применяя лемму 4.1, можно показать, что в том случае, если в графе есть S' -путь из u в v , существует и *простой* S' -путь длины $d^{S'}(u, v)$ из u в v . Согласно правилу (3) отсюда следует равенство $d^{S'}(u, v) = \min(d^S(u, v), d^S(u, w) + d^S(w, v))$.^{N2} операций (их можно выполнять последовательно или параллельно). Отсюда

Правило (5). Каждый V -путь является путем и наоборот.

Правило (6). Каждый V -путь является путем и наоборот. Поэтому оптимальный V -путь также является оптимальным путем. $\square$

```

begin(* Вначале положить  $S$  равным  $\emptyset$ , а  $D$  считать равной  $\emptyset$ -расстоянию *)
 $S := \emptyset$ ;
forall  $u, v$  do
    if  $u = v$  then  $D[u, v] := 0$ 
    else if  $uv \in E$  then  $D[u, v] := \omega_{uv}$ 
    else  $D[u, v] := \infty$ ;
(* Добавить к  $S$  «центровую» вершину *)
while  $S \neq V$  do
    (* Инвариант цикла:  $\forall u, v : D[u, v] = d^S(u, v)$  *)
    begin pick  $w$  from  $V \setminus S$ ;
    (* Выполнить глобальную обработку опорной вершины  $w$  *)
    forall  $u \in V$  do
        (* Выполнить локальную обработку опорной вершины  $w$  для  $u$  *)
        forall  $v \in V$  do
             $D[u, v] := \min(D[u, v], D[u, w] + D[w, v])$ ;
         $S := S \cup \{w\}$ 
    end (*  $\forall u, v : D[u, v] = d(u, v)$  *)
end

```

Алгоритм 4.4. Алгоритм Флойда—Уоршалла

При помощи утверждения 4.5 и метода динамического программирования нетрудно разработать алгоритм решения задачи построения кратчайших путей для всех пар вершин (см. алгоритм 4.4). В этом алгоритме сначала рассматриваются $\emptyset$ -пути, а затем последовательно строятся S -пути для расширяющихся множеств S , до тех пор пока не будут рассмотрены все пути.

Теорема 4.6. Алгоритм 4.4 вычисляет расстояние между всеми парами вершин за $\Theta(N^3)$ шагов.

Доказательство. В начале работы алгоритма $D[u, v] = 0$, если $u = v$, $D[u, v] = \omega_{uv}$, если $uv \in E$, и $D[u, v] = \infty$ в остальных случаях. При этом $S = \emptyset$. Согласно правилам 1) и 2) утверждения 4.5 мы имеем $\forall u, v : D[u, v] = d^S(u, v)$. Если к множеству S добавляется вершина w , то согласно правилам 3) и 4) того же утверждения оператор присваивания, уточняющий значение $D[u, v]$, обеспечивает сохранение выполнимости предложения $\forall u, v : D[u, v] = d^S(u, v)$, используемого в качестве инварианта цикла. Программа завершает выполнение, когда $S = V$. Принимая во внимание правила 5) и 6) утверждения 4.5, а также инвариант цикла, заключаем, что S -расстояния между всеми вершинами равны расстояниям.

Основной цикл выполняется N раз, и в каждой его итерации выполняется N^2 операций (их можно выполнять последовательно или параллельно). Отсюда получается верхняя оценка сложности алгоритма. $\square$

4.2.2. Алгоритм Туэга построения кратчайших путей

```

var  $S_u$    : set of nodes ;
     $D_u$    : array of weights ;
     $Nb_u$   : array of nodes ;

begin  $S_u := \emptyset$  ;
  forall  $v \in V$  do
    if  $v = u$ 
    then begin  $D_u[v] := 0$  ;  $Nb_u[v] := undef$  end
    else if  $v \in Neigh_u$ 
    then begin  $D_u[v] := \omega_{uv}$  ;  $Nb_u[v] := v$  end
    else begin  $D_u[v] := \infty$  ;  $Nb_u[v] := undef$  end;
  while  $S_u \neq V$  do
    begin pick  $w$  from  $V \setminus S_u$  ;
      (* Здесь все процессы должны выбрать одну и ту же вершину  $w$  *)
      if  $u = w$ 
      then «broadcast the table  $D_w$ »
      else «receive the table  $D_w$ » ;
      forall  $v \in V$  do
        if  $D_u[w] + D_w[v] < D_u[v]$  then
          begin  $D_u[v] := D_u[w] + D_w[v]$  ;
                 $Nb_u[v] := Nb_u[w]$ 
          end;
         $S_u := S_u \cup \{w\}$ 
      end
    end
  end

```

Алгоритм 4.5. Упрощенный вариант алгоритма (для узла u).

Распределенный алгоритм вычисления таблиц маршрутизации был разработан Туэгом в работе [194] на основе алгоритма Флойда—Уоршалла, который был описан в предыдущем параграфе. Мы должны убедиться в том, что алгоритм Флойда—Уоршалла подходит для этой цели, т. е. те допущения, на которых базируется этот алгоритм, верны для распределенных систем. Наиболее важным из этих допущений является предположение о том, что в графе нет циклов отрицательного веса. Распределенные системы и в самом деле обладают этим свойством, так как каждому отдельному каналу связи приписывается положительная стоимость. Можно принять даже более строгое предположение (см. А1 ниже).

В этом параграфе мы будем действовать в рамках следующих допущений.

А1. Каждый цикл в сети имеет положительный вес.

А2. Каждый процесс сети располагает информацией об отличительных признаках всех узлов системы (множества вершин V).

А3. Каждый процесс располагает информацией о том, какие узлы являются его соседями (для каждой вершины u эти сведения содержатся в массиве $Neigh_u$) и какой вес имеют каналы, соединяющие процесс с его соседями.

Нам будет проще разобраться с корректностью алгоритма Туэга (алгоритм 4.6), если вначале мы рассмотрим его упрощенный вариант, так называемый «простой алгоритм» (алгоритм 4.5).

Упрощенный алгоритм. Чтобы получить распределенный алгоритм, переменные и операции алгоритма Флойда—Уоршалла разносятся по разным узлам сети. Переменная $D[u, v]$ отдается процессу u ; для удобства обозначения мы будем записывать u на месте индекса следующим образом $D_u[v]$. Операции, которые присваивают значения переменной $D_u[v]$, должны выполняться в узле u , и, когда значение некоторой переменной, относящейся к узлу w , необходимо для этой операции, это значение должно быть послано процессу u . В алгоритме Флойда—Уоршалла все вершины должны использовать информацию от опорной вершины (она обозначена символом w в теле цикла), которая отправляет эту информацию одновременно всем вершинам посредством «широковещательной» операции. Наконец, в этом алгоритме нужно ввести специальную операцию, для того чтобы не только вычислять длины кратчайших S -путей (для этого используются переменные $D_u[v]$), но также и первый канал в каждом таком пути (для этого мы будем использовать переменные $Nb_u[v]$).

Допущение о том, что всякий цикл в сети имеет положительный вес, можно использовать для того, чтобы показать, что после каждого этапа обработки опорной вершины в таблицах маршрутизации не образуются циклы.

Лемма 4.7. Пусть заданы множество вершин S и вершина w . Предположим также, что

- 1) для всех вершин u верно равенство $D_u[w] = d^S(u, w)$;
- 2) если $d^S(u, w) < \infty$ и $u \neq w$, то значением переменной $Nb_u[w]$ является имя соседа вершины u на кратчайшем S -пути, ведущем к вершине w .

Тогда ориентированный граф $T_w = (V_w, E_w)$, где

$$(u \in V_w \iff D_u[w] < \infty) \quad \text{и} \quad (ux \in E_w \iff (u \neq w \wedge Nb_u[w] = x)),$$

является деревом, в корне которого расположена вершина w .

Доказательство. Прежде всего заметим, что в том случае, если для вершины $u \neq w$ выполняется неравенство $D_u[w] < \infty$, то $Nb_u[w] \neq undef$ и $D_{Nb_u[w]}[w] < \infty$. Значит, для каждой вершины $u \in V_w$, $u \neq w$, существует вершина x , $x \in V_w$, для которой верно равенство $Nb_u[w] = x$.

Для каждой вершины $u \in V_w$, $u \neq w$, в множестве E_w есть одно ребро, и поэтому число вершин в графе T_w превосходит количество ребер на единицу. Осталось показать, что граф T_w не содержит циклов. Коль скоро для ребра $ux \in E_w$ имеет место равенство $d^S(u, w) = \omega_{ux} + d^S(x, w)$, наличие цикла $\langle u_0, u_1, \dots, u_k \rangle$ в графе T_w повлекло бы за собой следующее соотношение:

$$d^S(u_0, w) = \omega_{u_0 u_1} + \omega_{u_1 u_2} + \dots + \omega_{u_{k-1} u_0} + d^S(u_0, w),$$

из которого вытекает то, что

$$0 = \omega_{u_0 u_1} + \omega_{u_1 u_2} + \dots + \omega_{u_{k-1} u_0},$$

а это противоречит допущению о том, что каждый цикл имеет положительный вес. $\square$

Теперь алгоритм Флойда—Уоршалла может быть непосредственно преобразован в алгоритм 4.5. Каждая вершина-процесс устанавливает начальные значения приписанным ей переменным и выполняет N итераций основного цикла. Этот алгоритм еще не дает нам окончательного решения, потому что мы не описали, как провести эффективное широковещательное распространение таблицы маршрутизации опорной вершины. Пока мы можем лишь поверить, что в том случае, когда операция «broadcast the table D_w » выполняется процессом w , а операция «receive the table D_w » выполняется другими процессами, каждый узел получит доступ к таблице D_w .

Нужно обратить особое внимание на операцию «pick w from $V \setminus S$ » и организовать ее выполнение так, чтобы все процессы выбирали опорные вершины в одном и том же порядке. Коль скоро предполагается, что всем процессам заранее известно множество V , мы можем считать, что вершины из этого множества выбираются в каком-либо заранее заданном порядке (например, имена вершин выбираются в алфавитном порядке).

Корректность упрощенного алгоритма устанавливается в следующей теореме.

Теорема 4.8. *Алгоритм 4.5 завершает свое выполнение в каждом узле сети после N итераций основного цикла. Когда алгоритм завершается в узле u , для каждой вершины v выполняется равенство $D_u[v] = d(u, v)$, и в том случае, если в сети есть путь из u в v , то значением переменной $Nb_u[v]$ является наименование первого канала в кратчайшем пути из u в v ; в противном случае $Nb_u[v] = undef$.*

Доказательство. Завершаемость и частичная корректность нашего алгоритма следуют из корректности алгоритма Флойда—Уоршалла (теорема 4.6). Отсюда следует и справедливость утверждения о значении переменной $Nb_u[v]$, поскольку значение $Nb_u[v]$ изменяется всякий раз, когда переменной $D_u[v]$ присваивается новое значение. $\square$

```

var  $S_u$  : set of nodes ;
     $D_u$  : array of weights ;
     $Nb_u$  : array of nodes ;

begin  $S_u := \emptyset$  ;
    forall  $v \in V$  do
        if  $v = u$ 
        then begin  $D_u[v] := 0$  ;  $Nb_u[v] := undef$  end
        else if  $v \in Neigh_u$ 
        then begin  $D_u[v] := \omega_{uv}$  ;  $Nb_u[v] := v$  end
        else begin  $D_u[v] := \infty$  ;  $Nb_u[v] := undef$  end;
    while  $S_u \neq V$  do
        begin pick  $w$  from  $V \setminus S_u$  ;
            (* Построить дерево  $T_w$  *)
            forall  $x \in Neigh_u$  do
                if  $Nb_u[w] = x$  then send  $\langle \mathbf{ys}, w \rangle$  to  $x$  else send  $\langle \mathbf{nys}, w \rangle$  to  $x$  ;
                 $rec_u := 0$  ; (*  $u$  должен получить  $|Neigh_u|$  сообщений *)
                while  $rec_u < |Neigh_u|$  do
                    begin receive  $\langle \mathbf{ys}, w \rangle$  or  $\langle \mathbf{nys}, w \rangle$  message ;  $rec_u := rec_u + 1$  end;
                if  $D_u[w] < \infty$  then (* принять участие в обработке опорной вершины *)
                    begin if  $u \neq w$  then receive  $\langle \mathbf{dtab}, w, D \rangle$  from  $Nb_u[w]$  ;
                        forall  $x \in Neigh_u$  do
                            if  $\langle \mathbf{ys}, w \rangle$  was received from  $x$  then send  $\langle \mathbf{dtab}, w, D \rangle$  to  $x$  ;
                        forall  $v \in V$  do (* local  $w$ -pivot *)
                            if  $D_u[w] + D[v] < D_u[v]$  then
                                begin  $D_u[v] := D_u[w] + D[v]$  ;  $Nb_u[v] := Nb_u[w]$  end
                            end;
                         $S_u := S_u \cup \{w\}$ 
                    end
                end
            end
        end
    end
end

```

Алгоритм 4.6. Алгоритм Туэга (для узла u).

Улучшенный алгоритм. Чтобы эффективно провести широковещательную рассылку в алгоритме 4.5, Туэг обратил внимание на то, что если узел u удовлетворяет условию $D_u[w] = \infty$ в начале этапа обработки опорной вершины, то по окончании обработки этой вершины таблица маршрутизации этого узла остается неизменной. Если $D_u[w] = \infty$, то неравенство $D_u[w] + D_w[v] < D_u[v]$ неверно для каждой вершины v . Следовательно, таблицу маршрутизации вершины w нужно доставить только в те узлы, которые принадлежат дереву T_w (в том виде, в котором оно было сформировано к началу этапа обработки опорной вершины), и операцию широковещательной рассылки можно провести эффективно, отправляя таблицу D_w только по тем каналам, которые входят в состав дерева T_w . Таким образом, узел w отправляет таблицу D_w своим сыновним вершинам в дереве T_w , и каждый узел в дереве T_w , получивший эту таблицу от родительской вершины в дереве T_w , переправляет ее своим сыновним вершинам в дереве T_w .

В начале этапа обработки опорной вершины w каждый узел u , для которого выполнено условие $D_u[w] < \infty$, осведомлен о том, какая вершина является его родителем в дереве T_w , но не располагает сведениями о том, какие вершины являются его сыновними вершинами. Поэтому любой узел v должен отправить сообщение каждому своему соседу u , сообщив процессу u , является ли v сыновней вершиной для u в дереве T_w . Полное решение задачи представлено в описании алгоритма 4.6. Всякий узел может принять участие в распространении таблицы опорной вершины w , как только он получит известия о том, какие из соседей являются его сыновними вершинами в дереве T_w . В алгоритме используются сообщения трех типов.

1. Сообщение $\langle \mathbf{ys}, w \rangle$ ($\mathbf{ys}$ от «**y**oung **s**on») отправляется от узла u к узлу x в начале этапа обработки опорной вершины w , если x является родительской вершиной для u в дереве T_w .

2. Сообщение $\langle \mathbf{nys}, w \rangle$ ($\mathbf{nys}$ от «**n**ot **y**oung **s**on») отправляется от узла u к узлу x в начале этапа обработки опорной вершины w , если x не является родительской вершиной для u в дереве T_w .

3. Сообщение $\langle \mathbf{dtab}, w, D \rangle$ отправляется по ходу обработки опорной вершины w по каждому ребру дерева T_w , чтобы доставить таблицу D_w в каждую вершину, которая должна будет воспользоваться этим значением.

Если известно, что для описания веса (ребра или пути) в совокупности с именем вершины достаточно W битов, то для оценки сложности представленного алгоритма можно обратиться к следующей теореме.

Теорема 4.9. Для каждой пары вершин u и v алгоритм 4.6 вычисляет расстояние между u и v , а в том случае, когда это расстояние конечно, также определяет первый канал в кратчайшем пути.

По ходу работы алгоритма по каждому каналу проходит $O(N)$ сообщений и $O(N^2 W)$ битов информации. Таким образом, суммарно по ходу работы алгоритма передается $O(N \cdot |E|)$ сообщений и $O(N^3 \cdot W)$ битов информации. Кроме того, в каждом узле используется память, объем которой составляет $O(N \cdot W)$ битов.

Доказательство. Алгоритм 4.6 построен на основе алгоритма 4.5, и это гарантирует его корректность.

На каждом этапе обработки опорной вершины w по каждому каналу связи проходят два сообщения вида $\langle \mathbf{ys}, w \rangle$ или $\langle \mathbf{nys}, w \rangle$ (по одному сообщению в каждом направлении) и не более одного сообщения вида $\langle \mathbf{dtab}, w, D \rangle$, и, значит, за время работы алгоритма по каждому каналу проходит не более $3N$ сообщений. Сообщения вида $\langle \mathbf{ys}, w \rangle$ или $\langle \mathbf{nys}, w \rangle$ содержат $O(W)$ битов, а сообщение вида $\langle \mathbf{dtab}, w, D \rangle$ содержит $O(NW)$ битов, и отсюда следует верхняя оценка битовой сложности обмена информацией по каждому каналу связи. За время работы алгоритма передается не более N^2 сообщений вида $\langle \mathbf{dtab}, w, D \rangle$ и $2N \cdot |E|$ сообщений вида $\langle \mathbf{ys}, w \rangle$ и $\langle \mathbf{nys}, w \rangle$; таким образом, всего по сети передается $O(N^2 \cdot NW + 2N \cdot |E| \cdot W) = O(N^3 W)$ битов информации. Для хранения таблиц D_u и Nb_u каждому процессу u требуется $O(NW)$ битов памяти. $\square$

На протяжении одного этапа обработки опорной вершины w каждому узлу разрешается передавать и получать только те сообщения, которые имеют отношение к текущему этапу, т. е. такие сообщения, в которых в качестве параметра содержится имя вершины w . Если в каналах соблюдается очередность сообщений, то первые сообщения, которые получит каждый узел в начале очередного этапа обработки опорной вершины, — это сообщения вида $\langle \mathbf{ys}, w \rangle$ и $\langle \mathbf{nys}, w \rangle$, передаваемые по каждому каналу; затем от узла $Nb_u[w]$ будет получено сообщение вида $\langle \mathbf{dtab}, w, D \rangle$ (если узел-адресат содержится в множестве вершин V_w). Аккуратное программирование позволяет избавиться от параметра w во всех сообщениях, если в каналах связи соблюдается очередность сообщений. Если же очередность следования сообщений нарушается, может случиться так, что узел получит сообщение с параметром w' , в то время как он ожидает поступления сообщения с параметром w , причем опорная вершина w' следует по порядку за опорной вершиной w . В этом случае параметр w позволяет разобраться с тем, какому этапу обработки опорных вершин соответствует полученное сообщение; далее, воспользовавшись локальной буферизацией сообщений (в рамках канала или процесса), можно устанавливать правильный порядок обработки сообщений.

Туэг провел дальнейшую оптимизацию предложенного алгоритма, опираясь на следующее утверждение. (Вершина u_2 считается *потомком* вершины u_1 , если u_2 принадлежит поддереву с корнем u_1 .)

Лемма 4.10. Пусть $u_1 \neq w$, и пусть вершина u_2 является потомком вершины u_1 в дереве T_w в начале этапа обработки опорной вершины w . Если на этапе обработки опорной вершины w изменяется расстояние между вершиной u_2 и некоторой вершиной v , то на этом этапе также изменяется расстояние между вершиной u_1 и вершиной v .

Доказательство. Коль скоро u_2 является потомком вершины вершины u_1 в дереве T_w ,

$$d^S(u_2, w) = d^S(u_2, u_1) + d^S(u_1, w). \quad (1)$$

Так как $u_1 \in S$, мы имеем неравенство

$$d^S(u_2, v) \leq d^S(u_2, u_1) + d^S(u_1, v). \quad (2)$$

На этом этапе значение $D_{u_2}[v]$ изменяется в узле u_2 в том и только том случае, когда

$$d^S(u_2, w) + d^S(w, v) < d^S(u_2, v). \quad (3)$$

Воспользовавшись неравенствами (2) и (3), подставив правую часть равенства (1) вместо $d^S(u_2, w)$ и вычтя из обеих частей образовавшегося неравенства $d^S(u_2, u_1)$, мы получаем соотношение

$$d^S(u_1, w) + d^S(w, v) < d^S(u_1, v), \quad (4)$$

из которого следует, что на этом этапе в узле u_1 также изменяется значение переменной $D_{u_1}[v]$. $\square$

Согласно доказанной лемме, алгоритм 4.6 можно улучшить следующим образом. После того как узел u получает сообщение $\langle \mathbf{dtab}, w, D \rangle$ и получает доступ к таблице D_w , в нем вначале выполняется локальная обработка опорной вершины w , а затем таблица D_w отправляется сыновним вершинам узла u в дереве T_w . При этом достаточно передать лишь те элементы таблицы $D[v]$, для которых в узле u в ходе локальной обработки опорной вершины w изменились значения $D_u[v]$. Предложенная модификация позволяет получать ациклические таблицы маршрутизации не только в конце каждого этапа обработки опорных вершин (о чем свидетельствует лемма 4.7), но также на всем протяжении этапа обработки опорных вершин.

4.2.3. Другие алгоритмы

Приведенное описание алгоритма Туэга является примером того, как распределенный алгоритм решения некоторой задачи получается непосредственно из последовательного алгоритма решения той же задачи. Для этого переменные последовательного алгоритма разнятся по разным процессам; присваивание переменной x (в последовательном алгоритме) выполняет тот процесс, которому назначена переменная x . Всякий раз, когда в операторе присваивания используются переменные, назначенные другим процессам, необходимо совершить обмен сообщениями между процессами, чтобы доставить значения этих переменных и провести синхронизацию действий процессов. Для того чтобы свести обмен сообщениями к минимуму, можно использовать те или иные особенности исходного последовательного алгоритма.

Алгоритм Туэга достаточно прост для понимания, имеет небольшую сложность и строит оптимальные маршруты; главным недостатком алгоритма является плохая устойчивость («робастность»). Как только изменяется топология сети, все вычисления нужно проводить заново. Кроме того, у алгоритма есть две особенности, из-за которых он становится менее привлекательным с точки зрения проектирования распределенных алгоритмов.

Во-первых, как уже было упомянуто, согласованный выбор очередной опорной вершины (w) всеми узлами сети предполагает, что множество участвующих в алгоритме процессов заранее известно. Но поскольку в общем случае множество участников заранее неизвестно, перед тем как запустить алгоритм Туэга, нужно применить вспомогательный распределенный алгоритм (например, алгоритм Финна (алгоритм 6.8)), чтобы вычислить это множество.

Во-вторых, в основу алгоритма Туэга положено многократное применение *неравенства треугольника* $d(u, v) \leq d(u, w) + d(w, v)$. Для вычисления правой части этого неравенства (в узле u) требуется информация о $d(w, v)$, а эта информация в общем случае оказывается *удаленной*, т.е. ею не обладает ни процесс u , ни его соседи. Зависимость от удаленных данных вынуждает нас организовать доставку этой информации удаленным вершинам; в алгоритме Туэга этим занимается блок ширококвещательного распространения сообщений.

В качестве альтернативы мы можем рассмотреть следующее уравнение, определяющее значения $d(u, v)$, которое можно использовать в алгоритме построения

кратчайших путей:

$$d(u, v) = \begin{cases} 0, & \text{если } u = v; \\ \min_{w \in \text{Neigh}_u} \omega_{uw} + d(w, v) & \text{иначе.} \end{cases} \quad (4.1)$$

Два свойства этого уравнения позволяют конструировать алгоритмы, которые будут отличаться от алгоритма Туэга.

1. *Локальность данных.* Для вычисления правой части уравнения (4.1) в узле u требуется только локально доступная информация (а именно ω_{uw}) или информация, которая имеется у соседей (а именно $d(w, v)$). Необходимость в обмене данными между удаленными вершинами отпадает.

2. *Независимость от вершин-адресатов.* Для вычисления расстояния между вершинами u и v требуется знать только расстояние $d(w, v)$ между v и всеми соседями w вершины u . Поэтому вычисление и анализ всех расстояний до заданной вершины-адресата v_0 можно проводить независимо от вычислений расстояний до других вершин сети.

В оставшейся части этого параграфа мы обсудим два алгоритма, в основу которых положено уравнение (4.1), а именно алгоритм Мерлина—Сигалла и алгоритм Чанди—Мисры. Несмотря на те преимущества, которые открываются в связи с локальностью данных, коммуникационная сложность этих алгоритмов ничуть не лучше, чем сложность алгоритма Туэга. Это объясняется свойством независимости от вершины-адресата, которым обладает уравнение (4.1); на самом деле использование результатов, которые были получены для других вершин-адресатов (как это осуществляется в алгоритме Туэга), представляется более выгодным методом, нежели введение локальности данных.

Но если уменьшения коммуникационной сложности не происходит, то почему же столь важна локальность данных? Зависимость вычисления от удаленных данных приводит к тому, что эти данные приходится ширококвещательно распространять много раз, если они становятся другими вследствие изменений топологической структуры сети из-за отказов и возобновления работы каналов и узлов. А проведение ширококвещательной рассылки сообщений (если иметь при этом в виду, что в ходе самой рассылки могут происходить и другие топологические изменения) — это очень непростая задача, решение которой требует больших затрат (см., например, [95]). Поэтому алгоритмы, в основу которых положено уравнение (4.1), можно проще адаптировать для учета топологических изменений. Примером тому может служить алгоритм, который мы подробно рассмотрим в § 4.6.3.

Алгоритм Мерлина—Сигалла. Алгоритм, предложенный Мерлином и Сигаллом в работе [147], вычисляет таблицы маршрутизации по отдельности для каждой вершины-адресата; вычисления таблиц для разных вершин-адресатов не оказывают совершенно никакого влияния друг на друга. Для каждой вершины-адресата v алгоритм строит дерево T_v с корнем в вершине v и проводит итеративное преобразование этого дерева, с тем чтобы в конце концов получить оптимальное входное дерево для вершины-адресата v .

В каждом узле u хранится оценка расстояния $D_u[v]$ до вершины v , а также имя того соседа, которому нужно передать пакет, адресованный узлу v , и этот сосед является родительской вершиной для узла u в дереве T_v . При каждой итерации всякий узел u отправляет известную ему оценку расстояния $D_u[v]$ всем своим соседям, *кроме* вершины $Nb_u[v]$ (для этого используется сообщение $\langle \text{mydist}, v, D_u[v] \rangle$). Как только узел u получает от своего соседа w сообщение $\langle \text{mydist}, v, d \rangle$ и обнаруживает, что $d + \omega_{uw} < D_u[v]$, в узле u значение переменной $Nb_u[v]$ полагается равным w , а значение переменной $D_u[v]$ полагается равным $d + \omega_{uw}$. Управляющим параметром этого преобразования служит вершина v , и для его проведения нужно провести обмен двумя сообщениями, состоящими из W битов, по каждому каналу связи.

В работе [147] установлено, что после проведения i итераций все кратчайшие пути, состоящие не более чем из i звеньев, будут правильно вычислены, и поэтому для вычисления всех кратчайших путей нужно совершить не более N итераций. Кратчайшие пути ко всем вершинам-адресатам вычисляются за счет независимого применения предложенного алгоритма ко всем вершинам-адресатам.

Теорема 4.11. *Алгоритм Мерлина—Сигалла вычисляет таблицы маршрутизации по кратчайшим путям, совершая при этом обмен $O(N^2)$ сообщениями и передавая $O(N^2 \cdot W)$ битов информации по каждому каналу связи, имея, таким образом, коммуникационную сложность $O(N^2 \cdot |E|)$ и битовую сложность $O(N^2 \cdot |E|W)$.*

Этот алгоритм можно приспособить к изменениям топологии сети и весовых характеристик каналов. Важная особенность этого алгоритма состоит в том, что по ходу проведения итераций таблицы маршрутизации остаются ациклическими.

Алгоритм Чанди—Мисры. Алгоритм, предложенный Чанди и Мисрой в работе [44], вычисляет все кратчайшие пути к заданной вершине-адресату на основе принципа *диффузных вычислений*. Суть его такова: распределенное вычисление начинается в одной-единственной вершине, и другие узлы сети поочередно присоединяются к вычислению только после получения сообщения.

Чтобы вычислить расстояние от каждой вершины до заданного узла v_0 (а также предпочтительный исходящий канал связи), каждый узел u устанавливает начальное значение $D_u[v_0] = \infty$ и ожидает получения сообщений. Узел v_0 отправляет сообщение $\langle \text{mydist}, v_0, 0 \rangle$ всем своим соседям. Как только вершина u получает сообщение $\langle \text{mydist}, v_0, d \rangle$ от соседа w и убеждается в том, что выполняется неравенство $d + \omega_{uw} < D_u[v_0]$, значением переменной $D_u[v_0]$ становится сумма $d + \omega_{uw}$, и всем соседям вершины u отправляется сообщение $\langle \text{mydist}, v_0, D_u[v_0] \rangle$ (см. алгоритм 4.7).

Легко показать, что значение $D_u[v_0]$ всегда является верхней оценкой величины $d(u, v_0)$, т. е. неравенство $d(u, v_0) \leq D_u[v_0]$ следует из инварианта алгоритма; см. упражнение 4.3. Чтобы убедиться в том, что алгоритм правильно вычисляет расстояния, необходимо показать, что рано или поздно будет достигнута такая конфигурация, для которой в каждом узле будет выполняться неравенство $D_u[v_0] \leq d(u, v_0)$. Мы приведем доказательство этого утверждения, в котором

```

var  $D_u[v_0]$  : вес      init  $\infty$  ;
     $Nb_u[v_0]$  : вершина init undef ;

```

Только для вершины v_0 :

```

begin  $D_{v_0}[v_0] := 0$  ;
    forall  $w \in Neigh_{v_0}$  do send  $\langle \text{mydist}, v_0, 0 \rangle$  to  $w$ 
end

```

Обработка сообщения $\langle \text{mydist}, v_0, d \rangle$, полученного вершиной u от соседа w :

```

{  $\langle \text{mydist}, v_0, d \rangle \in M_{wu}$  }
begin receive  $\langle \text{mydist}, v_0, d \rangle$  from  $w$  ;
    if  $d + \omega_{uw} < D_u[v_0]$  then
        begin  $D_u[v_0] := d + \omega_{uw}$  ;  $Nb_u[v_0] := w$  ;
            forall  $x \in Neigh_u$  do send  $\langle \text{mydist}, v_0, D_u[v_0] \rangle$  to  $x$ 
        end
    end
end

```

Алгоритм 4.7. Алгоритм Чанди—Мисры (для узла u).

подразумевается соблюдение допущения слабой справедливости, гласящего, что в любом вычислении каждое отправленное сообщение рано или поздно будет получено. Существует и такое доказательство, которое никак не опирается на указанное допущение, но оно гораздо сложнее.

Теорема 4.12. *В любом вычислении алгоритма 4.7 достигается такая конфигурация, что в каждом узле u выполняется неравенство $D_u[v_0] \leq d(u, v_0)$.*

Доказательство. Рассмотрим некоторое оптимальное входное дерево T для вершины v_0 и занумеруем²⁾ все вершины, отличные от v_0 , натуральными числами от 1 до $N - 1$ так, чтобы всякий раз, когда v_i является родительской вершиной для v_j , выполнялось неравенство $i < j$. Рассмотрим произвольное вычисление S и покажем, воспользовавшись индукцией по j , что для каждого номера j , $j \leq N - 1$, достигается такая конфигурация, в которой для всех номеров i , $i \leq j$, выполняется неравенство $D_{v_i}[v_0] \leq d(v_i, v_0)$. Заметим, что значение переменной $D_{v_i}[v_0]$ никогда не увеличивается по ходу работы алгоритма, и поэтому, как только соотношение $D_{v_i}[v_0] \leq d(v_i, v_0)$ оказывается верным в какой-либо конфигурации, оно будет также оставаться верным и во всех последующих конфигурациях.

²⁾Здесь в доказательстве допущена принципиальная ошибка. Для заданной вершины v_0 в сети может существовать несколько различных оптимальных входных деревьев, и разные выполнения алгоритма могут строить разные деревья. Правильнее было бы рассмотреть произвольное выполнение S рассматриваемого алгоритма и показать, что S обязательно завершается. После этого вершины сети нумеруются так, чтобы всякий раз, когда последнее изменение значения переменной $D_{v_i}[v_0]$ в вычислении S предшествует последнему изменению переменной $D_{v_j}[v_0]$ в том же вычислении S , выполнялось неравенство $i < j$. — Прим. перев.

Базис ($j = 0$). После выполнения блока инициализации v_0 имеем $d(v_0, v_0) = 0$, и $D_{v_0}[v_0] = 0$, поэтому неравенство $D_{v_0}[v_0] \leq d(v_0, v_0)$ соблюдается и после выполнения всего процесса.

Индуктивный переход $j \rightarrow j + 1$. Предположим, что была достигнута такая конфигурация, в которой для всех узлов с номерами $i \leq j$ выполняется неравенство $D_{v_i}[v_0] \leq d(v_i, v_0)$. Рассмотрим вершину v_{j+1} . Из нее в вершину v_0 существует кратчайший путь $v_{j+1}, v_i, \dots, v_0$ длины $d(v_{j+1}, v_0)$, в котором v_i является родительской вершиной для v_{j+1} в дереве T , и поэтому $i \leq j$. Следовательно, по индуктивному предположению может быть достигнута такая конфигурация, в которой $D_{v_i}[v_0] \leq d(v_i, v_0)$. Всякий раз, когда значение $D_{v_i}[v_0]$ уменьшается, процесс v_i отправляет сообщение $\langle \mathbf{mydist}, v_0, D_{v_i}[v_0] \rangle$ своим соседям, и, следовательно, сообщение $\langle \mathbf{mydist}, v_0, d \rangle$, в котором $d \leq d(v_i, v_0)$, хотя бы один раз отправляется процессу v_{j+1} .

По предположению это сообщение будет получено процессом v_{j+1} в некоторой конфигурации C . Согласно описанию алгоритма после получения этого сообщения будет выполняться неравенство $D_{v_{j+1}}[v_0] \leq d + \omega_{v_{j+1}v_i}$; как следует из выбора номера i , это влечет за собой неравенство $d + \omega_{v_{j+1}v_i} \leq d(v_{j+1}, v_0)$. $\square$

Полное описание алгоритма также включает в себя процедуру, посредством которой узлы могут обнаружить, что вычисление завершилось (обратите внимание на замечание в начале §4.3.3, относящееся к алгоритму Netchange). Эта процедура обнаружения завершения вычисления является разновидностью алгоритма Дейкстры—Шолтена, который обсуждается в §8.2.1.

Рассмотренный здесь алгоритм отличается от алгоритма Мерлина—Сигалла двумя особенностями. Во-первых, не ожидается того, что узел, отправивший в вершину u сообщение типа **mydist**, становится родительской вершиной для u . Эта характерная черта алгоритма Мерлина—Сигалла позволяет таблицам маршрутизации оставаться ациклическими по ходу вычисления и даже при изменении топологии сети. Во-вторых, обмен сообщениями типа **mydist** на этапах итерации не координируется, а выполняется абсолютно произвольно, и это оказывает нежелательное влияние на сложность алгоритма. Алгоритму может потребоваться провести экспоненциально много обменов сообщениями, чтобы вычислить пути к одной-единственной вершине-адресату v_0 . Если стоимость всех каналов одинакова (т. е. рассматривается задача оптимальной по числу звеньев маршрутизации), все кратчайшие пути к вершине v_0 вычисляются с использованием $O(N \cdot |E|)$ сообщений (каждое из которых состоит из $O(W)$ битов), и это приводит нас к следующему результату.

Теорема 4.13. *Алгоритм Чанди—Мисры вычисляет таблицы маршрутизации с наименьшим числом звеньев, совершая при этом обмен $O(N^2)$ сообщениями и передавая $O(N^2 \cdot W)$ битов информации по каждому каналу связи, и имея, таким образом, коммуникационную сложность $O(N^2 \cdot |E|)$ и битовую сложность $O(N^2 \cdot |E|W)$.*

Преимуществом алгоритма Чанди—Мисры по сравнению с алгоритмом Мерлина—Сигалла является его простота, меньшая сложность по объему памяти и меньшая сложность по времени исполнения.

4.3. Алгоритм Netchange

Алгоритм Netchange был предложен Таджибнаписом в работе [180]; он вычисляет таблицы маршрутизации, которые являются оптимальными по числу звеньев. Этот алгоритм имеет много общего с алгоритмом Чанди—Мисры, но при этом в нем поддерживается дополнительная информация, которая позволяет в случае возникновения неисправности в канале и восстановления работоспособности канала уточнять таблицы путем их *частичного* перевычисления. При изложении материала в этом параграфе мы будем придерживаться работы Лампорта [125]. При построении алгоритма мы будем опираться на следующие допущения.

N1. Каждый узел осведомлен о размере всей сети (N).

N2. В каналах поддерживается очередность сообщений.

N3. Узлы получают уведомления о возникновении неисправностей и восстановлении работоспособности примыкающих к ним каналов.

N4. Стоимость пути равна количеству каналов в этом пути.

Алгоритм может справиться с возникновением неисправностей и восстановлением работоспособности каналов, а также с добавлением новых каналов связи, но при этом предполагается, что всякий узел немедленно узнает о повреждении или восстановлении примыкающих к нему каналов связи. Повреждение и восстановление узлов сети в расчет не принимается; считается, что повреждение узла сети воспринимается его соседями как повреждение каналов, связывающих их с этим узлом. В каждом узле u алгоритм вычисляет и поддерживает таблицу $Nb_u[v]$, в которой для каждой вершины-адресата v указывается тот сосед узла u , которому должен быть отправлен пакет, адресованный вершине v . Мы не можем требовать, чтобы вычисление этих таблиц всегда завершалось за конечное число шагов, поскольку постоянные обрывы и восстановления каналов связи могут потребовать столь же частых перевычислений. К алгоритму предъявляются следующие требования.

R1. Если топология сети после конечного числа изменений остается далее неизменной, то алгоритм завершает работу после конечного числа шагов.

R2. Когда алгоритм завершает работу, таблицы $Nb_u[v]$ удовлетворяют следующим условиям:

а) если $v = u$, то $Nb_u[v] = local$;

б) если существует путь из вершины u в вершину $v \neq u$, то $Nb_u[v] = w$, где w — это первая вершина-сосед узла u , который следует за u в кратчайшем пути из u в v ;

в) если пути из вершины u в вершину v нет, то $Nb_u[v] = undef$.

4.3.1. Описание алгоритма

Алгоритм Таджибнаписа Netchange состоит из двух частей: алгоритм 4.8 и алгоритм 4.9. Сначала мы объясним содержательный смысл всех операций алгоритма, а затем формально докажем его корректность. Чтобы добиться ясности изложения, мы упростили моделирование топологических изменений по сравнению с тем, как это описано в работе [125]: предполагается, что уведомление

о каждом таком изменении обрабатывается одновременно в обоих узлах, которые оказываются затронутыми этим изменением. В § 4.3.3 мы укажем, как эти уведомления можно обрабатывать асинхронно.

```

var  $Neigh_u$       : множество вершин;      (* Соседи узла  $u$  *)
     $D_u$            : массив размера 0..  $N$ ;  (*  $D_u[v]$  — это оценка величины  $d(u, v)$  *)
     $Nb_u$           : массив вершин;        (*  $Nb_u[v]$  — это предпочтительные соседи вершины  $u$ 
                                           при отправлении сообщений вершине-адресату  $v$  *)
     $ndis_u$         : массив размера 0..  $N$ ; (*  $ndis_u[w, v]$  — это оценка величины  $d(w, v)$  *)

```

Инициализация:

```

begin forall  $w \in Neigh_u, v \in V$  do  $ndis_u[w, v] := N$ ;
  forall  $v \in V$  do
    begin  $D_u[v] := N$ ;  $Nb_u[v] := udef$  end;
     $D_u[u] := 0$ ;  $Nb_u[u] := local$ ;
    forall  $w \in Neigh_u$  do send  $\langle mydist, u, 0 \rangle$  to  $w$ 
  end

```

Процедура *Перевычисление* (v):

```

begin if  $v = u$ 
  then begin  $D_u[v] := 0$ ;  $Nb_u[v] := local$  end
  else begin (* Оценить расстояние до вершины  $v$  *)
     $d := 1 + \min\{ndis_u[w, v] : w \in Neigh_u\}$ ;
    if  $d < N$  then
      begin  $D_u[v] := d$ ;
         $Nb_u[v] := w$  с  $1 + ndis_u[w, v] = d$ 
      end
    else begin  $D_u[v] := N$ ;  $Nb_u[v] := udef$  end
  end;
  if переменная  $D_u[v]$  изменила значение then
    forall  $x \in Neigh_u$  do send  $\langle mydist, v, D_u[v] \rangle$  to  $x$ 
end

```

Алгоритм 4.8. Алгоритм Netchange (Часть 1, для узла u).

Выбор того соседа, которому отправляются пакеты, адресованные узлу v , основывается на оценках расстояния от каждой вершины до узла v . Предпочтение всегда отдается тому соседу, у которого эта оценка является наименьшей. В памяти узла u хранится оценка $D_u[v]$ расстояния $d(u, v)$ и оценки $ndis_u[w, v]$ расстояний $d(w, v)$ для каждого соседа w вершины u . Оценка $D_u[v]$ вычисляется на основании оценок $ndis_u[w, v]$, а оценки $ndis_u[w, v]$ получаются путем взаимосвязи с соседями.

Вычисление оценок $D_u[v]$ проводится следующим образом. Если $u = v$, то $d(u, v) = 0$, и в этом случае значение $D_u[v]$ полагается равным 0. Если $u \neq v$, то кратчайший путь из вершины u в вершину v (если таковой существует) состоит из канала, соединяющего узел u с одним из его соседей, и кратчайшего пути,

Обработка сообщения $\langle mydist, v, d \rangle$, полученного от соседа w :

```

{ Сообщение  $\langle mydist, v, d \rangle$  в начале очереди  $Q_{wv}$  }
begin receive  $\langle mydist, v, d \rangle$  from  $w$ ;
     $ndis_u[w, v] := d$ ; Перевычисление ( $v$ )
end

```

В случае выхода из строя канала uw :

```

begin receive  $\langle fail, w \rangle$ ;  $Neigh_u := Neigh_u \setminus \{w\}$ ;
    forall  $v \in V$  do Перевычисление ( $v$ )
end

```

В случае восстановления канала uw :

```

begin receive  $\langle repair, w \rangle$ ;  $Neigh_u := Neigh_u \cup \{w\}$ ;
    forall  $v \in V$  do
      begin  $ndis_u[w, v] := N$ ;
        send  $\langle mydist, v, D_u[v] \rangle$  to  $w$ 
      end
    end
end

```

Алгоритм 4.9. Алгоритм Netchange (Часть 2, для узла u).

который соединяет этого соседа с вершиной v . Следовательно,

$$d(u, v) = 1 + \min_{w \in Neigh_u} d(w, v).$$

Руководствуясь этим уравнением, мы получаем оценку величины $d(u, v)$ в узле $u \neq v$ за счет применения указанной формулы к *оценкам* величин $d(w, v)$, содержащимся в таблицах $ndis_u[w, v]$. Коль скоро в сети имеется N узлов, длина пути с наименьшим числом звеньев не превосходит $N - 1$. Если же вычисленная оценка длины не меньше чем N , то можно предполагать, что такого пути вообще не существует; именно для этой цели в таблице используется значение N .

В этом алгоритме требуется, чтобы в узле хранились оценки расстояния от соседних вершин до вершины v . Эти оценки поступают от соседних вершин, так как они передаются в сообщениях типа **mydist** следующим образом. Если в узле u была вычислена оценка d расстояния от него до вершины v ($D_u[v] = d$), то эта информация передается всем соседним вершинам в сообщении $\langle mydist, v, d \rangle$. После получения сообщения $\langle mydist, v, d \rangle$ от соседа w , в узле u переменной $ndis_u[w, v]$ присваивается значение d . В результате изменения значения переменной $ndis_u[w, v]$ оценка $d(u, v)$ в узле u может измениться, и поэтому эта оценка перевычисляется всякий раз, когда происходят изменения в таблице $ndis_u$. Если оценка расстояния и в самом деле изменяется, например становится равной d' , то об этом, конечно же, оповещаются все соседи при помощи сообщений $\langle mydist, v, d' \rangle$.

В ответ на выход из строя или восстановление работоспособности канала связи алгоритм вносит изменения в локальные таблицы и отправляет сообщение типа **mydist**, если при этом изменяются оценки расстояний. Предполагается,

что уведомления о поломках и починках каналов связи (допущение N3) поступают в виде сообщений типа **fail** и **repairl**. Канал связи между узлами u_1 и u_2 моделируется парой очередей $Q_{u_1u_2}$ для сообщений от u_1 к u_2 и $Q_{u_2u_1}$ для сообщений от u_2 к u_1 . Когда канал выходит из строя, эти очереди удаляются из конфигурации (что немедленно влечет за собой потерю всех сообщений в обеих очередях), а узлы по обе стороны канала связи получают сообщение типа **fail**. Если канал связи между узлами u_1 и u_2 обрывается, то процесс u_1 получает сообщение $\langle \text{fail}, u_2 \rangle$, а процесс u_2 — сообщение $\langle \text{fail}, u_1 \rangle$. Когда связь в канале восстанавливается (или в сети появляется новый канал связи), в конфигурации возникают две новые пустые очереди, а узлы по обе стороны этого канала связи получают сообщение типа **repair**. Если между узлами u_1 и u_2 образуется канал связи, то процесс u_1 получает сообщение $\langle \text{repair}, u_2 \rangle$, а процесс u_2 — сообщение $\langle \text{repair}, u_1 \rangle$.

Алгоритм предусматривает следующую реакцию на сообщения о разрыве или восстановлении канала связи. Когда выходит из строя канал связи между узлами u и w , вершина w удаляется из списка $Neigh_u$, и наоборот. Проводится переычисление оценок расстояний до каждой вершины, и в случае изменения оценки, об этом узнают все оставшиеся соседи. Это происходит, если неисправный канал ранее был одним из звеньев наилучшего маршрута и не осталось ни одного соседа w' , для которого верно равенство $ndis_u[w', v] = ndis_u[w, v]$. Когда канал связи восстанавливается (или добавляется новый канал), вершина w добавляется к списку вершин $Neigh_u$, но в памяти узла u еще нет никаких оценок расстояния $d(w, v)$ (и наоборот). Новому соседу w немедленно сообщаются оценки $D_u[v]$ расстояний от узла u до всех возможных вершин-адресатов v (путем отправления сообщений $\langle \text{mydist}, v, D_u[v] \rangle$). До тех пор пока узел u не получит такого сообщения от узла w , величина N будет использоваться в нем для оценки расстояния $d(w, v)$, т. е. значение переменной $ndis_u[w, v]$ полагается равным N .

$$P(u, w, v) \equiv \begin{aligned} & up(u, w) \iff w \in Neigh_u \\ & \wedge up(u, w) \wedge Q_{wu} \text{ содержит сообщение } \langle \text{mydist}, v, d \rangle \\ & \implies \text{последнее сообщение в очереди} \\ & \quad \text{удовлетворяет равенству } d = D_w[v] \end{aligned} \quad (1)$$

$$\begin{aligned} & \implies \text{последнее сообщение в очереди} \\ & \quad \text{удовлетворяет равенству } d = D_w[v] \end{aligned} \quad (2)$$

$$\begin{aligned} & \wedge up(u, w) \wedge Q_{wu} \text{ не содержит сообщений } \langle \text{mydist}, v, d \rangle \\ & \implies ndis_u[w, v] = D_w[v] \end{aligned} \quad (3)$$

$$L(u, v) \equiv \begin{aligned} & u = v \implies (D_u[v] = 0 \wedge Nb_u[v] = local) \end{aligned} \quad (4)$$

$$\begin{aligned} & \wedge (u \neq v \wedge \exists w \in Neigh_u : ndis_u[w, v] < N - 1) \\ & \implies (D_u[v] = 1 + \min_{w \in Neigh_u} ndis_u[w, v] = 1 + ndis_u[Nb_u[v], v]) \end{aligned} \quad (5)$$

$$\begin{aligned} & \wedge (u \neq v \wedge \forall w \in Neigh_u : ndis_u[w, v] \geq N - 1) \\ & \implies (D_u[v] = N \wedge Nb_u[v] = undef) \end{aligned} \quad (6)$$

Рис. 4.10. Инварианты $P(u, w, v)$ и $L(u, v)$

Инварианты алгоритма Netchange. Мы установим, что утверждения, приведенные на рис. 4.10, являются инвариантами. Формула $P(u, w, v)$ выражает утверждение о том, что если процесс u завершил обработку сообщений типа **mydist**, полученных от процесса w , то оценка расстояния $d(w, v)$ в узле u совпадает с оценкой расстояния $d(w, v)$ в узле w . Мы будем считать, что предикат $up(u, w)$ принимает значение true тогда и только тогда, когда между узлами u и w есть (двунаправленный) исправный канал связи. Формула $L(u, v)$ гласит, что оценки расстояний $d(u, v)$ и содержимое списка $Nb_u[v]$ в узле u всегда согласовано с теми данными, которые хранятся в локальной памяти узла u .

Вычисление алгоритма завершается, когда в каналах связи не остается ни одного сообщения, находящегося на этапе пересылки. Конфигурации такого вида не являются заключительными конфигурациями для всей системы в целом, потому что вычисление системы может быть продолжено, после того как какой-нибудь канал выйдет из строя или восстановит свою работоспособность (что потребует реакции со стороны алгоритма). Конфигурацию, не имеющую в каналах связи ни одного сообщения, мы будем называть *стабильной*; предикат **stable** определяется следующим соотношением:

$$\text{stable} \equiv \forall u, w : up(u, w) \implies Q_{wu} \text{ не содержит сообщений типа mydist.}$$

Необходимо принять во внимание, что первоначально каждый список $Neigh_u$ правильно отражает наличие и работоспособность каналов связи между узлами, т. е. первоначально соотношение (1) считается верным. Чтобы доказать инвариантность приведенных утверждений, мы должны рассмотреть три типа переходов.

1. Прием сообщения типа **mydist**. Все выполнение соответствующего фрагмента кода проводится автоматически и укладывается в один переход. В этом переходе происходит прием одного сообщения и (возможно) отправление нескольких сообщений.

2. Обрыв канала и обработка сообщения типа **fail** в узлах по обе стороны канала.

3. Восстановление канала и обработка сообщения типа **repair** в обоих узлах, соединенных этим каналом.

Лемма 4.14. Для любой тройки вершин u_0, w_0 и v_0 формула $P(u_0, w_0, v_0)$ является инвариантом.

Доказательство. В самом начале после выполнения процедуры инициализации в каждом узле соотношение (1) выполняется согласно сделанному допущению. Если первоначально выполняется условие $\neg up(u_0, w_0)$, то соотношения (2) и (3), очевидно, выполняются. Если же вначале выполняется условие $up(u_0, w_0)$, то имеет место равенство $ndis_{u_0}[w_0, v_0] = N$. Если $w_0 = v_0$, то $D_{w_0}[w_0] = 0$, но при этом в очереди $Q_{w_0u_0}$ содержится сообщение $\langle \text{mydist}, v_0, 0 \rangle$, и поэтому верны соотношения (2) и (3). Если $w_0 \neq v_0$, то $D_{w_0}[v_0] = N$ и в очереди канала нет сообщений, поэтому соотношения (2) и (3) также будут верны. Теперь рассмотрим поочередно три типа переходов, которые были упомянуты выше.

Тип 1. Предположим, что узел u получает сообщение $\langle \mathbf{mydist}, v, d \rangle$ от узла w . Это не влечет за собой никаких изменений топологии сети, и поэтому списки вершин-соседей $Neigh$ не изменяются, и соотношение (1) остается верным. Если $v \neq v_0$, то с получением этого сообщения в формуле $P(u_0, w_0, v_0)$ ничего не изменится.

Если $v = v_0$, $u = u_0$ и $w = w_0$, то значение $ndis_{u_0}[w_0, v_0]$ может измениться. Однако если в канале связи есть еще одно сообщение типа **mydist**, то значения, содержащиеся в этом сообщении, удовлетворяют соотношению (2), и поэтому выполнимость соотношения (2) сохраняется. Выполнимость соотношения (3) также сохраняется, поскольку его предпосылка ложна. Если полученное сообщение было последним сообщением в очереди сообщений этого канала, то согласно соотношению (2) имеет место равенство $d = D_{w_0}[v_0]$. Отсюда следует, что заключение формулы (3) становится истинным, и, значит, соотношение (3) остается верным. Поскольку предпосылка утверждения (2) становится ложной, выполнимость соотношения (2) сохраняется.

Если $v = v_0$, $u = w_0$ (и u_0 является соседом вершины u), то заключения утверждений (2) или (3) могут стать ложными в том случае, когда значение $D_{w_0}[v_0]$ изменяется в результате выполнения процедуры *Перевычислить*(v) в узле w_0 . Однако в этом случае сообщение $\langle \mathbf{mydist}, v_0, \dots \rangle$ с новым значением отправляется узлу u_0 , и это приводит к тому, что предпосылка утверждения (3) становится ложной, а заключение утверждения (2) становится истинным; поэтому оба утверждения (2) и (3) остаются верными. Это единственно возможный случай, когда сообщение $\langle \mathbf{mydist}, v_0, \dots \rangle$ становится в очередь $Q_{w_0 u_0}$, и в этом случае имеет место равенство $d = D_{w_0}[v_0]$.

Если $v = v_0$ и $u \neq u_0$, w_0 , то в формуле $P(u_0, w_0, v_0)$ ничего не изменится.

Тип 2. Предположим, что канал uw выходит из строя.

Если $u = u_0$ и $w = w_0$, то возникшая неисправность канала приводит к тому, что предпосылки утверждений (2) и (3) становятся ложными, и поэтому выполнимость этих утверждений сохраняется. Выполнимость утверждения (1) сохраняется, ввиду того что w_0 удаляется из списка $Neigh_{u_0}$. То же самое имеет место и в случае $u = w_0$ и $w = u_0$.

Если $u = w_0$, но $w \neq u_0$, то заключения утверждений (2) и (3) могут стать ложными, поскольку может измениться значение $D_{w_0}[v_0]$. Однако в этом случае после отправления узлом w_0 сообщения $\langle \mathbf{mydist}, v_0, \dots \rangle$ предпосылка утверждения (3) становится ложной, а заключение утверждения (2) становится истинным, и поэтому выполнимость утверждений (2) и (3) сохраняется.

Во всех остальных случаях в формуле $P(u_0, w_0, v_0)$ не происходит никаких изменений.

Тип 3. Предположим, что в сети появился канал uw .

Если $u = u_0$ и $w = w_0$, то предикат $up(u_0, w_0)$ принимает значение true, но ввиду того что вершина w_0 добавляется в список $Neigh_{u_0}$ (а вершина u_0 — в список $Neigh_{w_0}$), выполнимость утверждения (1) сохраняется.

После отправления узлом w_0 сообщения $\langle \mathbf{mydist}, v_0, D_{w_0}[v_0] \rangle$ заключение утверждения (2) становится истинным, а предпосылка утверждения (3) становится ложной, и поэтому выполнимость формулы $P(u_0, w_0, v_0)$ сохраняется.

Во всех остальных случаях в формуле $P(u_0, w_0, v_0)$ не происходит никаких изменений. $\square$

Лемма 4.15. Для любой пары вершин u_0 и v_0 формула $L(u_0, v_0)$ является инвариантом.

Доказательство. Первоначально $D_{u_0}[u_0] = 0$ и $Nb_{u_0}[u_0] = local$. Если $v_0 \neq u_0$, то в самом начале $ndis_{u_0}[w, v_0] = N$ для всех $w \in Neigh_{u_0}$, $D_{u_0}[v_0] = N$ и $Nb_{u_0}[v_0] = undef$.

Тип 1. Предположим, что узел u получает сообщение $\langle \mathbf{mydist}, v, d \rangle$ от узла w .

Если $u \neq u_0$ или $v \neq v_0$, то ни одна переменная, фигурирующая в формуле $L(u_0, v_0)$, не изменяет значения. Если $u = u_0$ и $v = v_0$, то значение переменной $ndis_{u_0}[w, v_0]$ изменяется, но при этом $D_{u_0}[v_0]$ и $Nb_{u_0}[v_0]$ перевычисляются в точности так, чтобы выполнялось утверждение $L(u_0, v_0)$.

Тип 2. Предположим, что канал uw оказывается неисправным.

Если $u = u_0$ или $w = u_0$, то список $Neigh_{u_0}$ изменяется, но при этом значения $D_{u_0}[v_0]$ и $Nb_{u_0}[v_0]$ вновь перевычисляются в точности так, чтобы выполнялось утверждение $L(u_0, v_0)$.

Тип 3. Предположим, что в сети образуется новый канал uw .

Если $u = u_0$, то список $Neigh_{u_0}$ изменяется за счет добавления вершины w , но коль скоро в узле u значение переменной $ndis_{u_0}[w, v_0]$ устанавливается равным N , выполнимость формулы $L(u_0, v_0)$ сохраняется. $\square$

4.3.2. Корректность алгоритма Netchange

Теперь мы докажем два требования корректности для рассматриваемого алгоритма.

Теорема 4.16. Как только достигается стабильная конфигурация, таблицы $Nb_u[v]$ удовлетворяют следующим условиям

- 1) если $u = v$, то $Nb_u[v] = local$;
- 2) если существует путь из вершины u в вершину $v \neq u$, то $Nb_u[v] = w$, где w — первый сосед вершины u , который встречается на кратчайшем пути из u в v ;
- 3) если пути из вершины u в вершину v не существует, то $Nb_u[v] = undef$.

Доказательство. Когда алгоритм завершает свое выполнение, наряду с предикатом **stable** выполняется и формула $P(u, w, v)$ для всех троек вершин u , v и w , и отсюда следует, что для всех троек u , v и w справедливо соотношение

$$up(u, w) \implies ndis_u[w, v] = D_w[v]. \quad (4.2)$$

Принимая во внимание утверждение $L(u, v)$ для всех пар вершин u и v , мы получаем следующее соотношение:

$$D_u[v] = \begin{cases} 0, & \text{если } u = v, \\ 1 + \min_{w \in Neigh_u} D_w[v], & \text{если } u \neq v \wedge \exists w \in Neigh_u: D_w[v] < N - 1, \\ N, & \text{если } u \neq v \wedge \forall w \in Neigh_u: D_w[v] \geq N - 1. \end{cases} \quad (4.3)$$

которого достаточно, чтобы доказать, что $D_u[v] = d(u, v)$ всякий раз, когда вершины u и v находятся в одной и той же компоненте связности сети, и $D_u[v] = N$, если u и v находятся в разных компонентах связности.

Вначале мы покажем, воспользовавшись индукцией по $d(u, v)$, что всякий раз, когда вершины u и v находятся в одной и той же компоненте связности сети, верно неравенство $D_u[v] \leq d(u, v)$.

Случай $d(u, v) = 0$. В этом случае $u = v$, и поэтому $D_u[v] = 0$.

Случай $d(u, v) = k + 1$. В этом случае существует такая вершина $w \in Neigh_u$, для которой $d(w, v) = k$. По индуктивному предположению $D_w[v] \leq k$, и отсюда вследствие соотношения (4.3) мы получаем $D_u[v] \leq k + 1$.

Теперь мы продемонстрируем, воспользовавшись индукцией по $D_u[v]$, что всякий раз, когда $D_u[v] < N$, из вершины u в вершину v существует путь, и при этом $d(u, v) \leq D_u[v]$.

Случай $D_u[v] = 0$. Из формулы (4.3) следует, что $D_u[v] = 0$ только тогда, когда $u = v$; это означает, что между вершинами u и v есть пустой путь и $d(u, v) = 0$.

Случай $D_u[v] = k + 1 < N$. Из формулы (4.3) следует, что существует такая вершина $w \in Neigh_u$, для которой $D_w[v] = k$. По индуктивному предположению между вершинами w и v существует путь и $d(w, v) \leq k$; отсюда следует, что между вершинами u и v также есть путь и $d(u, v) \leq k + 1$.

Таким образом, если u и v находятся в одной и той же компоненте связности, то $D_u[v] = d(u, v)$, а в противном случае $D_u[v] = N$. Отсюда, принимая во внимание формулу (4.2) и предложение $\forall u, v: L(u, v)$, получаем заявленный в утверждении теоремы результат о таблицах $Nb_u[v]$. $\square$

Чтобы убедиться в том, что стабильная ситуация рано или поздно наступит, после того как завершатся изменения топологии, мы введем нормирующую функцию по отношению к предикату **stable**. Для конфигурации γ нашего алгоритма будем полагать

$$t_i = (\text{число сообщений типа } \langle \mathbf{mydist}, \dots, i \rangle) + \\ + (\text{число упорядоченных пар } u, v, \text{ для которых } D_u[v] = i),$$

и значением функции f будем считать $(N + 1)$ -местный набор

$$f(\gamma) = (t_0, t_1, \dots, t_N).$$

На множестве таких наборов введем лексикографический порядок $\leq_l$. Здесь нужно вспомнить о том, что $(\mathbb{N}^{N+1}, \leq_l)$ является вполне упорядоченным множеством (упражнение 2.5).

Лемма 4.17. *Обработка сообщений типа **mydist** приводит к уменьшению значения f .*

Доказательство. Предположим, что в узел u , у которого $D_u[v] = d_1$, поступило сообщение $\langle \mathbf{mydist}, v, d_2 \rangle$, и после выполнения перевычисления новое значение переменной $D_u[v]$ становится равным d . Из описания алгоритма вытекает, что $d \leq d_2 + 1$.

Случай $d < d_1$. Тогда $d = d_2 + 1$, и отсюда следует, что значение t_{d_2} (равно как и t_{d_1}) уменьшается на единицу, и только значения t_d , у которых $d > d_2$, увеличиваются. Отсюда следует, что значение функции f уменьшается.

Случай $d = d_1$. Узел u не отправляет никаких новых сообщений типа **mydist**, и на функции f это сказывается только в том, что значение t_{d_2} уменьшается на единицу. Поэтому значение f уменьшается.

Случай $d > d_1$. Тогда значение t_{d_1} (равно как и t_{d_2}) уменьшается на единицу, и только значения t_d , у которых $d > d_1$, увеличиваются. Отсюда следует, что значение функции f уменьшается. $\square$

Теорема 4.18. *Если начиная с какого-то момента топология сети остается неизменной, то спустя конечное число шагов алгоритм достигает стабильной конфигурации.*

Доказательство. Если топология сети не претерпевает никаких изменений, то в дальнейшем происходит только обработка сообщений типа **mydist**, и согласно предыдущей лемме с каждым таким переходом значение функции f уменьшается. В таком случае вследствие фундированности множества значений функции f может произойти только конечное число переходов. Значит, после конечного числа шагов алгоритм достигнет конфигурации, в которой предикат **stable** обращается в истину. $\square$

4.3.3. Обсуждение алгоритма

Формальная корректность алгоритма, гарантирующая сходимость результата вычисления к правильным таблицам за конечное число шагов, после того как произойдет последнее топологическое изменение, мало что говорит нам о настоящем поведении алгоритма. На практике предикат **stable** может иметь значение false большую часть времени (это происходит при частых топологических изменениях), а когда предикат **stable** ложен, ничего определенного о таблицах маршрутизации не известно. Они могут содержать циклы или даже вообще давать неверную информацию о достижимости вершин-адресатов. Поэтому предложенный алгоритм можно использовать только в таких приложениях, где топологические изменения настолько редки, что время сходимости алгоритма невелико по сравнению со средним периодом времени, которое проходит между возникновением двух изменений в топологии сети. Еще более осложняет ситуацию то обстоятельство, что предикат **stable** задает глобальное свойство, и поэтому с точки зрения отдельного узла сети стабильная конфигурация алгоритма неотличима от нестабильной конфигурации. Это означает, что ни один узел не может знать наверняка, соот-

ветствуют ли его таблицы маршрутизации сложившейся топологии сети, и, следовательно, не может задержать отправление пакета, до тех пор пока не будет достигнута стабильная конфигурация.

Асинхронная обработка уведомлений. В этом параграфе мы до сих пор исходили из предположения о том, что уведомления о топологических изменениях обрабатываются автоматически на протяжении одного перехода при возникновении таких изменений. Обработка происходит одновременно по обе стороны удаленного или добавленного канала связи. Лампорт в работе [125] провел более подробный анализ, позволивший учитывать задержку обработки таких уведомлений. Канал связи, ведущий от узла w в узел u , моделируется посредством последовательного соединения трех очередей:

- 1) OQ_{wu} — выходная очередь узла w ;
- 2) TQ_{wu} — очередь сообщений (или пакетов данных), которые уже были переправлены;
- 3) IQ_{wu} — входная очередь узла u .

При нормальном функционировании канала процесс w отправляет сообщения процессу u , помещая их в очередь OQ_{wu} , затем сообщения переходят из очереди OQ_{wu} в очередь TQ_{wu} , из очереди TQ_{wu} в очередь IQ_{wu} , и, наконец, u получает сообщения, удаляя их при этом из очереди IQ_{wu} . Когда в канале возникает неисправность, из очереди TQ_{wu} выбрасываются все сообщения. При этом все сообщения из очереди OQ_{wu} впоследствии также выбрасываются, а не присоединяются к очереди TQ_{wu} , а в конец очереди IQ_{wu} добавляется сообщение $\langle \text{fail}, w \rangle$. Когда возобновляется нормальное функционирование канала связи, в конец очереди IQ_{wu} добавляется сообщение $\langle \text{repair}, w \rangle$. В этом случае предикаты $P(u, w, v)$ будут иметь чуть более сложную структуру, но сам алгоритм останется без изменения.

Маршрутизация по кратчайшим путям. Каждому каналу связи можно приписать весовой коэффициент; после этого рассмотренный нами алгоритм можно модифицировать так, чтобы он вместо путей с наименьшим числом звеньев вычислял кратчайшие пути. Процедура *Перевычисление* в алгоритме Netchange будет учитывать вес канала связи uw при оценке длины кратчайшего пути через узел w , если заменить константу 1 на весовой коэффициент ω_{uw} . При этом константа N должна быть заменена в алгоритме на какую-нибудь верхнюю оценку диаметра сети.

Довольно просто показать, что в том случае, когда модифицированный алгоритм достигает стабильной конфигурации, полученные таблицы маршрутизации действительно являются правильными и задают оптимальные пути (при условии, что все циклы в сети имеют положительный вес). А вот доказательство того, что алгоритм рано или поздно достигает заключительной конфигурации, требует введения более сложной нормирующей функции.

Можно даже провести такое обобщение этого алгоритма, чтобы он мог работать с каналами связи, имеющими переменный вес; реакцией узла u на изменение весового коэффициента канала должно быть перевычисление значений

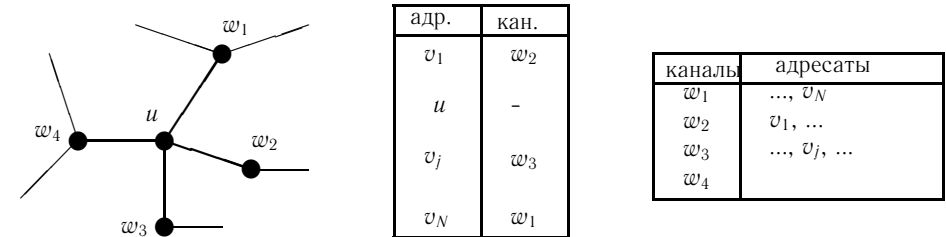


Рис. 4.11. Сокращение размера таблиц маршрутизации

переменных $D_u[v]$ для всех вершин v . Однако такой алгоритм мог бы иметь практическое применение только в тех случаях, когда средний период времени между изменениями весовых коэффициентов каналов связи значительно превосходит время сходимости алгоритма. К сожалению, это совершенно нереальное допущение. В данной ситуации предпочтение отдается таким алгоритмам, которые гарантируют ацикличность таблиц маршрутизации на протяжении всего периода сходимости; к их числу относится, например, алгоритм Мерлина—Сигалла.

4.4. Маршрутизация с использованием компактных таблиц

До сих пор мы имели дело с алгоритмами маршрутизации, которые в каждом узле сети создают и поддерживают таблицы маршрутизации с отдельным входом для каждой вершины-адресата. Когда пакет продвигается по сети, обращение к таблицам происходит в каждом узле пройденного пути (за исключением вершины-адресата). В этом параграфе мы займемся изучением такого устройства таблиц маршрутизации, которое позволяет сократить объем памяти и издержки, связанные с просмотром таблиц. Мы не будем здесь касаться вопросов о том, как вычислять такие таблицы при помощи распределенных алгоритмов. Для простоты описания алгоритмов на протяжении всего этого параграфа мы будем предполагать, что все рассматриваемые сети являются связными.

Общая стратегия получения таблиц меньшего размера, которая используется в каждом из трех алгоритмов, рассматриваемых в этом параграфе, очень проста. Если для каждой вершины-адресата в таблице того или иного узла отдельно указывается соответствующий исходящий канал связи, то таблица маршрутизации будет неизбежно иметь длину N ; таким образом, для хранения таблицы требуется $\Omega(N)$ битов памяти независимо от того, насколько компактно закодированы в ней имена вершин-адресатов. Теперь попробуем организовать устройство таблиц по-другому: для каждого канала связи, примыкающего к узлу, построим список вершин-адресатов, маршруты к которым начинаются с прохождения этого канала (см. рис. 4.11). Если к узлу примыкают deg каналов связи, то и «длина» таблицы будет равна deg ; реальная экономия памяти будет зависеть от того, насколько компактно мы сможем представить множество всех вершин-адресатов, соответствующих каждому каналу связи. Чтобы эффективно проводить поиск в таб-

лицах, они должны быть устроены так, чтобы для заданной вершины-адресата на основе таблиц можно было быстро выбрать соответствующий ей исходящий канал связи.

4.4.1. Схема древесной разметки

Первый компактный метод маршрутизации был предложен Санторо и Хатимом в работе [169]. Этот метод помечает каждую вершину целым числом от 0 до $N - 1$ таким образом, чтобы для каждого канала связи множество вершин-адресатов, сообщение с которыми осуществляется через этот канал, представляло собой интервал. Обозначим символом $\mathbb{Z}_N$ множество целых чисел $\{0, 1, \dots, N-1\}$. Арифметические операции в этом множестве будут проводиться по модулю N , т. е. $(N - 1) + 1 \equiv 0$, но при этом в множестве $\mathbb{Z}$ будет действовать такой же порядок расположения элементов, как в натуральном ряде.

Определение 4.19. Циклическим интервалом $[a, b)$ на множестве $\mathbb{Z}_N$ будем называть всякое множество целых чисел, удовлетворяющее следующему соотношению:

$$[a, b) = \begin{cases} \{a, a+1, \dots, b-1\}, & \text{если } a < b, \\ \{0, \dots, b-1, a, \dots, N-1\}, & \text{если } a \geq b. \end{cases}$$

Следует помнить о том, что $[a, a) = \mathbb{Z}_N$ и для любой пары $a \neq b$ дополнением интервала $[a, b)$ является интервал $[b, a)$. Циклический интервал $[a, b)$ называется *линейным*, если $a < b$.

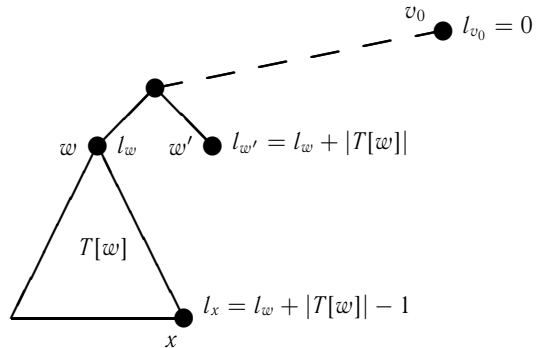


Рис. 4.12. Обход дерева по порядку

Теорема 4.20. Вершины дерева T можно занумеровать таким образом, чтобы для всякой вершины u и каждого исходящего из этой вершины канала множество узлов, путь в которые проходит через этот канал, образовывало циклический интервал.

Доказательство. Выберем произвольную вершину v_0 в качестве корня дерева, и для каждого узла w обозначим символом $T[w]$ поддерево дерева T , имеющее вершину w в качестве корня. Оказывается, можно так занумеровать вершины дерева, что для каждого узла w номера всех вершин в поддереве $T[w]$ будут образовывать линейный интервал; для этого достаточно совершить обход дерева по некоторому порядку, как показано на рис. 4.12. При таком порядке обхода первой вершиной поддерева $T[w]$, которая будет пройдена, является вершина w ; после прохождения вершины w все остальные вершины поддерева $T[w]$ посещаются прежде, чем обход перейдет к остальным вершинам дерева. Таким образом, вершины дерева $T[w]$ получают порядковые номера, которые образуют линейный интервал $[l_w, l_w + |T[w]|)$, где l_w — номер вершины w .

Пусть $[a_w, b_w)$ — это интервал целых чисел, которыми помечены вершины поддерева $T[w]$. Соседом вершины w может являться либо сыновняя, либо родительская вершина узла w . Узел w переправляет сыновней вершине u те пакеты, адресаты которых располагаются в дереве $T[u]$, т. е. имеют номера, лежащие в интервале $[a_u, b_u)$. Узел w переправляет своей родительской вершине все те пакеты, адресаты которых располагаются вне дерева $T[w]$, т. е. имеют номера, лежащие в множестве $\mathbb{Z}_N \setminus [a_w, b_w) = [b_w, a_w)$. $\square$

(* Пакет с адресом d был получен или создан в узле u *)

if $d = l_u$

then пакет доставлен по адресу

else begin выбрать такое число α_i , что $d \in [\alpha_i, \alpha_{i+1})$;
отправить пакет по каналу, помеченному α_i

end

Алгоритм 4.13. Продвижение пакетов на основе интервалов (для узла u).

Для задания одного циклического интервала можно обойтись $2 \log N$ битами, указывая только границы интервала. Поскольку в нашем случае приходится иметь дело с совокупностью непересекающихся интервалов, объединением которых является все множество $\mathbb{Z}_N$, для задания одного интервала будет достаточно $\log N$ битов памяти. Для каждого канала в память заносится только левая граница соответствующего интервала; правая граница равна следующей по порядку левой границе одного из интервалов той же самой вершины. Левая граница интервала, соответствующего каналу uw , примыкающему к вершине u , определяется следующим соотношением:

$$\alpha_{uw} = \begin{cases} l_w, & \text{если } w \text{ является сыновней вершиной для } u, \\ l_u + |T[u]|, & \text{если } w \text{ является родительской вершиной для } u. \end{cases}$$

Предположим, что $\deg_u$ каналов, примыкающих к вершине u , помечены числами $\alpha_1, \dots, \alpha_{\deg_u}$ и при этом $\alpha_1 < \dots < \alpha_{\deg_u}$. Тогда для продвижения сообщений используется алгоритм 4.13. Пометки каналов разбивают множество $\mathbb{Z}_N$ на $\deg_u$ интервалов, каждый из которых соответствует одному каналу (см. рис. 4.14).

Заметим, что самое большое один из этих интервалов может быть нелинейным. Если все пометки каналов упорядочены, то правильную пометку можно найти за $O(\log \deg_u)$ шагов при помощи дихотомического поиска. Индекс i вычисляется по модулю $\deg_u$, т. е. $\alpha_{\deg_u+1} = \alpha_1$.

Схема древесной разметки строит оптимальные маршруты в деревьях, так как в дереве между любыми двумя вершинами есть только один простой путь. Этой схемой можно воспользоваться и тогда, когда сеть не является деревом. В сети выбирается остовное дерево, и указанная схема применяется к этому дереву. Каналы, не принадлежащие остовному дереву, не принимаются в расчет; в таблицах маршрутизации каждый из них помечается специальным символом для обозначения того, что по этим каналам никакие пакеты не продвигаются.

Чтобы сравнить длины путей, которые выбираются схемой древесной разметки, с длинами оптимальных путей, рассмотрим расстояние $d_T(u, v)$ между вершинами u и v в дереве T и расстояние $d_G(u, v)$ между вершинами u и v в графе G . Обозначим символом D_G диаметр графа G , т. е. максимальное расстояние $d_G(u, v)$ по всем парам вершин u и v .

Лемма 4.21. *Не существует конечной верхней оценки для отношения величин $d_T(u, v)$ и $d_G(u, v)$. Это справедливо даже в том случае, когда длина пути полагается равной числу звеньев.*

Доказательство. Рассмотрим кольцо G , состоящее из N вершин. Тогда остовное дерево получается путем удаления из графа G одного канала, например xy . В этом случае $d_G(x, y) = 1$ и $d_T(x, y) = N - 1$, и поэтому отношение рассматриваемых величин равно $N - 1$. Это отношение может быть произвольно большим для больших колец. $\square$

Следующая лемма опирается на симметричность стоимостей каналов связи, т. е. в ней предполагает, что $\omega_{uw} = \omega_{wu}$. Отсюда следует, что $d_G(u, v) = d_G(v, u)$ для всех пар вершин u и v .

Лемма 4.22. *Дерево T может быть выбрано так, чтобы для любой пары вершин u и v выполнялось неравенство $d_T(u, v) \leq 2D_G$.*

Доказательство. Пусть T — оптимальное входное дерево для узла w_0 (так же как в теореме 4.2). Тогда

$$\begin{aligned} d_T(u, v) &\leq d_T(u, w_0) + d_T(w_0, v) = \\ &= d_T(u, w_0) + d_T(v, w_0) = \quad (\text{согласно симметричности } \omega) \\ &= d_G(u, w_0) + d_G(v, w_0) \leq \quad (\text{согласно выбору } T) \\ &\leq D_G + D_G \quad (\text{по определению } D_G). \end{aligned} \quad \square$$

Подводя итоги, заметим, что путь, который выбирается схемой древесной разметки, может быть сколь угодно хуже оптимального пути между той же парой вершин (лемма 4.21), но если выбрать подходящее остовное дерево, то он не более чем в два раза хуже пути между двумя другими вершинами коммуникационной системы (лемма 4.22). Отсюда следует, что схема вполне пригодна, если в большинстве случаев коммуникация проводится между узлами, отстоящими друг от друга на расстояние $\Theta(D_G)$, но не годится, если в большинстве случаев

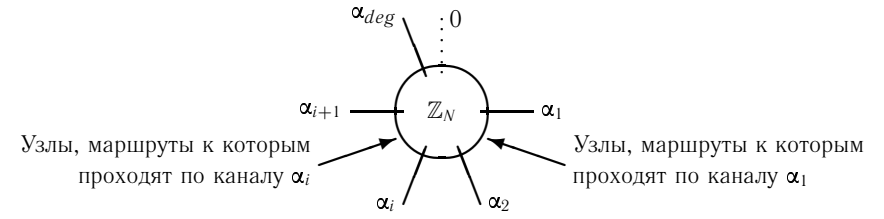


Рис. 4.14. Разбиение множества Z_N в узле

коммуникация проводится между узлами, удаленными друг от друга на короткое расстояние в графе G .

Помимо неудобства, связанного с протяженностью выбираемых путей, схема древесной разметки имеет следующие недостатки.

1. Каналы, не принадлежащие дереву T , не используются, а это означает, что ресурсы сети расходуются неэкономно.
2. Весь трафик сосредоточен в дереве, а это приводит к перегрузкам в сети.
3. Всякая неисправность в одном канале дерева T приводит к разрыву сети.

4.4.2. Интервальная маршрутизация

Ван Ливен и Тан в работе [128] предложили такое обобщение схемы древесной разметки, которое позволило применять эту схему к произвольным сетям таким образом, что почти каждый канал оказывается задействованным в продвижении пакетов.

Определение 4.23. *Схемой интервальной разметки (ILS) называется такая разметка узлов и каналов коммуникационной сети, при которой*

- 1) узлы сети помечаются различными числами из множества Z_N ,
- 2) для каждого узла примыкающие к нему каналы помечаются различными числами из множества Z_N .

Для заданной ILS интервальный алгоритм маршрутизации продвигает пакеты точно так же, как это делает алгоритм 4.13.

Определение 4.24. *Схема интервальной разметки называется правильной, если всякий пакет, который продвигается в сети в соответствии с этой схемой, рано или поздно достигает своего адресата.*

Как будет показано в теореме 4.25, для каждой связной сети G существует правильная схема интервальной разметки однако для произвольной связной сети эта схема обычно не очень эффективна. После того как мы докажем теорему о существовании правильной интервальной разметки, мы изучим вопрос об оптимальности путей, которые выбираются такой схемой.

Теорема 4.25. *Для каждой связной сети G существует правильная схема интервальной разметки.*

Доказательство. Правильная схема интервальной разметки строится путем расширения схемы древесной разметки Санторо и Хатиба, примененной к основному дереву T рассматриваемой сети. Для заданного основного дерева всякое ребро, не принадлежащее этому дереву, мы будем называть *стягивающим ребром*. Кроме того, вершину v будем называть *предком* вершины u , если $u \in T[v]$. Так как при построении схемы главная трудность состоит в проведении разметки стягивающих ребер (все ребра, принадлежащие дереву, будут помечены так, как того требует схема древесной разметки), основное дерево выбирается таким образом, чтобы все стягивающие ребра имели особое расположение. Для этого рассмотрим следующее утверждение.

Лемма 4.26. *Существует такое основное дерево, в котором все стягивающие ребра соединяют вершину u и ее предка.*

Доказательство. Этим свойством обладает всякое основное дерево, которое строится методом обхода сети в глубину (см. [184] и § 6.6.4). $\square$

В дальнейшем мы будем полагать, что дерево T построено методом обхода сети G в глубину.

Определение 4.27. *Схемой интервальной разметки в глубину для сети G (относительно дерева T) называется такая схема разметки, которая удовлетворяет следующим правилам.*

1. Вершины сети помечаются по порядку в соответствии с обходом дерева T , т. е. вершины поддерева $T[w]$ помечаются числами из интервала $[l_w, l_w + |T[w|)]$. Будем полагать $k_w = l_w + |T[w|]$.

2. Пометка α_{uw} ребра uw , примыкающего к узлу u , определяется так:

- а) если uw — это стягивающее ребро, то $\alpha_{uw} = l_w$;
- б) если w — это сыновняя вершина для u (в дереве T), то $\alpha_{uw} = l_w$;
- в) если w — это родительская вершина для u , то $\alpha_{uw} = k_u$ за исключением того случая, когда $k_u = N$ и вершина u имеет стягивающее ребро, соединяющее эту вершину с корнем;

(в последнем случае данное стягивающее ребро в узле u помечается числом 0 согласно правилу (а), и поэтому пометка этого ребра числом k_u нарушала бы принцип однозначности, требующий, чтобы все пометки ребер в узле u были разными; пометки расставляются по модулю N , и поэтому $N \equiv 0$);

- г) Если w — это родительская вершина для u , вершина u имеет стягивающее ребро, соединяющее эту вершину с корнем, и $k_u = N$, то $\alpha_{uw} = l_w$.

Пример схемы интервальной разметки в глубину приведен на рис. 4.15. Следует обратить внимание на то, что все стягивающие ребра помечены согласно правилу (2а), ребра, ведущие в родительские вершины 4, 8 и 10, помечены согласно правилу (2с), а ребро, ведущее в родительскую вершину 9, помечено согласно правилу (2д).

Покажем теперь, что схема интервальной разметки в глубину является правильной схемой. Заметим, что $v \in T[u] \iff l_v \in [l_u, k_u]$. В трех леммах, которые приведены ниже, рассматривается одна и та же ситуация, когда пакет из узла u

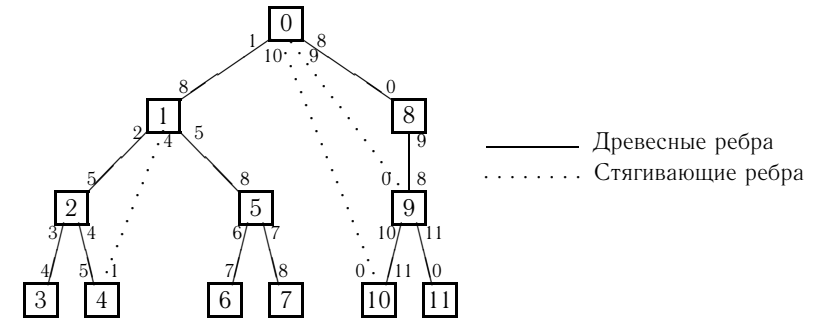


Рис. 4.15. Схема интервальной разметки в глубину

продвигается согласно алгоритму 4.13 по направлению к вершине v через узел w , являющийся соседом вершины u . Это подразумевает, что в узле u для некоторой пометки α выполняется соотношение $l_v \in [\alpha_{uw}, \alpha]$ и нет такой пометки $\alpha' \neq \alpha_{uw}$, для которой в узле u выполняется включение $\alpha' \in [\alpha_{uw}, l_v]$.

Лемма 4.28. *Если $l_u > l_v$, то $l_w < l_u$.*

Доказательство. Вначале рассмотрим случай, когда $\alpha_{uw} \leq l_v$. Вершина w не является сыновней вершиной для u , поскольку в этом случае мы имели бы $\alpha_{uw} = l_w > l_u > l_v$. Если uw — стягивающее ребро, то $l_w = \alpha_{uw} \leq l_v < l_u$. Если w — родительская вершина для u , то в любом случае выполняется неравенство $l_w < l_u$.

Далее рассмотрим случай, когда α_{uw} — это наибольшее число, помечающее ребро, примыкающее к узлу u , и не существует пометки $\alpha' \leq l_v$ (т. е. l_v лежит в нижней части нелинейного интервала). В этом случае ребро, соединяющее u с родительской вершиной, помечено не 0, а числом k_u (поскольку $0 \leq l_v$ и нет пометки $\alpha' \leq l_v$). Тогда число k_u является максимальной пометкой; ребро (древесное или стягивающее), ведущее в вершину-потомок w' , имеет пометку $\alpha_{uw'} = l_{w'} < k_u$, а ребро, ведущее в вершину-предок w' , имеет пометку $\alpha_{uw'} = l_{w'} < l_u$. Значит, w — родительская вершина для u , и отсюда следует неравенство $l_w < l_u$. $\square$

В двух других леммах рассматривается случай, когда $l_u < l_v$. Мы докажем, что тогда либо $v \in T[u]$, либо $l_v \geq k_u$, и при этом в последнем случае выполняется неравенство $k_u < N$, благодаря чему ребро, ведущее в родительскую вершину для u , помечено числом k_u .

Лемма 4.29. *Если $l_u < l_v$, то $l_w \leq l_v$.*

Доказательство. Вначале рассмотрим случай, когда $v \in T[u]$. Рассмотрим ту сыновнюю вершину w' для u , для которой выполняется включение $v \in T[w']$. Тогда имеем соотношения $\alpha_{uw'} = l_{w'} \leq l_v$, из которых следует, что $\alpha_{uw'} \leq \alpha_{uw} \leq l_v < k_{w'}$. Мы заключили, что w не является родительской вершиной для u , и поэтому $l_w = \alpha_{uw}$, откуда следует, что $l_w \leq l_v$.

Далее рассмотрим случай, когда $l_v \geq k_u$. Убедимся, что в этом случае w является родительской вершиной для u . Ребро, ведущее к родительской вершине, помечено числом k_u , и при этом $k_u \leq l_v$. Ребро, ведущее в сыновнюю вершину w' для узла u , помечено числом $l_{w'} < k_u$, стягивающее ребро, ведущее в вершину-потомок w' , помечено числом $l_{w'} < k_u$, а стягивающее ребро, ведущее в вершину-предок w' , помечено числом $l_{w'} < l_u$. Так как w — это родительская вершина для u , выполняется неравенство $l_w < l_u < l_v$. $\square$

Нормирующую функцию, позволяющую отслеживать доставку сообщений в вершину v , можно определить следующим образом. *Наименьшим общим предком* двух узлов u и v называется наиболее удаленная от корня вершина, которая является предком обеих вершин u и v . Условимся, что запись $lca(u, v)$ будет обозначать пометку наименьшего общего потомка вершин u и v , и будем полагать

$$f_v(u) = (-lca(u, v), l_u).$$

Лемма 4.30. Если $l_u < l_v$, то $f_v(w) < f_v(u)$.

Доказательство. Начнем с того случая, когда $v \in T[u]$ (отсюда следует, что $lca(u, v) = l_u$). Если w' — это та сыновняя вершина для u , для которой имеет место включение $v \in T[w']$, то (так же как и в предшествующей лемме) выполняются неравенства $l_{w'} \leq l_w < k_{w'}$, и поэтому верно включение $w \in T[w']$, из которого следуют неравенства $lca(w, v) \geq l_{w'} > l_u$. Таким образом, $f_v(w) < f_v(u)$.

Далее рассмотрим случай, когда $l_v \geq k_u$. Так же как и в предыдущей лемме, w является родительской вершиной для u , и, коль скоро $v \notin T[u]$, выполняется равенство $lca(w, v) = lca(u, v)$. Но теперь $l_w < l_u$, вследствие чего верно неравенство $f_v(w) < f_v(u)$. $\square$

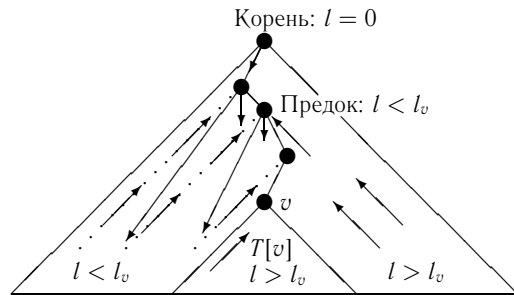


Рис. 4.16. Продвижение пакета по направлению к вершине v в схеме интервальной разметки в глубину

Теперь мы можем убедиться в том, что каждый пакет достигает своего адресата. Поток пакетов, направленных в вершину v , изображен на рис. 4.16. Допустим, что пакет, адресованный узлу v , был сформирован в вершине u . Согласно лемме 4.28, при прохождении каждого звена маршрута метка вершины убывает, до тех пор пока спустя конечное число продвижений пакет не достигнет такой вершины w , для которой выполняется неравенство $l_w \leq l_v$. Согласно лемме 4.29

каждая вершина, которой после этого передается пакет, имеет пометку, не превосходящую числа l_v . Тогда спустя конечное число продвижений пакет достигнет вершины v , так как согласно лемме 4.30, пока пакет не достиг вершины v , при каждом его продвижении значение функции f_v убывает. $\square$

Эффективность интервальной маршрутизации: общий случай. Теорема 4.25 гласит, что для каждой сети есть правильная ILS, но в этой теореме ничего не говорится об эффективности путей, которые выбираются на основе такой схемы. Нужно ясно представлять себе, что схемы интервальной разметки в глубину используются для того, чтобы показать *существование* такого рода схем для каждой сети, но это отнюдь не означает, что это наилучшие из возможных схем. Например, если применить схему интервальной разметки в глубину к кольцу, состоящему из N вершин, то найдутся две такие вершины u и v , что $d(u, v) = 2$, а схеме требуется пройти маршрут из $N - 2$ звеньев, чтобы доставить пакет из вершины u в вершину v (упражнение 4.8). Для того же кольца существует ILS, которая обеспечивает доставку каждого пакета по маршруту с наименьшим числом звеньев (теорема 4.34).

Чтобы оценить качество методов маршрутизации, вначале введем следующее определение.

Определение 4.31. ILS называется *оптимальной*, если она продвигает все пакеты по оптимальным путям. ILS называется *добрососедской*, если она доставляет пакет от любой вершины ко всякому ее соседу за один шаг. ILS называется *линейной*, если каждому ребру соответствует линейный интервал.

Мы будем называть ILS *минимальной по числу звеньев* (или *минимальной по длине пути*), если она является оптимальной относительно меры стоимости, определяемой числом звеньев пути (или соответственно длиной пути). Легко видеть, что минимальная по числу звеньев схема является добрососедской. Также легко проверить, что ILS является линейной в том и только том случае, когда к каждой вершине u с пометкой $l_u \neq 0$ примыкает ребро, помеченное 0, и к каждой вершине с пометкой 0 примыкает ребро с пометкой 0 или 1. Оказывается, для произвольных сетей рассмотренные схемы порождают, вообще говоря, не очень качественные маршруты, но для некоторых типов сетей, имеющих специальную топологическую структуру, качество маршрутов оказывается очень хорошим. Благодаря этому рассмотренные схемы находят применение при решении задачи маршрутизации в процессорных сетях регулярной структуры, наподобие тех, которые возникают в параллельных вычислительных системах с виртуальной глобальной разделяемой памятью.

В точности неизвестно, как для одной и той же произвольной сети соотносятся друг с другом по качеству решений наилучшая схема интервальной разметки и оптимальный алгоритм маршрутизации. Некоторые нижние оценки длины путей, свидетельствующие о том, что оптимальные ILS существуют не всегда, были получены Ружичкой.

Теорема 4.32 ([165]). Существует такая сеть G , что для каждой правильной ILS для сети G найдется пара таких вершин u и v , что пакет

от вершины u к вершине v доставляется по маршруту, состоящему по меньшей мере из $\frac{3}{2}D_G$ звеньев.

Также неизвестно, как соотносятся по качеству решения для одной и той же сети наилучшая схема интервальной разметки в глубину и наилучшая из всех возможных схем интервальной разметки. В упражнении 4.7 приводится очень плохая схема интервальной разметки в глубину для сети, имеющей оптимальную (согласно теореме 4.37) ILS, но, быть может, для той же самой сети есть более качественная схема интервальной разметки в глубину.

В тех случаях, когда в сети друг с другом сообщаются преимущественно соседние вершины, добрососедство является достаточным требованием для ILS. Как видно из рис. 4.15 схема интервальной разметки в глубину не обязательно является добрососедской: узел 4 переправляет пакет в вершину 2 через вершину 1.

Многоинтервальные схемы маршрутизации. Эффективность методов маршрутизации можно повысить, позволив ребрам иметь несколько пометок; в этом случае мы будем говорить о *многоинтервальных схемах маршрутизации*. Действительно, тогда мы можем считать, что множество вершин-адресатов задается объединением нескольких интервалов; увеличивая число интервалов, можно добиться того, что даже для произвольной сети может быть получена оптимальная маршрутизация. Чтобы убедиться в этом, рассмотрим вначале оптимальные таблицы маршрутизации, вычисленные, например, при помощи алгоритма Netchange, и заметим, что для каждого ребра соответствующее множество вершин-адресатов может быть представлено в виде объединения циклических интервалов. При таком элементарном подходе каждому ребру будет приписано не более $N/2$ пометок и для каждой вершины будет не более N пометок. Объем памяти, необходимый для хранения подобных интервальных таблиц, будет точно таким же, какой необходим для хранения классических таблиц полной маршрутизации.

Теперь попробуем понять, как связаны между собой сложность по объему памяти и эффективность маршрутизации. Вполне естественно задаться вопросом о том, сколько разных меток необходимо для получения оптимальной маршрутизации. Начало исследований в этом направлении положила работа Фламмини и др. [83], в которой был разработан метод поиска подходящих схем и доказательства нижних оценок. Из результатов этой работы видно, что в общем случае для достижения оптимальности может потребоваться приписать $\Theta(N)$ пометок каждой линии связи. Но стоит только пойти на маленький компромисс, как сразу же происходит существенное сокращение размера таблиц. Работа Ружички [166] подвела итоги исследования этого компромисса.

Маршрутизация при помощи схем линейных интервалов. Для того чтобы применять интервальные методы маршрутизации, очень важно иметь возможность использовать циклические интервалы. Хотя для некоторых сетей можно построить правильные и даже оптимальные схемы разметки линейными интерва-

лами, тем не менее не каждую сеть можно правильно разметить, ограничившись только линейными интервалами. Вопрос о том, насколько широко можно применять схемы разметки линейными интервалами, был исследован в работе Беккера, ван Ливен и Тана [18].

Теорема 4.33. *Существует такая сеть, для которой нет правильной схемы разметки линейными интервалами.*

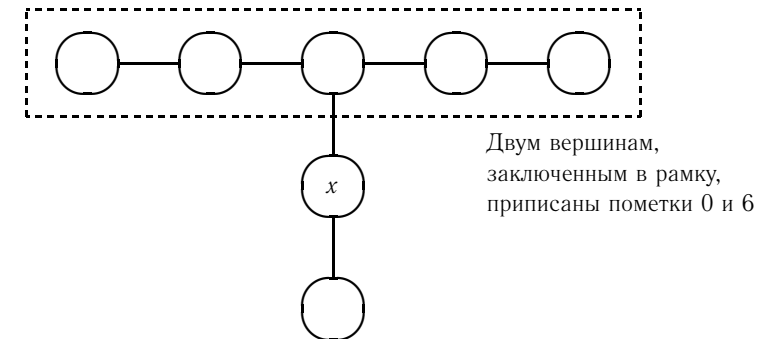


Рис. 4.17. Граф с тремя отростками

Доказательство. Рассмотрим граф с тремя отростками длины два (см. рис. 4.17). Наименьшая пометка (0) и наибольшая разметка (6) приписана двум вершинам, и, коль скоро в графе есть три отростка, имеется хотя бы один отросток, в котором нет ни наибольшей, ни наименьшей пометки. Рассмотрим первую, считая от центра, вершину x в этом отростке. Узел x продвигает в центр пакеты, адресованные узлам с пометками 0 и 6, а единственный линейный интервал, который содержит оба числа 0 и 6, — это все множество $\mathbb{Z}_N$. Следовательно, узел x должен будет также продвигать в центр и те пакеты, которые адресованы другому его соседу, и эти пакеты никогда не достигнут адресата. $\square$

Беккер, ван Ливен и Тан полностью описали класс сетей, топология которых допускает построение линейных ILS, оптимальных по длине маршрута, а также получили ряд результатов, описывающих классы графов, топология которых допускает построение адаптивных линейных ILS, оптимальных по числу звеньев. Полностью охарактеризовать класс сетей, для которых имеются линейные схемы, удалось Fragniaud и Gavoille в работе [94]; Эйлем и др. в работе [76] показали, что длина путей, которые порождаются линейными схемами в таких сетях, может достигать величины $O(D^2)$.

Оптимальность интервальной маршрутизации: топологии специального вида. Оказывается, для некоторых классов сетей, имеющих регулярную структуру, можно построить оптимальные схемы интервальной разметки. Сети с такой

структурой возникают, например, при конструировании параллельных компьютеров.

Теорема 4.34 ([128]). Для колец с N вершинами существуют минимальные по числу звеньев схемы интервальной разметки.

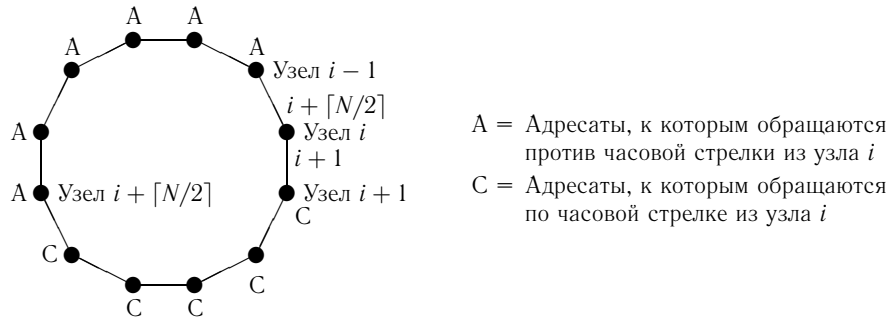


Рис. 4.18. Оптимальная ILS для кольца

Доказательство. Узлы помечаются числами от 0 до $N - 1$ в направлении по часовой стрелке. В узле i каналу, примыкающему к этой вершине в направлении по часовой стрелке, приписывается число $i + 1$, а каналу, примыкающему к этой вершине в направлении против часовой стрелки, приписывается число $(i + \lceil N/2 \rceil) \bmod N$ (см. рис. 4.18). При такой схеме разметки узел с номером i отправляет пакеты, предназначенные вершинам $i + 1, \dots, (i + \lceil N/2 \rceil) - 1$, по часовой стрелке, а пакеты, предназначенные вершинам $(i + \lceil N/2 \rceil), \dots, i - 1$, — против часовой стрелки, и этот способ маршрутизации является оптимальным. $\square$

Коль скоро ILS, построенная при доказательстве теоремы 4.34, является оптимальной, она также является и добрососедской; но она нелинейна.

Теорема 4.35 ([128]). Для решеток размера $n \times n$ существуют минимальные по числу звеньев схемы интервальной разметки.

Доказательство. Пометки приписываются вершинам в порядке расположения рядов, а в каждом ряду в порядке расположения вершин, т.е. i -й вершине, расположенной в j -м ряду, приписывается число $(j - 1)n + (i - 1)$. Канал, ведущий вверх, помечается 0, канал, ведущий влево, помечается числом $(j - 1)n$, канал, ведущий вправо, помечается числом $(j - 1)n + i$, а канал, ведущий вниз, помечается числом $j \cdot n$ (см. рис. 4.19).

Теперь можно легко проверить, что всякий раз, когда узел u отправляет пакет в адрес вершины v , выполняется один из следующих случаев:

случай 1: если вершина v лежит выше, чем вершина u , то u продвигает пакет по каналу, направленному вверх;

случай 2: если вершина v лежит ниже, чем вершина u , то u продвигает пакет по каналу, направленному вниз;

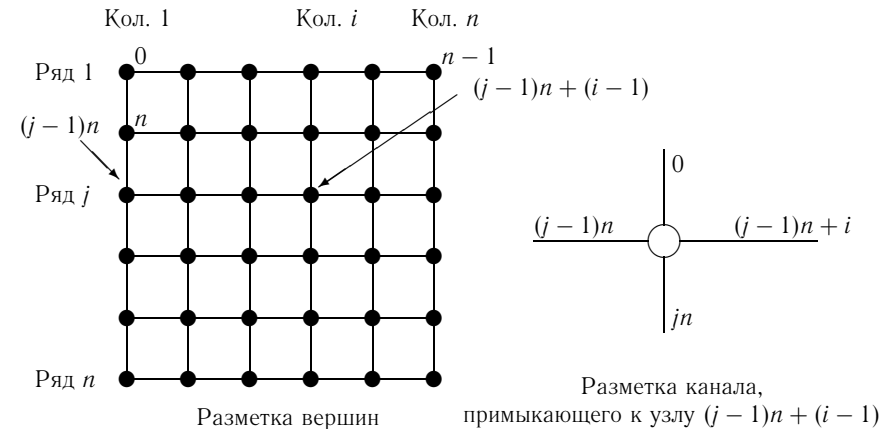


Рис. 4.19. Оптимальная ILS для решетки размера $n \times n$

случай 3: если v лежит в том же ряду, что и u , но слева, то вершина u продвигает пакет по каналу, направленному влево;

случай 4: если v лежит в том же ряду, что и u , но справа, то вершина u продвигает пакет по каналу, направленному вправо.

Во всех этих случаях узел u передает пакет другому узлу, который расположен ближе к вершине v . Отсюда следует оптимальность выбранного пути. $\square$

Коль скоро ILS, построенная при доказательстве теоремы 4.35, является оптимальной, она также является и добрососедской; схема является линейной.

Следующие два результата будут сформулированы без доказательств; в качестве упражнения читателям предлагается построить схемы разметки, о которых говорится в этих теоремах.

Теорема 4.36. Для гиперкубов существуют минимальные по числу звеньев схемы интервальной разметки.

Теорема 4.37 ([93]). Для внешне плоских графов, каналы в которых имеют произвольные весовые коэффициенты, существуют минимальные по длине пути схемы интервальной разметки.

По сравнению с классическими методами маршрутизации, предусматривающими отдельный выбор предпочтительного канала для каждой вершины-адресата, интервальная маршрутизация имеет ряд преимуществ.

1. *Малая сложность по объему памяти.* Если степень вершины равна $\deg$, то для хранения таблицы маршрутизации достаточно $O(\deg \cdot \log N)$ битов памяти.

2. *Эффективное вычисление таблиц маршрутизации.* Таблицы маршрутизации для схемы интервальной разметки в глубину можно вычислить при помощи распределенного поиска в глубину, который проводится с использованием $O(E)$ обменов сообщениями за время $O(N)$ (см. § 6.6.4).

3. *Оптимальность.* Для некоторых классов сетей интервальные методы маршрутизации позволяют выбирать оптимальные пути (см., например, теоремы 4.34–4.37).

Благодаря этим положительным качествам рассмотренные в этом параграфе методы маршрутизации пригодны для процессорных сетей с регулярной топологией. Поскольку такого рода топологические структуры часто образуются в транспьютерах, интервальная маршрутизация реализована в микроэлектронной схеме маршрутизации Inmos C104 (см. § 1.1.5).

К сожалению, в случае применения метода маршрутизации, использующего схему интервальной разметки в глубину, к сетям с произвольной топологией проявляется и целый ряд недостатков.

1. *Плохая устойчивость.* Если в сети происходит обрыв или восстановление одного из каналов, то адаптировать схему интервальной разметки в глубину к этим изменениям весьма непросто. После этих изменений дерево поиска в глубину, положенное в основу ILS, может уже не удовлетворять условию, требующему, чтобы стягивающие ребра всегда соединяли вершину и ее предка. В результате даже небольшое изменение топологии сети может повлечь за собой проведение полного перевычисления всех таблиц маршрутизации, вплоть до проведения новой разметки вершин сети.

2. *Неоптимальность.* Схемы интервальной разметки в глубину могут продвигать пакеты по маршрутам длины $\Omega(N)$ даже в тех случаях, когда диаметр сетей невелик (см. упражнение 4.7).

В научных публикациях рассматривались разные варианты методов интервальной маршрутизации. Многоинтервальные схемы маршрутизации, о которых мы говорили ранее, оказались очень удобными для практических приложений, поскольку современные технологии хранения информации позволяют снизить издержки, связанные с введением дополнительных пометок. Многомерный вариант ILS описан Фламмини и др. в работе [84]. Именами вершин и пометками каналов служат точки и интервалы в многомерном пространстве, а не в множестве $\mathbb{Z}_N$, и этот метод имеет некоторые преимущества в случае его применения к сетям, имеющим многомерную структуру, подобную декартовым произведениям графов.

Более гибким подходом, позволяющим воспользоваться особенностями структуры сети, является *булева маршрутизация*, предложенная Фламмини и др. [82]. Здесь в качестве имен вершин выступают булевы векторы, а ребра помечаются предикатами; сообщение, адресованное узлу с именем λ , может быть отправлено по каналу связи с пометкой $\mathcal{L}$ тогда и только тогда, когда $\mathcal{L}(\lambda)$ имеет истинное значение. Этот метод позволяет эффективно справляться с сетями, обладающими очень сложной структурой, но при этом, конечно, возникают значительные трудности, связанные с интерпретацией пометок.

4.4.3. Префиксная маршрутизация

Чтобы преодолеть недостатки, присущие методам интервальной маршрутизации, Беккер, ван Ливен и Тан в работе [19] разработали метод маршрутизации, в котором таблицы маршрутизации вычисляются на основе произвольного остов-

ного дерева. Снятие всяких ограничений, налагаемых на структуру остовных деревьев, позволяет повысить как устойчивость, так и эффективность маршрутизации. Если добавить новый канал между существующими вершинами, то остовное дерево по-прежнему останется остовным деревом, а новый канал станет в нем стягивающим ребром. Если к графу добавить новый узел вместе с каналами, соединяющими его с существующими вершинами, то остовное дерево можно расширить, добавив новую вершину и один из ее каналов; остальные каналы становятся стягивающими ребрами. Для повышения оптимальности можно выбирать остовные деревья малой глубины, о которых говорится в лемме 4.22.

В префиксной маршрутизации в качестве пометок для вершин и каналов используются строки, а не целые числа, как это принято в интервальной маршрутизации. Пусть задан алфавит Σ ; пометками будут служить строки (слова) в алфавите Σ . Символом ϵ обозначим пустую строку, а для множества всех строк в алфавите Σ будем использовать обозначение Σ^* . Чтобы выбрать канал, по которому нужно продвигать пакет, алгоритм рассматривает все пометки каналов, являющиеся *префиксом* адреса вершины назначения. Выбирается наиболее длинная из таких пометок, и продвижение сообщения проводится по соответствующему каналу. Например, допустим, что каналы, примыкающие к некоторому узлу, помечены строками **aabb**, **abba**, **aab**, **aabc** и **aa** и при этом из этого узла нужно доставить сообщение по адресу **aabbc**. Пометки каналов **aabb**, **aab** и **aa** являются префиксами строки **aabbc**, наиболее длинным префиксом является строка **aabb**, и, следовательно, узел продвигает пакет по каналу, помеченному строкой **aabb**. Процедура маршрутизации описана в алгоритме 4.20. Запись $\alpha \triangleleft \beta$ служит для обозначения того, что строка α является префиксом строки β .

(* Пакет, адресованный узлу d , получен или сформирован в узле u *)

if $d = l_u$

then пакет доставлен по адресу

else begin α_i := самая длинная пометка канала, для которой выполняется $\alpha_i \triangleleft d$;
отправить пакет по каналу с пометкой α_i

end

Алгоритм 4.20. Продвижение на основе префиксов (для узла u).

Определение 4.38. *Схемой префиксной разметки* (в алфавите Σ) для сети G называется такая разметка узлов и каналов связи сети, при которой

- 1) узлы сети G помечаются различными строками в алфавите Σ^* ,
- 2) для каждого узла примыкающие к нему каналы помечаются различными строками.

Для заданной схемы префиксной маршрутизации (PLS) продвижение пакетов проводится согласно алгоритму 4.20.

Определение 4.39. Схема префиксной разметки называется *правильной*, если всякий пакет, продвигающийся в сети в соответствии с этой схемой, достигает своего адресата.

Теорема 4.40. Для каждой связной сети G существует правильная PLS.

Доказательство. Мы определим класс схем префиксной разметки и докажем, подобно тому как мы это сделали в теореме 4.25, что схемы из этого класса являются правильными. Обозначим символом T произвольное корневое остовное дерево графа G .

Определение 4.41. Деревесной PLS для графа G (относительно T) назовем схему префиксной разметки, которая построена согласно следующим правилам:

- 1) корневая вершина помечена строкой ϵ ;
- 2) если w — сыновняя вершина для u , то строка l_w получается из строки l_u добавлением одной буквы, т.е. если $u_1, \dots, u_k$ — это сыновние вершины для u в дереве T , то $l_{u_i} = l_u \cdot a_i$, где $a_1, \dots, a_k$ — k разных букв алфавита Σ ;
- 3) если uw — стягивающее ребро, то $\alpha_{uw} = l_w$;
- 4) если w — сыновняя вершина для u , то $\alpha_{uw} = l_w$;
- 5) если w — родительская вершина для u , то $\alpha_{uw} = \epsilon$, за исключением случая, когда вершина u соединена с корнем стягивающим ребром; в этом случае $\alpha_{uw} = l_w$.

В древесной PLS к каждой вершине, за исключением корневой, примыкает канал, помеченный строкой ϵ , и этот канал соединяет вершину с одним из ее предков (родительской вершиной или корнем дерева). Следует иметь в виду, что для каждого канала uw выполняется одно из двух соотношений $\alpha_{uw} = l_w$ или $\alpha_{uw} = \epsilon$. Для любой пары вершин u и v вершина v является предком вершины u в том и только том случае, когда $l_v \triangleleft l_u$.

Прежде всего нам нужно показать, что ни один пакет не «зависает» в тех узлах, которые не являются его адресатами, т.е. есть каждый узел, не являющийся адресатом пакета, может продвинуть этот пакет, руководствуясь алгоритмом 4.20.

Лемма 4.42. Для любой пары узлов u и v , $u \neq v$, существует канал связи, примыкающий к вершине u и помеченный строкой, которая является префиксом пометки l_v .

Доказательство. Если вершина u не является корнем дерева T , то к u примыкает канал, помеченный пустой строкой ϵ , которая является префиксом l_v . Если u — корень дерева, то вершина v отлична от корня, и поэтому $v \in T[u]$. Если w — это такая сыновняя вершина для u , для которой выполняется включение $v \in T[w]$, то по определению древесной схемы выполняется соотношение $\alpha_{uw} \triangleleft l_v$. $\square$

В трех леммах, которые приведены ниже, рассматривается одна и та же ситуация, когда пакет из узла u продвигается согласно алгоритму 4.20 по направлению к вершине v через узел w , являющийся соседом вершины u .

Лемма 4.43. Если $u \in T[v]$, то w является предком u .

Доказательство. Если $\alpha_{uw} = \epsilon$, то по построению схемы вершина w является предком узла u . Если $\alpha_{uw} = l_w$, то, ввиду того что $\alpha_{uw} \triangleleft l_v$, выполняется соотношение $l_w \triangleleft l_v$. Отсюда следует, что w является предком вершины v , а значит также и предком вершины u . $\square$

Лемма 4.44. Если узел u является предком узла v , то узел w является предком узла v , и при этом узел w расположен к вершине v ближе, чем узел u .

Доказательство. Рассмотрим такую сыновнюю вершину w' для узла u , для которой выполняется включение $v \in T[w']$. Тогда $\alpha_{uw'} = l_{w'}$ — это непустой префикс пометки l_v . Так как α_{uw} — это наиболее длинный префикс (среди всех пометок каналов, примыкающих к u) строки l_v , отсюда следует, что $\alpha_{uw'} \triangleleft \alpha_{uw} \triangleleft l_v$, и поэтому узел w является предком вершины v , который расположен ниже узла u . $\square$

Лемма 4.45. Если $u \notin T[v]$, то либо узел w является предком узла v , либо выполняется неравенство $d_T(w, v) < d_T(u, v)$.

Доказательство. Если $\alpha_{uw} = \epsilon$, то w является либо родительской вершиной для u , либо корнем дерева. Родительская вершина для узла u расположена ближе к узлу v , чем сам узел u , потому что $u \notin T[v]$, а корень дерева является предком узла v . Если $\alpha_{uw} = l_w$, то, ввиду того что $\alpha_{uw} \triangleleft l_v$, узел w является предком узла v . $\square$

Глубиной корневого дерева T называется величина *depth*, равная числу звеньев самого протяженного простого пути, соединяющего корень дерева с одним из его листьев. Как можно заметить, каждый пакет, адресованный узлу v , достигает вершины-адресата, пройдя путь, состоящий не более чем из $2 \cdot \text{depth}$ звеньев. Если пакет сформирован в узле, который является предком вершины v , то согласно лемме 4.44 узел v достигается не более чем за *depth* шагов. Если пакет сформирован в одном из узлов поддерева $T[v]$, то согласно лемме 4.43 один из предков v достигается менее чем за *depth* шагов, после чего, как было только что установлено, пакет достигает вершины v не более чем за *depth* шагов. (Учитывая, что в этом случае путь проходит только через узлы, являющиеся предками вершины-отправителя, можно заключить, что длина пути на самом деле также не будет превышать величины *depth*.) Во всех остальных случаях согласно лемме 4.45 какой-нибудь предок узла v будет достигнут не более чем за *depth* шагов, после чего пакет достигает вершины v не более чем за *depth* шагов. (Таким образом, в этом случае длина пройденного пути ограничена величиной $2 \cdot \text{depth}$.) Этим завершается доказательство теоремы 4.40. $\square$

Следствие 4.46. Для каждой сети G диаметра D_G , измеряемого числом звеньев пути, существует такая схема префиксной разметки, которая гарантирует доставку всех пакетов не более чем за $2D_G$ шагов.

Доказательство. Нужно воспользоваться древесной PLS, дерево которой построено так, как это указано в лемме 4.22. $\square$

Мы завершим наше изучение древесной схемы разметки кратким обсуждением вопроса об объеме памяти, необходимом для ее реализации. Как и прежде, *depth* будет обозначать глубину дерева T . Обозначим символом k максимальное число сыновних вершин для каждого узла дерева T . Тогда самая длинная пометка состоит из *depth* букв, в то время как алфавит Σ должен содержать по

меньшей мере k букв. Это означает, что для хранения произвольной пометки достаточно выделить $depth \cdot \log k$ битов памяти. Вся таблица маршрутизации в узле, к которому примыкают deg каналов связи, укладывается в $O(deg \cdot depth \cdot \log k)$ битов.

Ряд других схем префиксной разметки был предложен в работе Беккера и др. [19]. В этой работе был полностью описан класс всех топологических структур, для которых есть *оптимальная* схема префиксной разметки даже в том случае, когда весовые коэффициенты каналов связи могут динамически изменяться.

4.5. Иерархическая маршрутизация

Один из способов, позволяющих понизить различные показатели стоимости маршрутизации, предлагает воспользоваться *иерархическим разбиением* сети и связанными с ним иерархическими методами маршрутизации. Во многих случаях цель состоит в том, чтобы извлечь пользу из того факта, что большая часть коммуникаций в компьютерной сети проводится локально, т. е. между узлами, которые отстоят друг от друга сравнительно недалеко. Мы сейчас покажем, что некоторые из показателей стоимости метода маршрутизации зависят от размера всей сети, а не от длины выбранного пути.

1. *Длина адресов.* Так как все N узлов сети имеют разные адреса, для хранения каждого адреса требуется не менее $\log N$ битов; если адрес служит для кодирования некоторой информации, как это осуществляется в схеме префиксной маршрутизации, то для представления адреса может потребоваться и большее число битов.

2. *Размер таблицы маршрутизации.* В тех методах маршрутизации, которые были описаны в §§ 4.6.2 и 4.6.3, таблица содержит запись для каждого узла сети, и, следовательно, ее размер линейно зависит от размера сети.

3. *Стоимость обращений к таблице.* Разумно считать, что стоимость одного обращения к таблице возрастает по мере увеличения размеров таблицы и длины адресов. Общее время, затраченное на обращение к таблицам, при доставке одного сообщения зависит также от того, сколько раз приходится обращаться к таблицам.

Иерархический метод маршрутизации предполагает разбиение сети на кластеры, каждый из которых представляет собой связное множество узлов. Если отправитель и адресат пакета находятся в одном кластере, стоимость доставки сообщения будет мала, так как при проведении маршрутизации внутри кластера его можно рассматривать как небольшую изолированную сеть. В том методе, который описан в § 4.5.1, в каждом кластере зафиксирован некоторый узел (*центр* кластера), при помощи которого можно строить и более сложные маршруты, необходимые для доставки пакетов в другие кластеры. Таким образом, работа с обширными таблицами и длинными адресами проводится только в таких центрах. Каждый кластер, в свою очередь, может быть подразделен на подкластеры, благодаря чему достигается многоуровневое разделение узлов.

Совсем необязательно, чтобы взаимосвязь между кластерами осуществлялась только через их центры; недостаток такого способа коммуникации состоит

в том, что весь кластер целиком оказывается чувствительным к неисправностям, которые могут случиться в центре. Ленферт и др. в работе [132] предложили иерархический метод маршрутизации, в котором каждый узел наделен способностью отправлять сообщения за пределы кластера. Тем не менее, в этом методе используются только небольшие таблицы, потому что весь кластер, которому принадлежит узел, рассматривается как одна вершина. Авербах и др. в работе [17] использовали принцип иерархической маршрутизации для построения такого класса схем маршрутизации, в котором достигается компромисс между эффективностью маршрутизации и объемом расходуемой памяти.

4.5.1. Сокращение числа узлов, в которых выбирается маршрут

До сих пор мы изучали только такие методы маршрутизации, в которых выбор маршрута проводится в каждой промежуточной вершине. Это означает, что в маршруте длины l приходится l раз обращаться к таблицам маршрутизации. Для стратегии маршрутизации с наименьшим числом звеньев величина l ограничена диаметром сети, но в общем случае, для ациклической стратегии маршрутизации (наподобие интервальной маршрутизации), верхняя оценка этой величины может быть равна $N - 1$. В этом параграфе мы займемся изучением одного метода, который позволяет сократить количество обращений к таблицам.

Нам потребуется следующая лемма, которая устанавливает возможность разбиения сети на связанные кластеры.

Лемма 4.47. Для любого числа $s \leq N$ существует такое разбиение сети на кластеры $C_1, \dots, C_m$, что

- 1) каждый кластер является связным подграфом;
- 2) каждый кластер содержит не менее s вершин;
- 3) радиус каждого кластера не превышает $2s$.

Доказательство. Рассмотрим максимальную совокупность $D_1, \dots, D_m$ попарно непересекающихся связных подграфов, каждый из которых имеет радиус не больше s и содержит не менее s вершин. Каждая вершина, не вошедшая в множество $\cup_{i=1}^m D_i$, может быть соединена с одним из этих подмножеств путем, длина которого не превосходит s , ибо в противном случае такой путь можно было бы взять в качестве отдельного кластера. Образует новые кластеры C_i путем добавления каждой вершины, не вошедшей в множество $\cup_{i=1}^m D_i$, к ближайшему к ней кластеру D_i . Полученные в результате этого расширения кластеры по-прежнему содержат не менее s вершин каждый, по-прежнему остаются связными, и их радиус не превосходит величины $2s$. $\square$

В рассматриваемом методе маршрутизации каждому пакету приписывается окраска, причем количество разных цветов невелико. Узлы ведут себя следующим образом. Пакет, в зависимости от его окраски, либо немедленно продвигается по заранее выделенному каналу, либо обращается с вызовом к более сложной

процедуре выбора маршрута. Здесь допускается возможность того, что для обработки пакетов разные узлы используют разные протоколы.

Теорема 4.48 ([127]). *Для каждой сети, состоящей из N узлов, существует такой метод маршрутизации, которому требуется обращаться не более $O(\sqrt{N})$ раз к таблицам маршрутизации для доставки каждого пакета; в этом методе используются три цвета.*

Доказательство. Предположим, что задано такое разбиение сети на m кластеров, о котором говорится в лемме 4.47. В таком случае в каждом кластере C_i есть такой узел c_i , что для любой вершины $v \in C_i$ выполняется неравенство $d(v, c_i) \leq 2s$, поскольку радиус кластера C_i не превосходит $2s$. Рассмотрим поддерево T графа G , которое имеет минимальный размер среди всех поддеревьев, соединяющих все вершины c_i . Так как поддерево T является минимальным, у него есть не более m листьев, также не более $m - 2$ вершин-развилки (т. е. вершин, степень которых больше 2, см. упражнение 4.9). Вершины дерева T разделим на три класса: *центры* c_i , вершины-развилки и оставшиеся вершины, которые будем называть *путевыми*.

Наш метод построения маршрута работает так: вначале пакет переправляется в центр c_i того кластера, которому принадлежит узел-отправитель (зеленая фаза), затем пакет переправляется по дереву T в центр c_j того кластера, которому принадлежит узел-адресат (синяя фаза), и, наконец, он переправляется в кластере C_j в сам узел-адресат (красная фаза). В зеленой фазе используются заранее выбранные входные деревья, корнями которых служат центры каждого кластера; в этой фазе нет никаких обращений к таблицам. Путевые вершины дерева T инцидентны в точности двум древесным каналам, и поэтому, получив синий пакет по одному из каналов, они всякий раз продвигают его по другому каналу. В вершинах-развилках и центрах дерева T для построения маршрута приходится обращаться к таблицам. В красной фазе для продвижения пакета применяется стратегия построения кратчайших маршрутов в пределах одного кластера; поэтому число обращений к таблицам в этой фазе ограничено величиной $2s$. Таким образом, общее число обращений к таблицам ограничено величиной $2m - 2 + 2s$, которая, в свою очередь, не превосходит величины $2N/s - 2 + 2s$. Выбрав $s \approx \sqrt{N}$, мы получаем требуемую оценку $O(\sqrt{N})$. $\square$

В теореме 4.48 установлена верхняя оценка общего числа обращений к таблицам, которые необходимо совершить для доставки каждого пакета, и эта оценка не зависит от какого-либо конкретного алгоритма построения таблиц маршрутизации. В качестве метода построения маршрута в дереве T можно взять древесную схему маршрутизации Санторо и Хатиба, но и к самому дереву T можно также применить принцип кластеризации, чтобы еще более сократить число обращений к таблицам.

Теорема 4.49 ([127]). *Для любой сети, состоящей из N вершин, и для любого натурального числа $f \leq \log N$ существует метод построения маршрутов, позволяющий доставлять пакеты по любому адресу, осуществляя*

при этом не более $O(f \cdot N^{1/f})$ обращений к таблицам маршрутизации и используя $2f + 1$ цветов.

Доказательство. Идея доказательства подобна той, которая использовалась для доказательства теоремы 4.48, однако, вместо того, чтобы выбирать $s \approx \sqrt{N}$, метод построения схемы маршрутизации применяется рекурсивно к дереву T (с тем же самым размером кластера s). Это дерево является связной сетью, в которой, по существу, имеется менее $2m$ вершин, поскольку на путевые вершины дерева T , перекладывающие пакеты из одного канала в другой, можно не обращать внимания.

Эта кластеризация повторяется f раз. Допустим, что в исходной сети G содержится N узлов. Тогда дерево, построенное после первой кластеризации, содержит не более N/s центров и N/s вершин-развилки, т. е. всего не более $N(2/s)$ существенных вершин. Если же дерево, построенное после i -кратной кластеризации, содержит m_i существенных вершин, то после проведения очередной $(i + 1)$ -й кластеризации будет получено дерево, в котором содержится не более m_i/s центров и m_i/s вершин-развилки, т. е. не более $m_i(2/s)$ существенных вершин. Дерево, построенное после f -кратной кластеризации, будет иметь не более $m_f = N(2/s)^f$ существенных вершин.

На каждом уровне кластеризации добавляются два новых цвета, и поэтому для проведения f -кратной кластеризации необходимо использовать $2f + 1$ цветов. На самом верхнем уровне кластеризации к таблицам приходится обращаться не более $2m_f$ раз, и, кроме того, на каждом промежуточном уровне кластеризации к таблицам маршрутизации приходится обращаться s раз, чтобы локализовать вершину-адресат. В результате общее число обращений к таблицам становится равным $2m_f + fs$. Если выбрать $s \approx 2N^{1/f}$, то получим $m_f = O(1)$; поэтому число обращений к таблицам будет ограничено величиной $f \cdot s = O(f \cdot N^{1/f})$. $\square$

Если мы будем использовать приблизительно $\log N$ цветов, то в результате можем получить метод построения маршрутов, в котором требуется осуществлять $O(\log N)$ обращений к таблицам маршрутизации. Однако проверка окраски пакета в этом случае также становится разновидностью обращения к таблице, хотя для этой цели приходится использовать маленькие таблицы (размера $O(\log N)$ в худшем случае), и доля вершин, в которых проверяется окраска, также мала.

4.6. Упражнения к главе 4

4.6.1

Упражнение 4.1. Допустим, что таблицы маршрутизации так обновляются после каждого изменения топологической структуры сети, что они остаются ациклическими по ходу обновления. Может ли это служить гарантией того, что пакеты всегда доставляются по адресу даже в том случае, когда сеть претерпевает бесконечно большое количество топологических изменений?

Докажите, что ни один алгоритм маршрутизации не способен обеспечить доставку пакетов по адресу, если сеть испытывает непрерывные изменения топологии.

4.6.2

Упражнение 4.2. Студент предложил исключить из алгоритма 4.6 отправку сообщений $\langle \text{pys}, w \rangle$. Он привел следующий аргумент: узел u так узнает о том, что его сосед *не является* сыновней вершиной, если от этого соседа не поступает сообщений $\langle \text{ys}, w \rangle$.

Можно ли таким образом модифицировать алгоритм? Какова будет сложность такого алгоритма?

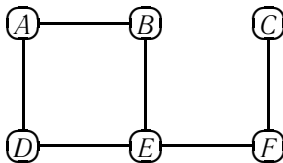
Упражнение 4.3. Докажите, что приведенная ниже формула задает инвариант алгоритма Чанди—Мисры вычисления путей, ведущих в вершину v_0 (алгоритм 4.7).

$$\forall u, w: \langle \text{mydist}, v_0, d \rangle \in M_{wu} \implies d(w, v_0) \leq d \\ \wedge \forall u: d(u, v_0) \leq D_u[v_0].$$

Приведите пример выполнения, в котором количество обменов сообщениями экспоненциально зависит от числа каналов в сети.

4.6.3

Упражнение 4.4. Выясните, какие значения будут иметь все переменные в заключительной конфигурации алгоритма Netchange в том случае, когда этот алгоритм применяется к сети, имеющей следующую топологическую структуру:



После того как была достигнута заключительная конфигурация, в сети возник новый канал связи между узлами A и F . Какое сообщение узел F отправит узлу A при обработке уведомления $\langle \text{repair}, A \rangle$? Какое послание узел A отправит узлу F в ответ на это сообщение?

4.6.4

Упражнение 4.5. Приведите пример, свидетельствующий о том, что лемма 4.22 неверна для сетей с асимметричной характеристикой стоимости каналов.

Упражнение 4.6. Существует ли такая ILS, в которой для построения маршрутов используются *не все* каналы? Существует ли правильная схема, обладающая этим свойством? Существует ли оптимальная схема, обладающая этим свойством?

Упражнение 4.7. Приведите пример такого графа G и такого дерева поиска в глубину T для графа G , которые обладают следующими свойствами: граф G

имеет $N = n^2$ вершин, диаметр графа G и глубина дерева T являются величинами порядка $O(n)$, и при этом существует такая пара узлов u и v , что схема интервальной разметки в глубину обеспечивает доставку пакета из вершины u в вершину v по маршруту, состоящему из $N - 1$ звеньев.

(Граф можно выбрать таким, что G будет внешне планарным, и отсюда согласно теореме 4.37 следует, что G действительно имеет оптимальную ILS.)

Упражнение 4.8. Постройте схему интервальной разметки в глубину для кольца, состоящего из N вершин. Покажите, что существуют такие узлы u и v , что $d(u, v) = 2$, а схема вынуждена построить маршрут, состоящий из $N - 2$ звеньев, для доставки пакета из вершины u в вершину v .

4.6.5

Упражнение 4.9. Докажите, что из минимальности дерева T , используемого в доказательстве теоремы 4.48, следует, что это дерево имеет не более m листьев. Докажите, что всякое дерево с m листьями имеет не более $m - 2$ вершин-развилок.

ГЛАВА 5

НЕБЛОКИРУЕМАЯ КОММУТАЦИЯ ПАКЕТОВ

Сообщения (пакеты), перемещающиеся по коммуникационной сети с пакетной коммутацией, должны сохраняться в памяти каждого узла, перед тем как они отправлены в следующий узел по пути их следования к вершине-адресату. Для этой цели в области памяти каждого узла заводятся специальные буферы. Так как емкость буферов в каждом узле конечна, может случиться так, что ни один пакет не может быть продвинут, потому что все буферы в следующем узле сети уже заполнены. Эта ситуация изображена на рис. 5.1. Каждый из четырех узлов имеет B буферов, и в каждый буфер можно поместить ровно один пакет. Из узла s в узел t были отправлены B пакетов, адресованных узлу v , а из узла v в узел u были отправлены B пакетов, адресованных узлу s . Все буферы в узлах u и v теперь заполнены, и вследствие этого ни один пакет, хранящийся в узлах t и u , нельзя переправить по назначению.

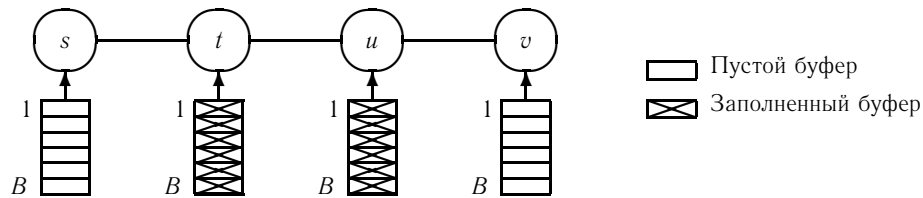


Рис. 5.1. Пример блокировки хранения-продвижения

Ситуацию, когда группа пакетов ни за что не может достичь своего адресата, из-за того что каждый из них простаивает в ожидании пока освободиться буфер, занятый другим пакетом из той же группы, принято называть *блокировкой* или *тупиком хранения-продвижения*. (В конце этой главы мы бегло ознакомимся с другими типами тупиковых ситуаций.) При проектировании сетей с коммутацией пакетов важно уметь справляться с блокировками хранения-продвижения. В этой главе мы обсудим ряд методов (их обычно называют *контроллерами*), которые можно использовать для того, чтобы не допустить возникновения тупиковых ситуаций, путем наложения определенных ограничений на порядок формирования и продвижения пакетов. Методы предотвращения ситуаций блокировки хранения-продвижения реализуются на сетевом уровне эталонной модели взаимодействия открытых систем OSI (§ 1.2.2).

Мы рассмотрим две разновидности этих методов, основанных на *структурированном* и *неструктурированном* устройстве области буферной памяти. Методы, использующие структурированную буферную память (§ 5.5.2), выделяют в узле для каждого пакета особый буфер, который должен быть занят всякий раз, когда пакет сформирован или получен. Если этот буфер уже занят, то принимать этот пакет нельзя. Методы, использующие неструктурированную буферную память (§ 5.5.3), имеют дело с равноправными буферами; в таком методе лишь предписывается, можно или нельзя принимать пакет, но не указывается, в какой буфер следует его поместить. Необходимые определения и обозначения мы введем в § 5.5.1, а в заключение обсудим в § 5.4 ряд дальнейших вопросов.

5.1. Введение

Как обычно, сеть моделируется графом $G = (V, E)$; расстояние между вершинами измеряется числом звеньев пути. В каждом узле выделено B буферов для временного хранения пакетов. Множество всех буферов обозначим символом B , а отдельные буфера будем обозначать символами b, c, b_u , и т. п.

Обработка пакетов в узлах сети осуществляется при помощи действий следующих трех типов, которые могут выполняться в сети.

1. *Формирование пакета*. В узле u «создается» новый пакет p (в действительности прием этого пакета происходит в протоколе более высокого уровня) и помещается пустой буфер узла u . В этом случае узел u называется *источником* пакета p .

2. *Продвижение пакета*. Пакет p перемещается из узла u в свободный буфер следующего узла w на маршруте его продвижения (выбор маршрута осуществляет используемый алгоритм маршрутизации). В результате этого действия опустошается буфер, ранее занятый пакетом p . И хотя контроллеры, которые мы далее опишем, могут наложить запрет на это действие, предполагается, что в сети такой шаг всегда допустим, т. е. если контроллер не запрещает это действие, то его можно осуществить.

Легко видеть, что в системах с синхронным обменом сообщениями это действие выполняется за один переход согласно определению 2.7. В системах с асинхронным обменом сообщениями такое действие нельзя выполнить в рамках одного перехода, руководствуясь определением 2.6, но его можно реализовать, например, следующим образом. Узел u неоднократно пересылает пакет p в узел w , но не удаляет этого пакета из буфера, до тех пор пока не получит подтверждения. Когда пакет p поступает в узел w , то процесс w принимает решение о том, может ли этот пакет быть помещен в один из буферов. Если это возможно, то пакет помещается в нужный буфер и узлу u отправляется подтверждение; в противном случае пакет просто игнорируется. Конечно, можно предложить и более эффективные способы реализации указанного действия. Например, можно потребовать, чтобы узел u не отправлял пакет p , до тех пор пока он не будет осведомлен о том, что узел w примет пакет p . Во всяком случае данное действие состоит из нескольких переходов того типа, который был введен в определении 2.6, но в этой главе нам выгодно считать, что действие выполняется за один шаг.

3. *Поглощение пакета.* Пакет p , занимающий некоторый буфер узла-адресата, удаляется из буфера. Считается, что в сети такое действие всегда разрешено.

Обозначим символом $\mathcal{P}$ совокупность всех путей, по которым могут следовать пакеты. Эта совокупность определяется конкретным алгоритмом маршрутизации (см. гл. 4); мы не будем касаться здесь вопроса о том, как образуется множество $\mathcal{P}$. Пусть k обозначает число звеньев самого протяженного пути из множества $\mathcal{P}$. Совсем не обязательно, чтобы число k было равно диаметру графа G ; число k может превосходить диаметр графа, если алгоритм маршрутизации не отдает предпочтение кратчайшим путям, но может быть также и меньше диаметра, если взаимосвязь в графе G осуществляется только между узлами, находящимися недалеко друг от друга.

Как видно из примера, приведенного в начале этой главы, тупиковые ситуации могут возникать, если разрешается неограниченное выполнение указанных действий (с учетом того, что в узле u должен быть пустой буфер, для того чтобы сформировался пакет, и узел w должен иметь свободный буфер, чтобы принять направленный в этот узел пакет). Мы приведем определение контроллера — алгоритма, который разрешает или запрещает выполнение тех или иных действий в сети, подчиняясь при этом следующим требованиям.

1. Всегда разрешено поглощение пакета, доставленного по назначению.
2. Всегда разрешено формирование пакета в узле, все буферы которого свободны.

3. Контроллер может использовать только локальную информацию; иными словами, решение о том, может ли узел u принять тот или иной пакет, вырабатывается на основе той информации, которой располагает узел u или которая содержится в самом пакете.

Второе из приведенных требований исключает тривиальные решения задачи предотвращения блокировок (см. определение 5.2) путем установления всеобъемлющего запрета на прием какого-либо пакета в сети. Как и в гл. 2, мы будем использовать запись Z_u для обозначения множества всех состояний процесса u , а запись $\mathcal{M}$ — для обозначения множества всех возможных сообщений (пакетов).

Определение 5.1. *Контроллером* для сети $G = (V, E)$ называется совокупность пар $\text{con} = \{\text{Gen}_u, \text{For}_u\}_{u \in V}$, где $\text{Gen}_u \subseteq Z_u \times \mathcal{M}$ и $\text{For}_u \subseteq Z_u \times \mathcal{M}$. Если $c_u \in Z_u$ — это такое состояние процесса u , в котором все буферы узла u свободны, то для любого пакета $p \in \mathcal{M}$ имеет место включение $(c_u, p) \in \text{Gen}_u$.

Контроллер **con** разрешает формирование пакета p в узле u , который находится в состоянии c_u , тогда и только тогда, когда $(c_u, p) \in \text{Gen}_u$, и разрешает продвижение пакета p из узла u в узел w тогда и только тогда, когда $(c_w, p) \in \text{For}_w$. В формальном определении контроллера нет никаких условий, касающихся поглощения пакетов, потому что поглощение пакетов (в узле-адресате) всегда разрешено. Действиями сети под управлением контроллера **con** являются все те и только те действия сети, которые разрешены контроллером **con**.

Говорят, что пакет заблокирован в сети, если никакая последовательность действий сети не позволяет доставить этот пакет по назначению.

Определение 5.2. Для заданной сети G , контроллера **con** для G и конфигурации γ сети G пакет p (присутствующий в конфигурации γ) называется *заблокированным*, если не существует никакой последовательности действий под управлением контроллера **con**, применимой к конфигурации γ , в результате выполнения которых пакет p будет поглощен. Конфигурация называется *тупиковой*, если в ней присутствует заблокированный пакет.

Как следует из примера, изображенного на рис. 5.1, тупиковые конфигурации существуют для любого контроллера. Задача контроллера состоит в том, чтобы не позволить сети попасть в такие конфигурации. Начальными конфигурациями сети объявляются такие конфигурации, в которых отсутствуют какие-либо пакеты.

Определение 5.3. Контроллер называется *беступиковым*, если под управлением этого контроллера невозможно попасть из начальной конфигурации в тупиковую.

5.2. Структурированные решения

Мы рассмотрим здесь класс контроллеров, в основу которых положены т. н. буферные графы, введенные Мерлином и Швейцером в работе [148]. Принцип устройства буферных графов определяется тем соображением, что тупиковая ситуация (при отсутствии контроллера) возникает, как только образуется цикл ожиданий. Цикл ожиданий предполагает наличие такой последовательности пакетов $p_0, \dots, p_{s-1}$, что для каждого i пакет p_i стремится переместиться в буфер, занятый пакетом p_{i+1} (индексы вычисляются по модулю s). Циклических ожиданий можно избежать, если перемещать пакеты по путям в ациклическом графе (буферном графе). В § 5.2.1 будут даны определения буферных графов и сопутствующего им класса контроллеров, а также будут приведены два простых примера буферных графов. В § 5.2.2 будет рассказано о более изощренном способе построения буферных графов, а также будут приведены два иллюстрирующих примера.

5.2.1. Буферные графы

Пусть задана сеть G с множеством буферов $\mathcal{B}$.

Определение 5.4. *Буферным графом* (для графа G с буферами $\mathcal{B}$) называется ориентированный граф BG , вершинами которого служат буферы сети, т. е. $BG = (\mathcal{B}, \vec{BE})$, удовлетворяющий следующим условиям:

- 1) BG — ациклический граф (граф, не содержащий контуров);
- 2) если $bc \in \vec{BE}$, то b и c либо являются буферами одного и того же узла, либо являются буферами двух узлов, соединенных каналом связи в графе G ;
- 3) для каждого пути $P \in \mathcal{P}$ существует путь в графе BG , образом которого (см. ниже) является путь P .

Второе условие задает отображение множества путей в графе BG в множество путей в графе G . Это отображение устроено так. Рассмотрим произвольный путь

$b_0, b_1, \dots, b_s$ в графе BG и сопоставим каждому буферу b_i тот узел u_i , в котором располагается этот буфер. В результате получим такую последовательность узлов $u_0, u_1, \dots, u_s$, в которой для любого номера $i < s$ выполняется либо включение $u_i u_{i+1} \in E$, либо включение $u_i = u_{i+1}$. Тот путь в графе G , который получается из этой последовательности за счет удаления последовательно идущих дубликатов одного и того же узла, будем называть *образом* исходного пути $b_0, b_1, \dots, b_s$ из BG .

Пакет нельзя помещать в произвольно выбранный буфер; он должен быть помещен в такой буфер, из которого сможет впоследствии достичь узла-адресата, следуя по пути в графе BG . В приведенном ниже определении как раз говорится о том, какой буфер следует считать подходящим.

Определение 5.5. Пусть пакет p , адресованный узлу v , пребывает в узле u . Буфер b узла u считается *подходящим* для пакета p , если в графе BG существует путь из вершины b в какой-нибудь буфер c узла v , и образом этого пути в графе BG является один из тех путей в графе G , по которому пакет p может следовать в сети G .

Один из таких путей в графе BG будет выделен как *гарантированный путь*, и запись $nb(p, b)$ будет использоваться для обозначения следующего буфера в этом гарантированном пути. Для каждого вновь сформированного в узле u пакета p существует предписанный этому пакету подходящий буфер $fb(p)$.

Знакосочетания fb и nb являются акронимами выражений «first buffer» и «next buffer». Следует иметь в виду, что буфер $nb(p, b)$ всегда является подходящим для пакета p . Во всех буферных графах, которые мы будем рассматривать в этом параграфе, буфер $nb(p, b)$ будет располагаться в другом узле, нежели тот, которому приписан буфер b . Впоследствии мы обсудим использование «внутренних» ребер в графе BG , т. е. тех ребер, которые инцидентны двум буферам одного и того же узла.

Контроллер буферного графа. Буферный граф BG можно использовать для реализации беступикового контроллера $\mathbf{bgc}_{BG}$ при условии, что буфер $nb(p, b)$ будет закодирован в каждом пакете p и/или состоянии того узла, в котором оказывается пакет p .

Определение 5.6. Контроллер $\mathbf{bgc}_{BG}$ устроен следующим образом.

1. Пакет p разрешается сформировать в узле u в том и только том случае, когда буфер $fb(p)$ свободен. Как только этот пакет будет сформирован, он помещается в указанный буфер.

2. Пакет p разрешается переправить из буфера в узле u в буфер узла w (при этом возможно, что $u = w$) в том и только том случае, когда буфер $nb(p, b)$ (в узле w) свободен. Если это перемещение происходит, то пакет p помещается в буфер $nb(p, b)$.

Теорема 5.7. Контроллер $\mathbf{bgc}_{BG}$ является беступиковым.

Доказательство. Если в узле u все буферы пусты, то в узле u разрешается сформировать любой пакет. Это означает, что $\mathbf{bgc}_{BG}$ действительно является контроллером.

Для каждого буфера сети $b \in \mathcal{B}$ назовем *классом* буфера b длину самого протяженного пути в графе BG , который оканчивается в вершине b . Нужно заметить, что классы буферов в любом пути в графе BG (в частности, в гарантированном пути) строго возрастают, т. е. класс буфера $nb(p, b)$ превосходит класс буфера b .

Так как контроллер разрешает помещать пакеты только в подходящие буферы и в самом начале нет никаких пакетов, в каждой достижимой конфигурации сети, находящейся под управлением контроллером $\mathbf{bgc}_{BG}$, пакеты содержатся только в подходящих буферах. Воспользовавшись нисходящей индукцией по классам буферов, нетрудно показать, что в такой конфигурации ни один буфер класса r не содержит заблокированного пакета. Обозначим символом R наивысший класс буферов.

Случай $r = R$. Буфер b наивысшего класса R , расположенный в узле u , не имеет исходящих дуг в графе BG . Следовательно, всякий пакет, для которого буфер b является подходящим, адресован узлу u и может быть поглощен, как только он окажется в буфере b . Отсюда следует, что ни один буфер класса R не содержит заблокированных пакетов.

Случай $r < R$. Индуктивная гипотеза гласит, что для любого числа r' , удовлетворяющего неравенству $r < r' \leq R$, ни один буфер класса r' не содержит в достижимых конфигурациях заблокированного пакета.

Рассмотрим произвольную достижимую конфигурацию γ , в которой пакет p помещен в буфер b класса $r < R$, приписанный узлу u . Если узел u является адресатом пакета p , то пакет p может быть поглощен и поэтому не может считаться заблокированным. В противном случае рассмотрим следующий буфер $nb(p, b) = c$ в гарантированном пути из вершины b и заметим, что класс r' буфера c превосходит число r . По индуктивной гипотезе буфер c не может содержать заблокированного пакета, и поэтому есть конфигурация δ , достижимая из конфигурации γ , в которой буфер c пуст. В конфигурации δ пакет p может быть перемещен в буфер c , и на основании индуктивной гипотезы этот пакет не считается заблокированным в результирующей конфигурации δ' . Следовательно, пакет p не заблокирован в конфигурации γ . $\square$

Как видно из доказательства теоремы, если в гарантированном пути встречаются «внутренние» дуги буферного графа (дуги между буферами одного и того же узла), то контроллер обязан разрешить выполнить дополнительное действие, в результате чего пакет перемещается в другой буфер того же самого узла. Обычно гарантированный путь не содержит таких дуг. Они используются только в качестве необязательных действий, чтобы повысить эффективность продвижения пакетов, как это происходит в следующей ситуации. Предположим, что пакет p помещен в буфер b_1 узла u , а буфер $nb(p, b_1)$ в узле w занят. Однако в узле u может быть свободный буфер b_2 , подходящий для пакета p , и, кроме того, буфер $nb(p, b_2)$ в узле w может оказаться свободным. В таком случае пакет может быть перемещен через буферы b_2 и $nb(p, b_2)$.

Теперь мы обсудим два примера использования буферных графов, а именно схему назначения и схему пройденных шагов.

Схема назначения. В схеме назначения в каждом узле u выделяется N буферов, по одному буферу $b_u[v]$ для каждой возможной вершины-адресата v . Узел v называется *назначением* буфера $b_u[v]$. В этой схеме предполагается, что алгоритм маршрутизации продвигает все пакеты, адресованные узлу v , по ориентированному дереву T_v , корнем которого является узел v . (На самом деле это допущение можно ослабить; достаточно того, чтобы каналы, предназначенные для построения маршрутов к узлу v , образовывали ациклический подграф графа G .)

Пусть задан буферный граф $BG_d = (B, \vec{BE})$, в котором $b_u[v_1]b_w[v_2] \in \vec{BE}$ тогда и только тогда, когда $v_1 = v_2$ и пара uw образует дугу в дереве T_{v_1} . Чтобы убедиться в том, что в графе BG_d отсутствуют контуры, заметим, что в нем нет дуг между буферами с разными назначениями, а буферы с одним и тем же назначением v образуют дерево, изоморфное дереву T_v . Всякий путь $P \in \mathcal{P}$, оканчивающийся в узле v , является путем в дереве T_v , и по определению рассматриваемого буферного графа BG_d в нем существует такой путь, проходящий по буферам с назначением v , образом которого является путь P . Именно этот путь в графе BG_d и будет считаться гарантированным путем. Это означает, что для пакета p , адресованного узлу v и сформированного в узле u , должно выполняться равенство $fb(p) = b_u[v]$, и если его нужно отправить в узел w , то должно выполняться равенство $nb(p, b) = b_w[v]$.

Определение 5.8. Контроллер **dest** — это контроллер буферного графа $\mathbf{bgc}_{BG_d}$, в котором функции fb и nb определены так, как в предыдущем абзаце.

Теорема 5.9. Для произвольной связной сети существует беступиковый контроллер, который использует N буферов в каждом узле и позволяет перемещать пакеты по произвольным заранее выбранным входным деревьям.

Доказательство. Как видно из определения контроллера **dest**, он действительно является беступиковым и использует указанное число буферов. $\square$

Как было упомянуто выше, требование маршрутизации по входным деревьям может быть ослаблено: можно потребовать, чтобы пакеты, адресованные одному и тому же узлу, отправлялись по каналам, которые образуют ациклический граф. Однако, как показывает пример, изображенный на рис. 5.2, недостаточно того, чтобы в множестве $\mathcal{P}$ содержались только простые пути. Здесь пакеты из узла u_1 в узел v следуют по простому пути $\langle u_1, w_1, u_2, \dots, v \rangle$, а пакеты из узла u_2 в узел v следуют по простому пути $\langle u_2, w_2, u_1, \dots, v \rangle$. Каждый путь в множестве $\mathcal{P}$ является простым; совокупность всех каналов, используемых для доставки пакетов в узел v , содержит цикл $\langle u_1, w_1, u_2, w_2, u_1 \rangle$. Далее см. упражнение 5.2.

Контроллер **dest** очень прост в использовании, но имеет недостаток, который состоит в том, что в каждом узле требуется заводить слишком много буферов, а именно столько, сколько имеется узлов в сети. Можно определить двойствен-

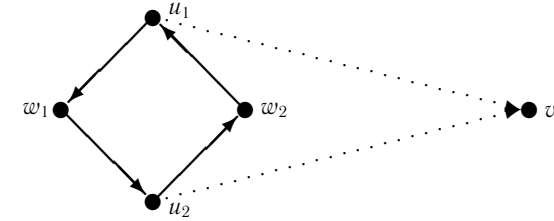


Рис. 5.2. Маршрутизация, недопустимая для контроллера **dest**

ную схему *источников*, в которой каждый буфер соответствует вершине v_i и используется для хранения пакетов, *сформированных* в узле v_i .

Схема пройденных шагов. В схеме пройденных шагов каждый узел u имеет $k + 1$ буферов $b_u[0], \dots, b_u[k]$. Предполагается, что в каждом пакете имеется *счетчик шагов*, регистрирующий, сколько раз продвинулся пакет по каналам связи, после того как покинул вершину-источник.

Пусть задан буферный граф $BG_h = (B, \vec{BE})$, в котором $b_u[i]b_w[j] \in \vec{BE}$ тогда и только тогда, когда $i + 1 = j$ и $uw \in E$. Чтобы убедиться в том, что в графе BG_h отсутствуют контуры, заметим, что по мере продвижения пакета индекс буферов, в которые он помещается, строго возрастает. Коль скоро каждый путь из множества $\mathcal{P}$ состоит не более чем из k звеньев, в буферном графе обязательно найдется соответствующий путь: если $P = u_0, \dots, u_l$, где $l \leq k$, — это путь в сети, то он является образом пути

$$b_{u_0}[0], b_{u_1}[1], \dots, b_{u_l}[l]$$

в графе BG_h . Этот гарантированный путь определяется следующими соотношениями: $fb(p) = b_u[0]$ для пакета p , сформированного в узле u , и $nb(p, b_u[i]) = b_w[i + 1]$ для пакета, который должен продвинуться из узла u в узел w .

Определение 5.10. Контроллер **hsf** — это контроллер $\mathbf{bgc}_{BG_h}$, в котором функции fb и nb определены так, как указано в предыдущем абзаце.

Теорема 5.11. Для всякой связной сети диаметра D существует беступиковый контроллер, который использует $D + 1$ буферов в каждом узле и требует продвижения пакетов по путям с наименьшим числом звеньев.

Доказательство. Для путей с наименьшим числом звеньев мы имеем $k = D$. Тогда **hsf** действительно является беступиковым контроллером, которому требуется $D + 1$ буферов в каждом узле. (Можно ограничиться и меньшим числом буферов, если обмен сообщениями между удаленными узлами не проводится.) $\square$

В схеме пройденных шагов буферы с индексом i используются для хранения пакетов, которые к этому моменту продвинулись i раз по каналам связи. Можно построить и двойственную схему *предстоящих шагов*. В ней буферы с индек-

сом i используются для хранения пакетов, которым еще предстоит совершить i продвижений по каналам связи (см. упражнение 5.3).

5.2.2. Ориентации сети G

В этом параграфе мы изучим один метод построения более изощренных буферных графов, которым нужно всего лишь несколько буферов в каждом узле. Контроллер пройденных шагов помещает пакеты в буферы, индекс которых увеличивается с каждым перемещением пакета. Теперь мы сделаем так, что индекс буферов будет увеличиваться гораздо медленнее (и сэкономим, таким образом, общее количество буферов в каждом узле), за счет того что возрастание индекса (не путать с классом буфера!) не обязательно будет происходить на каждом шаге продвижения пакета. Чтобы не допустить появления циклов в буферном графе, каналы, по которым продвижение пакета происходит без увеличения буферного индекса, должны образовывать ациклический граф.

Определение 5.12. *Ациклической ориентацией* графа G называется ориентированный ациклический граф, который образуется за счет установления ориентации всех ребер графа G (см. рис. 5.3).

Последовательность ациклических ориентаций $G_1, \dots, G_B$ графа G называется *ациклическим ориентированным покрытием* размера B для совокупности путей $\mathcal{P}$, если любой путь $P \in \mathcal{P}$ можно представить в виде последовательного соединения B путей $P_1, \dots, P_B$, где каждый путь P_i проходит в ориентированном графе G_i .

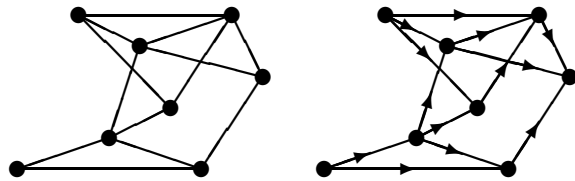


Рис. 5.3. Граф и его ациклическая ориентация

Если есть ациклическое ориентированное покрытие размера B , то можно построить такой контроллер, которому требуется иметь только B буферов в каждом узле. Пакет, сформированный в узле u , всегда помещается в буфер $b_u[1]$. Если пакет находится в буфере $b_u[i]$ узла u , то при продвижении в узел w он либо помещается в буфер $b_w[i]$ в том случае, когда в графе G_i дуга направлена из вершины u в вершину w , либо помещается в буфер $b_w[i+1]$ в том случае, когда в графе G_i дуга направлена из вершины w в вершину u .

Теорема 5.13. *Если множество путей $\mathcal{P}$ имеет ациклическое ориентированное покрытие размера B , то можно построить беступиковый контроллер, которому требуется иметь только B буферов в каждом узле.*

Доказательство. Пусть задано ациклическое ориентированное покрытие $G_1, \dots, G_B$ и пусть в узле u выделены буферы $b_u[1], \dots, b_u[B]$. Условимся использовать запись $uw \in \vec{E}_i$, если в графе G_i ребро uw направлено из вершины u в вершину w , и запись $wu \in \vec{E}_i$, если в графе G_i ребро uw направлено из вершины w в вершину u . Рассмотрим буферный граф $BG_a = (\mathcal{B}, \vec{BE})$, в котором включение $b_u[i]b_w[j] \in \vec{BE}$ имеет место тогда и только тогда, когда $uw \in E$ и при этом либо выполняется соотношение $(i = j \wedge uw \in \vec{E}_i)$, либо выполняется соотношение $(i + 1 = j \wedge wu \in \vec{E}_i)$. Чтобы убедиться в том, что этот граф не имеет контуров, заметим, что циклов, содержащих буферы с разными индексами, быть не может, поскольку согласно определению этого графа в нем нет дуг, ведущих из одного буфера в другой буфер с меньшим индексом. Не может быть также и циклов, образованных буферами одного и того же индекса i , потому что связь между этими буферами устроена в соответствии с ациклическим графом G_i .

Читателям предоставляется возможность самостоятельно показать (см. упражнение 5.4), что для любого пути $P \in \mathcal{P}$ есть гарантированный путь, образом которого является P , и этот путь определяется следующим образом:

$$\begin{aligned} fb(p) &= b_u[1], \\ nb(p, b_u[i]) &= \begin{cases} b_w[i], & \text{если } uw \in \vec{E}_i, \\ b_w[i+1], & \text{если } wu \in \vec{E}_i. \end{cases} \end{aligned}$$

Определив буферный граф, в котором каждому узлу отводится B буферов, мы приходим к выводу о том, что существует контроллер, которому требуется иметь только B буферов в каждом узле. Этим и завершается доказательство теоремы. $\square$

Контроллеры такого вида мы будем называть *контроллерами ациклического ориентированного покрытия* или коротко АОС-контроллерами. Теперь мы перейдем к демонстрации того, насколько легко можно построить контроллеры, воспользовавшись ациклическими ориентированными покрытиями.

Коммутация пакетов в кольце. При помощи ациклических ориентированных покрытий можно построить беступиковые контроллеры для различных классов сетей. Мы начнем с того, что покажем, как построить контроллер для колец, используя только три буфера в каждом узле. В приведенной ниже теореме считается, что весовые коэффициенты каналов симметричны, т. е. $\omega_{uw} = \omega_{wu}$.

Теорема 5.14. *Для кольцевой сети можно построить контроллер, которому требуется иметь только три буфера в каждом узле, чтобы провести маршрутизацию пакетов по кратчайшим путям.*

Доказательство. Согласно теореме 5.13 для этого достаточно подобрать ациклическое ориентированное покрытие размера три для совокупности путей, которая содержит для каждой пары узлов кратчайший путь между ними.

Введем следующие обозначения. Условимся, что для каждой пары узлов u и v запись $d_c(u, v)$ будет обозначать длину пути, направленного по часовой стрелке, из вершины u в вершину v , запись $d_a(u, v)$ — длину пути, направленного против часовой стрелки, из вершины u в вершину v (заметим, что $d_c(v, u) = d_a(u, v)$), а запись $d(u, v)$ — минимальную из двух величин $\min(d_c(u, v) \text{ и } d_a(u, v))$. Сумма C весовых коэффициентов всех каналов называется *длиной кольца*. Очевидно, что для любой пары узлов u, v верно равенство $d_c(u, v) + d_a(u, v) = C$, и поэтому $d(u, v) \leq C/2$.

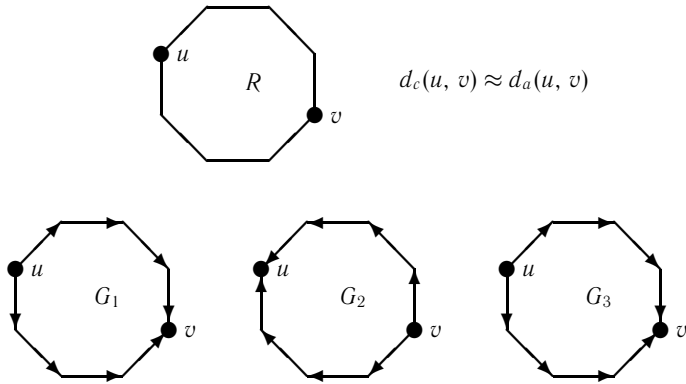


Рис. 5.4. Ациклическое ориентированное покрытие для кольца

Вначале рассмотрим простой случай, когда есть пара таких узлов u и v , что $d(u, v) = C/2$. Ориентированные графы G_1 и G_3 получены путем ориентации всех ребер по направлению к вершине v , а ориентированный граф G_2 получен путем ориентации всех ребер по направлению к вершине u (см. рис. 5.4).

Кратчайший путь из узла u в узел v содержится в графе G_1 или G_3 , а кратчайший путь из v в u содержится в графе G_2 . Рассмотрим теперь другую пару узлов x, y , отличную от пары u, v . Тогда, ввиду того что $d(x, y) \leq C/2$, существует кратчайший путь P из узла x в узел y , который не содержит либо вершину u , либо вершину v . Если P не содержит ни u , ни v , то этот путь целиком содержится либо в графе G_1 , либо в графе G_2 . Если P проходит через v , то он образуется последовательным соединением пути в графе G_1 и пути в графе G_2 . Если P проходит через u , то он образуется последовательным соединением пути в графе G_2 и пути в графе G_3 .

Если нет такой пары узлов u, v , для которой верно равенство $d(u, v) = C/2$, то выберем такую пару узлов, для которой величина $d(u, v)$ наиболее близка к $C/2$. Нетрудно показать, что если бы нашлась пара узлов x, y , кратчайший путь между которыми было бы нельзя представить в виде последовательного соединения путей в указанных ориентированных графах, то величина $d(x, y)$ была бы ближе к $C/2$, нежели $d(u, v)$. $\square$

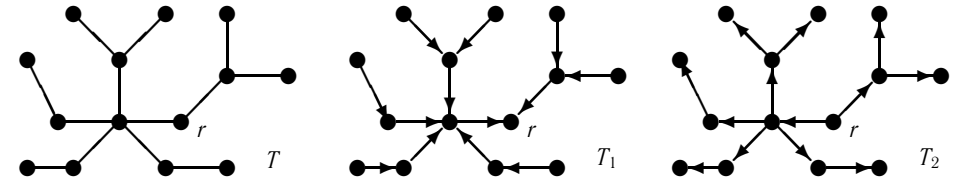


Рис. 5.5. Ациклическое ориентированное покрытие для дерева

Коммутация пакетов в дереве. Для древесной сети можно построить контроллер, которому требуется иметь только два буфера в каждом узле.

Теорема 5.15. Для древесной сети существует контроллер, которому требуется иметь только два буфера в каждом узле.

Доказательство. Согласно теореме 5.13 достаточно подобрать ациклическую ориентацию для дерева, которая сможет покрыть все простые пути. Выберем произвольную вершину r и рассмотрим два дерева: T_1 , в котором все дуги направлены к вершине r , и T_2 , в котором все дуги направлены от вершины r (см. рис. 5.5). Всякий простой путь из вершины u в вершину v представляет собой последовательное соединение пути из u в вершину, являющуюся наименьшим общим предком вершин u и v в дереве T_1 , и пути из этого наименьшего общего предка в вершину v в дереве T_2 . $\square$

Поскольку описанную здесь схему можно применить к остовному дереву наименьшей глубины, построенному для произвольной сети, мы фактически установили, что для любой сети возможна беступиковая маршрутизация, требующая 2 буфера в каждом узле. Но пути, которые при этом будут задействованы, в общем случае могут оказаться неоптимальными.

Применение АОС-контроллеров. Как показывает нам пример с кольцевыми сетями, для применения АОС-контроллеров могут потребоваться весьма специальные пути, по которым должны распространяться сообщения. В связи с этим возникает ряд вопросов, исследованию которых был посвящен ряд работ последних лет.

Можно ли, используя минимальное число буферов, переправлять пакеты по оптимальным путям? Если это невозможно, то какова взаимосвязь между буферной сложностью и эффективностью маршрутизации? Штефанкович в работе [179] показал, что АОС-контроллеры *не столь уж экономны*. Ему удалось построить класс сетей, для которых существует беступиковая оптимальная маршрутизация, требующая фиксированного постоянного числа буферов в каждом узле, но при этом наилучшему оптимальному АОС-контроллеру требуется иметь $\Omega(\lg N / \lg \lg N)$ буферов в каждом узле. До сих пор неизвестно, насколько велик разрыв между наилучшим АОС-контроллером и наилучшим контроллером общего вида.

Можно ли задавать пути, которые используются АОС-контроллерами, при помощи компактных схем маршрутизации, наподобие интервальной маршрутизации? Для гиперкуба удалось установить, что 1) существует оптимальная схема интервальной разметки, 2) существует контроллер, которому для использования оптимальных путей требуется иметь 2 буфера в каждом узле, но при этом 3) *нет таких* АОС-контроллеров, которые можно было бы так встроить в схему интервальной разметки, чтобы они могли ограничиться использованием только двух буферов в каждом узле. Исследования в этом направлении еще продолжаются.

5.3. Неструктурированные решения

Теперь мы обратимся к классу контроллеров, предложенных Туэгом и Ульманом в работе [196]. Эти контроллеры не выдают предписаний, в какой буфер следует поместить тот или иной пакет, и используют только простую локальную информацию, наподобие счетчика шагов или числа буферов, занятых в узле.

5.3.1. Контроллеры с прямым и обратным отсчетом

Контроллер с прямым отсчетом. Условимся, что для всякого пакета p запись s_p будет обозначать число продвижений, которые необходимо совершить пакету, чтобы достичь адресата; естественно, что $0 \leq s_p \leq k$. Бывает так, что значение s_p не нужно передавать вместе с пакетом, поскольку многие алгоритмы вычисляют эти данные в каждом узле (см., например, алгоритм Netchange, описанный в § 4.6.3). Условимся также, что для каждого узла u запись f_u будет обозначать количество свободных буферов узла u . Естественно, что всегда выполняется неравенство $0 \leq f_u \leq B$.

Определение 5.16. *Контроллер с прямым отсчетом* **FC** разрешает узлу u принять пакет p в том и только том случае, когда выполняется неравенство $s_p < f_u$.

Контроллер разрешает прием пакета, если в узле имеется больше свободных буферов, чем число продвижений, которые необходимо совершить пакету.

Теорема 5.17. *Если $B > k$, то контроллер **FC** является беступиковым.*

Доказательство. Чтобы убедиться в том, что пакет разрешается сформировать в пустом узле, обратим внимание на то, что если все буферы узла u пусты, то $f_u = B$. Новому пакету предстоит совершить не более k продвижений, и, вследствие того что $B > k$, этот пакет допускается в узел u .

Беступиковость контроллера **FC** будет доказана от противного; Допустим, что γ — это достижимая тупиковая конфигурация контроллера. Теперь рассмотрим конфигурацию δ , которая получается из конфигурации γ путем применения максимальной последовательности действий продвижения и поглощения пакетов. В конфигурации δ не может быть продвинут ни один пакет, но, ввиду того что конфигурация γ является предположительно тупиковой, в конфигурации δ остается хотя бы один пакет. Рассмотрим пакет p , который в конфигурации δ расположен

ближе всего к своему узлу-адресату, т. е. пакет, имеющий наименьшее значение счетчика s_p среди всех пакетов, оставшихся в конфигурации δ .

Обратимся к узлу u , в котором расположен пакет p . Так как узел u не является адресатом пакета p (иначе он был бы поглощен в конфигурации δ), существует такой сосед w узла u , которому должен быть передан пакет p . Коль скоро **FC** не разрешает выполнить это действие, имеет место неравенство

$$s_p - 1 \geq f_w.$$

Поскольку $s_p \leq k$ и при этом, согласно сделанному предположению, $k < B$, отсюда следует неравенство $f_w < B$, которое означает, что в конфигурации δ хотя бы один пакет расположен в вершине w . Из всех пакетов, размещенных в узле w , выберем тот пакет q , который прибыл в узел w самым последним, и рассмотрим число свободных буферов f'_w в узле w непосредственно перед тем, как пакету q было разрешено переместиться в узел w . Ввиду того что пакет q занимает один из этих f'_w буферов, и коль скоро, (в силу выбора этого пакета) все пакеты, попавшие в узел w после q , впоследствии были удалены из узла w , должно выполняться неравенство $f'_w \leq f_w + 1$.

Так как пакету q было разрешено переместиться в узел w , справедливо неравенство $s_q < f'_w$. Сопоставив все четыре полученных нами неравенства, приходим к соотношению

$$s_q < f'_w \leq f_w + 1 \leq s_p,$$

которое противоречит выбору p . $\square$

Контроллер с обратным отсчетом. Чтобы построить контроллер, «двойственный» **FC**, разрешение на прием пакета нужно выдавать на основании числа шагов, пройденных этим пакетом. Условимся, что для всякого пакета p запись t_p будет обозначать число продвижений, которые уже удалось совершить пакету. Естественно, что всегда верно неравенство $0 \leq t_p \leq k$.

Определение 5.18. *Контроллер с обратным отсчетом* **BC** разрешает узлу u принять пакет p в том и только том случае, когда выполняется неравенство $t_p > k - f_u$.

Доказательство утверждения о том, что контроллер **BC** является беступиковым (упражнение 5.6), аналогично доказательству теоремы 5.17.

5.3.2. Контроллеры с состояниями прямого и обратного отсчета

Воспользовавшись более подробной информацией о пакетах, хранящихся в узле, можно построить контроллер, который функционирует почти так же, как и контроллер с прямым отсчетом, но разрешает при этом совершать больше действий.

Контроллер с состояниями прямого отсчета. Запись s_p будет обозначать то же самое, что и в предыдущем параграфе. Для узла u определим (как функцию

состояния процесса u) целочисленный вектор состояния $\langle j_0, \dots, j_k \rangle$, где j_s — количество пакетов p , размещенных в узле u и удовлетворяющих условию $s_p = s$.

Определение 5.19. Контроллер с состояниями прямого отсчета **FS** разрешает прием пакета p в узле u , имеющем вектор состояния $\langle j_0, \dots, j_k \rangle$, тогда и только тогда, когда выполняется соотношение

$$\forall i, 0 \leq i \leq s_p: i < B - \sum_{s=i}^k j_s.$$

Теорема 5.20. Если $B > k$, то контроллер **FS** является беступиковым.

Доказательство. Читателю предоставляется возможность самостоятельно убедиться в том, что пустому узлу разрешается принять любой пакет. Допустим, что существует достижимая тупиковая конфигурация γ . Теперь рассмотрим конфигурацию δ , которая получается из конфигурации γ путем применения максимальной последовательности действий продвижения и поглощения пакетов. В конфигурации δ есть хотя бы один пакет, но при этом ни один пакет не может быть перемещен. Выберем пакет p с наименьшим значением характеристики s_p ; предположим, что пакет p размещен в узле u и должен быть продвинут в узел w . Рассмотрим вектор состояний $\langle j_0, \dots, j_k \rangle$ узла w в конфигурации δ .

Если в узле w нет пакетов, то $\sum_{s=0}^k j_s = 0$, и отсюда следует, что узлу w разрешается принять пакет p , что противоречит выбору конфигурации δ . Значит, в узле w есть хотя бы один пакет. Из всех пакетов, размещенных в узле w , рассмотрим тот пакет q , который находится ближе всего к своему узлу-адресату, т.е. удовлетворяет условию $s_q = \min\{s: j_s > 0\}$. Покажем, что тогда должно выполняться неравенство $s_q < s_p$, противоречащее выбору пакета p .

Из всех пакетов, размещенных в узле w , обратим внимание на тот пакет r , которому в последнюю очередь разрешили попасть в узел w ; для этого пакета, естественно, выполняется неравенство $s_q \leq s_r$. Рассмотрим вектор состояний $\langle j'_0, \dots, j'_k \rangle$ узла w непосредственно перед приемом пакета r . Так как пакету r было разрешено попасть в узел w , это означает, что было соблюдено условие

$$\forall i, 0 \leq i \leq s_r: i < B - \sum_{s=i}^k j'_s.$$

После приема пакета r некоторые пакеты покинули узел w , и, согласно выбору r , среди них были все те пакеты, которым было разрешено попасть в узел w , после того как туда поступил пакет r . Отсюда следуют соотношения

$$\begin{aligned} j_s &\leq j'_s && \text{для } s \neq s_r, \\ j_{s_r} &\leq j'_{s_r} + 1. \end{aligned}$$

Но это означает также, что должны выполняться неравенства

$$\forall i, 0 \leq i \leq s_r: i < B - \sum_{s=i}^k j_s + 1.$$

Таким образом, для $i = s_q$ должно быть верно неравенство

$$s_q \leq B - \sum_{s=s_q}^k j_s.$$

Теперь мы воспользуемся тем обстоятельством, что пакету p не было разрешено продвинуться в узел w , т.е.

$$\exists i_0, 0 \leq i_0 \leq s_p - 1: i_0 \geq B - \sum_{s=i_0}^k j_s.$$

Это приводит к неравенству

$$\begin{aligned} s_p &> s_p - 1 \\ &\geq i_0 && \text{см. выше} \\ &\geq B - \sum_{s=i_0}^k j_s && \text{см. выше} \\ &\geq B - \sum_{s=0}^k j_s && \text{ввиду того, что } j_s \geq 0 \text{ (и } i_0 \geq 0) \\ &\geq B - \sum_{s=s_q}^k j_s && \text{ввиду того, что } j_s = 0 \text{ для } s < s_q \\ &\geq s_q, \end{aligned}$$

которое и составляет ожидаемое противоречие. $\square$

Контроллер с состояниями обратного отсчета. Существует также и контроллер, «двойственный» контроллеру с состояниями прямого отсчета, в котором учитывается более подробная информация, нежели в контроллере с обратным отсчетом, и за счет этого разрешается большее количество действий. Будем считать, что запись t_p используется в том же качестве, что и ранее. Определим вектор состояния $\langle i_0, \dots, i_k \rangle$, где i_t равно числу пакетов в узле u , которые уже совершили в точности t продвижений.

Определение 5.21. Контроллер с состояниями обратного отсчета **BS** разрешает прием пакета p в узле u , имеющем вектор состояния $\langle i_0, \dots, i_k \rangle$, тогда и только тогда, когда выполняется соотношение

$$\forall j, t_p \leq j \leq k: j > \sum_{t=0}^j i_t - B + k.$$

Доказательство утверждения о том, что контроллер **BS** является беступиковым, аналогично доказательству теоремы 5.20.

Сравнительный анализ контроллеров. Контроллер с состояниями прямого отсчета более либерален, чем контроллер с прямым отсчетом, поскольку позволяет совершать большее количество действий.

Лемма 5.22. Каждое действие, разрешенное контроллером **FC**, также разрешено контроллером **FS**.

Доказательство. Предположим, что контроллер **FC** разрешает узлу u принять пакет p . Тогда должно быть выполнено неравенство $s_p < f_u = B - \sum_{s=0}^k j_s$, и поэтому для всех $i \leq s_p$ верно неравенство $i < B - \sum_{s=i}^k j_s$, откуда следует, что контроллер **FS** разрешает совершить указанное действие. $\square$

В работе [196] было показано, что контроллер **FC** более либерален, чем контроллер **BC**, контроллер **FS** более либерален, чем контроллер **BS**, а контроллер **BS** более либерален, чем контроллер **BC**. Кроме того, было установлено, что каждый из этих четырех контроллеров является наиболее либеральным среди всех контроллеров, которые используют ту же самую информацию.

5.4. Прочие вопросы

Как видно из результатов, представленных в этой главе, важная роль в них отводится количеству буферов, которые требуются контроллеру. Обычно пропускная способность сети возрастает с увеличением числа доступных буферов. Для неструктурированных решений дается только нижняя оценка числа буферов; для использования большего числа буферов контроллеры не нужно модифицировать. А вот в структурированных решениях дополнительные буферы нужно суметь каким-то образом ввести в буферный граф, и это можно сделать либо *статически*, либо *динамически*. Если это сделать статически, то каждый буфер будет иметь заданное положение в буферном графе, т. е. буферный граф строится более «просторно», нежели это описано в примере из § 5.5.2. Обычно вместо одиночного буфера $fb(p)$ или $nb(p, b)$ сразу несколько буферов назначаются в качестве возможных точек начала или продолжения пути в буферном графе. Если введение дополнительных буферов проводится динамически, то вначале при построении буферного графа используется как можно меньше буферов; эти буферы в графе мы будем называть *логическими* буферами. При выполнении операций каждый реальный буфер (который мы будем называть *физическим* буфером) может быть использован в качестве любого логического буфера, но при этом для каждого логического буфера должен быть хотя бы один физический буфер, которому отводится роль этого логического буфера. В такой схеме для предотвращения блокировки необходимо отвести место только для небольшого числа буферов, а остальные буферы можно использовать свободно, гибко варьируя их назначение.

В этой главе мы исходили из предположения о том, что пакеты имеют один и тот же размер и поэтому буферы имеют одинаковый размер и могут вмещать в точности один пакет. Задачу можно исследовать и в рамках предположения о том, что размер пакетов может варьироваться. Для этого случая Бодлаендер в работе [30] адаптировал те решения, которые мы рассматривали в § 5.5.3.

5.4.1. Изменения топологии

До сих пор мы не принимали во внимание возможность того, что по мере продвижения пакетов от источника к адресату топология сети может измениться. После того как происходит такое изменение, таблицы маршрутизации в каждом узле уточняются, и после этого продвижение пакетов осуществляется на основе уточненных таблиц. В результате такой модификации таблиц пакет может последовать таким путем в сети, который никогда не был бы избран, если бы изменения не случились; может случиться даже так, что весь путь пакета в целом будет содержать циклы.

Влияние, которые оказывают приведенные соображения на методы предотвращения блокировки, изученные в этой главе, на первый взгляд противоречат нашей интуиции. Контроллер **dest**, корректность которого опирается на предположение о том, что в множестве $\mathcal{P}$ содержатся только простые пути, можно продолжать использовать без всяких модификаций. А вот контроллеры, зависящие только от верхней оценки числа звеньев в путях по которым следуют пакеты, требуют серьезного внимания.

Контроллер dest. Спустя какое-то определенное время, после того как случится последнее топологическое изменение, таблицы маршрутизации должны стабилизироваться и стать ациклическими таблицами. И даже в том случае, если *во время* вычисления таблиц образуется ситуация циклического простоя, как только вычисление таблиц завершится, буферный граф снова станет ациклическим, и все пакеты окажутся помещенными в подходящие буферы. Следовательно, едва лишь таблицы маршрутизации установятся, образовавшаяся в результате конфигурация не будет содержать заблокированных пакетов.

Контроллеры пройденных шагов. Рассмотрим контроллер, который зависит от предположения о том, что всякий пакет следует путем, состоящим самое большее из k звеньев. Можно выбрать достаточно большое значение k , чтобы можно было гарантировать с *высокой вероятностью*, что каждый пакет достигает своего адресата, совершив не более k шагов, даже в том случае, когда по ходу его пересылки от источника к адресату случаются топологические изменения. Для всех пакетов, которые достигают своего адресата за k шагов, контроллер пройденных шагов, а также контроллеры с прямым и обратным отсчетом можно использовать без всяких ограничений. Однако может случиться так, что пакет не достигает своего адресата за k шагов в связи с тем, что изменения в топологии сети и соответствующие уточнения таблиц произошли неудачно. Если это так, то пакет будет помещен в неподходящий буфер, и он будет навсегда заблокирован в узле, не дойдя до адресата.

Ситуации такого рода могут быть разрешены только во взаимодействии с протоколами более высокого уровня в иерархии протоколов. Простейшее решение предусматривает отмену пакета. С точки зрения протокола сквозной передачи данных пакет теперь считается потерянным, но с этой потерей можно справиться при помощи протокола сеансового уровня, как было показано в § 3.3.2.

Отмена пакетов необходима и для того, чтобы реализовать допущение, которое было сделано в § 3.3.2, о том, что всякий пакет имеет максимальный срок жизни μ . Если для продвижения пакета требуется время μ_0 , ограничение времени жизни пакета в сквозной передаче данных величиной μ приводит к тому, что максимальное число шагов, которые может совершить пакет, ограничено числом $k = \mu/\mu_0$.

5.4.2. Другие типы блокировок

В этой главе рассматривались только блокировки хранения-продвижения. Если верны те допущения, которые были сделаны в § 5.5.1, блокировка хранения-продвижения — это единственно возможный тип блокировки, который может случиться. Однако в сетях, которые применяются на практике, эти допущения справедливы не всегда, и, как было отмечено в работе Мерлина и Швейцера [149], это приводит к тому, что возникают другие типы блокировок. Мерлин и Швейцер выделили четыре типа блокировок, а именно, потомственную блокировку, блокировку высвобождения копий, блокировку темпа и блокировку сборки пакетов, и показали, как предотвратить блокировки этих типов, распространив на них метод буферных графов.

Потомственная блокировка возникает, когда пакет p может порождать в сети другой пакет q , например отчет об отказе, который направляется в узел, являющийся источником исходного пакета, если ему пришлось столкнуться с неисправностью канала. В связи с этим возникает причинно-следственная связь между формированием нового пакета (q) и продвижением или поглощением уже существующего пакета (p), и тем самым нарушается допущение, сделанное в § 5.5.1, которое гласит, что сеть всегда разрешает продвижение и поглощение пакета.

Для предотвращения потомственной блокировки нужно завести две копии буферного графа, один из которых будет использоваться для основных сообщений, а другой для производных сообщений (потомков). Если потомки могут в свою очередь порождать новое потомство, нужно использовать несколько уровней буферных графов.

Блокировка высвобождения копий возникает, когда узел-источник сохраняет копию пакета, до тех пор пока от узла-адресата не поступит подтверждение о получении пакета. (Вспомните, как устроен протокол передачи сообщений с таймерами, описанный в § 3.3.2, и представьте себе, что последовательность слов in_p размещена в той же самой области памяти, которая используется механизмом маршрутизации для временного хранения пакетов.) Вследствие этого нарушается допущение о том, что после продвижения пакета занятый им буфер освобождается (см. § 5.5.1).

Для предотвращения блокировки высвобождения копий можно предложить две модификации метода буферных графов. В первом случае мы опираемся на предположение о том, что блокировка высвобождения копий всегда возникает за счет того, что образуется циклический простой основных сообщений, а также

подтверждающих сообщений. Решение состоит в том, чтобы рассматривать подтверждения как производные сообщения (потомки) и помещать их в отдельные копии буферного графа. Второй вариант решения в большинстве случаев требует меньшего числа буферов. Суть его состоит в том, что вновь порожденные пакеты помещаются в специальные отведенные для них *ключевые буферы*, в которые запрещается помещать переправляемые пакеты.

Блокировка темпа возникает, когда в сети есть узлы, у которых внутренняя память ограничена, и поэтому они не могут поглощать сообщения, до тех пор пока не будут сформированы какие-то другие сообщения. Например, терминал телетайпа должен сначала выдать некоторый символ, а уже потом принять следующий символ для печати. Вследствие этого нарушается наше допущение о том, что пакет, который достиг узла-адресата, может быть всегда поглощен.

Чтобы предотвратить блокировку темпа, нужно проводить различие между продвигающимися пакетами и откликами на продвижение, имея в виду, что поглощение пакетов первого типа может произойти только после того, как будут сформированы пакеты второго типа. Для сообщений этих двух типов используются разные копии буферного графа.

Блокировка сборки пакета может возникнуть в сетях, в которых для доставки больших сообщений приходится разбивать их на маленькие порции и передавать по частям; при этом ни один из маленьких пакетов нельзя изъять из сети до тех пор, пока все части большого сообщения не достигнут своего адресата. (Вспомните, как устроен протокол раздвижного окна, описанный в § 3.3.1, в котором слова удаляются из массива out_p только после того, как будут получены все слова с меньшим значением индекса.) Вследствие этого нарушается наше предположение о том, что пакет, который достиг узла-адресата, может быть всегда поглощен.

Блокировку сборки пакета можно предотвратить, если использовать отдельные группы буферов для продвигающихся пакетов и пакетов, которые участвуют в сборке.

5.4.3. Активный тупик

Как следует из определения заблокированных пакетов (определение 5.2), для каждого пакета, находящегося под управлением беступикового контроллера, есть хотя бы одно вычисление, в котором этот пакет будет поглощен. Но так как число различных допустимых вычислений, вообще говоря, очень велико, это отнюдь не подразумевает того, что каждый пакет рано или поздно будет доставлен по назначению даже в том случае, когда вычисление может продолжаться сколько угодно долго. Достаточно взглянуть на пример, изображенный на рис. 5.6. Предположим, что в узле u берет начало бесконечный поток пакетов, которые нужно отправить в узел v , и всякий раз, когда буфер в узле w освобождается, очередному пакету из узла u разрешается совершить продвижение. В узле s имеется пакет, адресованный узлу t , и этот пакет не заблокирован, потому что всякий раз, когда

буфер в узле w освобождается, существует такое *возможное* продолжение вычисления, по ходу которого этому самому пакету будет разрешено продвинуться в узел w и затем достичь узла t . Хотя такое продолжение вычисления является возможным, оно не является неизбежным, и пакет может навеки остаться в узле. Ситуацию такого рода принято называть *активным тупиком*.

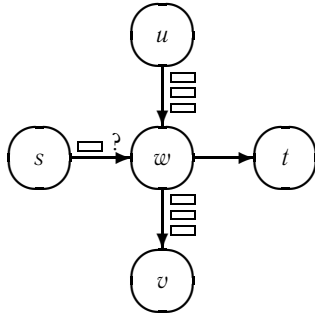


Рис. 5.6. Пример активного тупика

Возможности тех контроллеров, которые мы изучили в этой главе, можно так расширить, чтобы они могли предотвращать также и активные тупики.

Определение 5.23. Пусть задана некоторая сеть, контроллер **соп** и конфигурация γ . Тогда пакет p считается *оставленным без внимания*, если существует бесконечная последовательность действий, применимых к конфигурации γ , при выполнении которых пакет p не будет поглощен. Конфигурация γ называется *активной тупиковой конфигурацией*, если в этой конфигурации есть хоть один пакет, оставленный без внимания. Говорят, что контроллер *избегает активных тупиков*, из конфигурации, в которой нет никаких пакетов, нельзя достичь ни одной активной тупиковой конфигурации.

Далее в этом параграфе мы докажем, что контроллеры с буферным графом избегают активных тупиков. В самом конце мы вкратце обсудим, как поступить с неструктурированными решениями.

Контроллер с буферным графом. Можно показать, что все контроллеры, описанные в § 5.5.2, избегают активных тупиков без всякой модификации их действий, если на бесконечных вычислениях соблюдаются допущения строгой справедливости F1 и F2, а также допущение слабой справедливости F3.

F1. Если непрерывно предпринимаются попытки сформировать пакет p , то каждое бесконечное вычисление, в котором буфер $fb(p)$ оказывается свободным в бесконечном множестве конфигураций, обязательно содержит пакет p .

F2. Если в конфигурации γ пакет p должен быть отправлен из узла u в узел w , то в каждом бесконечном вычислении, начинающемся конфигурацией γ и име-

ющем пустой буфер $nb(p, b)$ в бесконечном множестве конфигураций, происходит продвижение пакета p .

F3. Если пакет p в конфигурации γ достиг своего узла-адресата, то в каждом бесконечном вычислении, начинающемся конфигурацией γ , происходит поглощение пакета p .

Лемма 5.24. Если соблюдаются допущения справедливости F2 и F3, то в любом бесконечном вычислении под управлением контроллера **bgc** каждый буфер оказывается свободным бесконечно часто.

Доказательство. Доказательство проводится с использованием нисходящей индукции по классам буферов. Как и в доказательстве теоремы 5.7, обозначим символом R наибольший класс буферов. Напоминаем, что в конфигурациях, которые можно достичь под управлением контроллера **bgc**, все пакеты помещаются в подходящие буферы.

Случай $r = R$. Буфер класса R не имеет исходящих дуг в буферном графе, и поэтому пакет, помещенный в такой буфер, уже достиг своего узла-адресата. Следовательно, согласно допущению F3 после каждой конфигурации, в которой этот буфер занят, рано или поздно будет достигнута конфигурация, в которой этот буфер пуст. Это означает, что буфер свободен бесконечно часто.

Случай $r < R$. Рассмотрим конфигурацию γ , в которой буфер b класса $r < R$ занят пакетом p . Если p уже достиг предназначенного узла, то согласно допущению F3 позднее будет достигнута конфигурация, в которой буфер b освободится. В противном случае пакет p должен быть переправлен в буфер $nb(p, b)$, который имеет буферный класс $r' > r$. По индуктивному предположению в каждом бесконечном вычислении, которое начинается конфигурацией γ , этот буфер освобождается бесконечно часто. В силу допущения F2 отсюда следует, что пакет p будет продвигаться бесконечно часто, и, значит, после конфигурации γ будет достигнута конфигурация, в которой буфер b будет пустым. $\square$

Теорема 5.25. Если соблюдаются допущения справедливости F1, F2 и F3, то в любом бесконечном вычислении под управлением контроллера **bgc** каждый пакет, который может возникнуть в сети, будет поглощен в соответствующем узле-адресате.

Доказательство. По лемме 5.24 все буферы оказываются пустыми в бесконечно многих конфигурациях. Согласно допущению F1 это означает, что каждый пакет, который пытаются ввести в сеть бесконечно часто, рано или поздно будет сформирован. Согласно допущению F2 он будет продвигаться, до тех пор пока не достигнет своего узла-адресата. Согласно допущению F3 он будет поглощен в узле-адресате. $\square$

Можно предполагать, что механизм недетерминированного выбора в распределенных системах просто обеспечивает выполнимость трех упомянутых допущений. В противном случае соблюдение этих допущений может быть *навязано* путем встраивания в контроллер специальной процедуры, которая в случае освобождения буфера будет отдавать приоритет тем пакетам, которые имеют более продолжительный возраст.

Неструктурированные решения. Чтобы контроллеры, описанные в § 5.5.3, могли избегать активных тупиков, их нужно модифицировать. В этом можно убедиться, построив бесконечное вычисление, в котором непрерывный поток пакетов может привести к тому, что контроллер запретит продвижение какого-то другого пакета. Туэг в работе [195] привел пример такого вычисления для контроллера **FC** и представил такую модификацию контроллера **FS** (аналогичную рассмотренной нами модификации контроллеров с буферным графом), которая позволяет избегать активных тупиков.

5.5. Упражнения к главе 5

5.5.1

Упражнение 5.1. Покажите, что не существует беступиковых контроллеров, которые используют только по одному буферу в каждом узле и разрешают каждому узлу отправлять пакет в какой-то другой узел.

5.5.2

Упражнение 5.2. Покажите, что контроллер **dest** не будет являться беступиковым, если маршрутизация пакетов проводится так, как показано на рис. 5.2.

Упражнение 5.3. (Схема предстоящих шагов) Опишите устройство буферного графа, а также функции fb и nb для контроллера, использующего буфер $b_u[i]$ для хранения пакетов, которым предстоит совершить i продвижений, чтобы достичь своих узлов-адресатов.

Как определить буферный класс для $b_u[i]$? Должен ли каждый пакет вести учет сделанных шагов?

Упражнение 5.4. Завершите доказательство утверждения о том, что граф BG_a (определенный в доказательстве теоремы 5.1.3) действительно является буферным графом, т. е. для каждого пути $P \in \mathcal{P}$ существует гарантированный путь, образом которого является P . Покажите, что функции fb и nb действительно, как утверждается в доказательстве теоремы, описывают путь в буферном графе BG_a .

Курсовая работа 5.5. Докажите, что существует такой беступиковый контроллер, который осуществляет коммутацию пакетов в гиперкубе, используя при этом только два буфера в каждом узле и разрешая проводить маршрутизацию пакетов по путям с наименьшим числом звеньев.

Можно ли получить совокупность используемых путей при помощи интервального алгоритма маршрутизации (см. § 4.4.2)? Можно ли воспользоваться для этого схемами разметки линейными интервалами?

5.5.3

Упражнение 5.6. Докажите, что контроллеры **BC** и **BS** являются беступиковыми.

Упражнение 5.7. Докажите, что всякое действие, разрешенное контроллером **BC**, будет также разрешено контроллером **FC**.

ГЛАВА 6

ВОЛНОВЫЕ АЛГОРИТМЫ И АЛГОРИТМЫ ОБХОДА

При проектировании распределенных алгоритмов для разных приложений некоторые общие проблемы, относящиеся к сетевым процессам, часто становятся промежуточными задачами. К таким элементарным задачам относятся широко-вещательное распространение информации (например, сообщения о начале или завершении вычисления), установление глобальной синхронизации между процессами, запуск выполнения некоторого действия в каждом процессе, вычисление функции при условии, что каждый из процессов содержит только часть входных данных. Эти задачи всегда решаются путем обмена сообщениями согласно некоторой предписанной схеме, которая зависит от топологии сети и позволяет задействовать все процессы. И действительно, как будет видно читателю в последующих главах, эти проблемы являются настолько основополагающими, что решения более сложных задач, таких как избрание лидера (гл. 7), определение завершения вычисления (гл. 8), обеспечение взаимно исключенного доступа, можно представить так, чтобы взаимосвязь между процессами проводилась только в соответствии с указанными схемами обмена сообщениями.

Схемы обмена сообщениями, которые в дальнейшем будем называть волновыми алгоритмами, настолько важны, что мы исследуем их отдельно, независимо от какого-либо конкретного прикладного алгоритма, в который встроены эти схемы. В этой главе приводится формальное определение волновых алгоритмов (§ 6.1.1) и обосновываются некоторые общие результаты, относящиеся к этому понятию (§ 6.1.2). Мы дадим строгую формулировку замечания о том, что одни и те же алгоритмы можно использовать во всех перечисленных выше основополагающих задачах, к числу которых относится широко-вещательное распространение сообщений, синхронизация и вычисление глобальных функций (§§ 6.1.3 – 6.1.5). В § 6.6.2 приведены некоторые наиболее употребительные волновые алгоритмы. В § 6.6.3 рассматриваются алгоритмы обхода, которые являются не чем иным, как волновыми алгоритмами, обладающими тем дополнительным свойством, что все события в каждом вычислении алгоритма линейно упорядочены по отношению причинно-следственной зависимости. В § 6.6.4 приведен ряд алгоритмов для распределенного поиска в глубину.

Рассматривать волновые алгоритмы в виде отдельной темы, несмотря на то что они обычно применяются в качестве процедур, входящих в состав более сложных алгоритмов, имеет смысл по двум причинам. Во-первых, введение этого понятия облегчает последующее изучение более сложных алгоритмов, поскольку свойства процедур, используемых в этих алгоритмах, будут уже усвоены. Во-вторых, некоторые проблемы распределенных вычислений могут быть решены

при помощи конструкций общего вида, которые дают требуемый алгоритм решения, стоит лишь выбрать в качестве их параметра нужный волновой алгоритм. Одну и ту же конструкцию можно затем применять для получения алгоритмов при различных топологиях сетей или при различных допущениях, касающихся первоначальных сведений о процессах.

6.1. Определение и применение волновых алгоритмов

Далее в этой главе мы будем полагать, если только не будет оговорено противное, что сеть имеет фиксированную топологию (никаких изменений топологии не происходит), является неориентированной (по каждому каналу сообщения могут передаваться в обоих направлениях) и связной (между любыми двумя процессами пролегает путь). Множество всех процессов обозначим $\mathbb{P}$, а множество всех каналов обозначим E . Как и в предшествующих главах, мы будем считать, что в системе используется асинхронный обмен сообщениями и отсутствуют понятия глобального времени и вещественных часов. Все алгоритмы, приведенные в настоящей главе, можно использовать и при наличии синхронного обмена сообщениями (возможно, с некоторыми видоизменениями во избежание блокировки) или с глобальными часами, если таковые доступны. Однако некоторые более общие теоремы в этих случаях будут неверны (см. пример 6.1).

6.1.1. Определение волновых алгоритмов

Как было отмечено в гл. 2, в распределенном алгоритме допускается большое многообразие возможных вычислений вследствие недетерминизма как в самих процессах, так и в подсистеме коммуникаций. Всякое вычисление — это множество событий, частично упорядоченных по отношению $\preceq$ причинно-следственного предшествования согласно определению 2.20. Количество событий в вычислении C обозначим $|C|$, а подмножество событий, произошедших в процессе p , обозначим C_p . Предполагается, что существуют внутренние события специального типа, которые называются событиями *решения*; в алгоритмах настоящей главы такие события просто представлены оператором *decide*. В волновом алгоритме проводится обмен конечным числом сообщений, а затем принимается решение, которое имеет причинно-следственную зависимость от какого-либо события в каждом из процессов.

Определение 6.1. *Волновым алгоритмом* называется распределенный алгоритм, который удовлетворяет следующим трем требованиям.

1. **Завершение.** Каждое вычисление конечно:

$$\forall C : |C| < \infty.$$

2. **Решение.** Каждое вычисление содержит хотя бы одно событие решения:

$$\forall C : \exists e \in C : e \text{ является событием решения } decide.$$

3. **Зависимость.** В любом вычислении всякому событию решения предшествует в причинно-следственном отношении хотя бы одно событие в каждом из

процессов:

$$\forall C : \forall e \in C : (e \text{ — событие решения } decide \Rightarrow \forall q \in C : \exists f \in C_q : f \preceq e).$$

Вычисление волнового алгоритма называется *волной*. Еще одно обозначение: при выполнении алгоритма проводится различие между *инициаторами*, которые также называются *стартовыми процессами*, и *неинициаторами*, которые также называются *последователями*. Процесс является инициатором, если он запускает выполнение своего локального алгоритма самопроизвольно, т. е. алгоритм запускается по некоторому условию внутри процесса. Неинициатор становится вовлеченным в распределенный алгоритм, только когда по ходу вычисления поступает сообщение, которое запускает выполнение алгоритма процесса. Первое событие инициатора — это внутреннее событие или отправленное сообщение, а первое событие неинициатора — это событие приема сообщения.

Своим разнообразием волновые алгоритмы обязаны тому, что они могут отличаться во многих отношениях. Чтобы сделать осмысленным изучение в этой главе большого числа алгоритмов и чтобы облегчить выбор того или иного алгоритма для конкретных целей, мы приводим здесь список тех аспектов, в которых волновые алгоритмы отличаются друг от друга (см. также таблицу 6.18).

1. **Централизация.** Алгоритм называется *централизованным*, если у каждого вычисления есть в точности один инициализатор, и *децентрализованным*, если алгоритм может быть запущен спонтанно некоторым произвольным подмножеством процессов. Централизованные алгоритмы также называются *алгоритмами с одним источником*, а децентрализованные — *алгоритмами со многими источниками*. Как видно из таблицы 6.19, централизация оказывает заметное влияние на сложность волновых алгоритмов.

2. **Топология.** Алгоритм может быть спроектирован в расчете на специальную топологию, наподобие кольца, дерева, клики и т.п. (см. § 2.4.1 и § Б.4.2).

3. **Первоначальные сведения.** Алгоритм может исходить из предположения о том, что процессам изначально доступны сведения того или иного рода (см. § 2.4.4). Примерами таких сведений могут служить

- а) *самоидентификация*: каждый процесс изначально знает свое собственное уникальное имя;
- б) *идентификация соседей*: каждый процесс изначально знает имена своих соседей;
- в) *восприятие направления*: см. § Б.4.3.

4. **Число решений.** Во всех волновых алгоритмах, рассматриваемых в этой главе, в каждом процессе принимается не более одного решения. Число процессов, в которых выполняется событие решения, может варьироваться; в одних алгоритмах решение принимает только один процесс, в других это могут делать все процессы. В древесном алгоритме (см. § 6.2.2) в точности два процесса принимают решение.

5. **Сложность.** В настоящей главе мерами сложности служат количество обменов сообщениями, число битов информации, отправленных при обмене сообщениями, и время, затрачиваемое одним вычислением (согласно определению из § 6.4.4). См. также § 2.4.5.

В этой главе в описании каждого волнового алгоритма указываются используемые переменные и, при необходимости, та информация, которая содержится в сообщениях. В большинстве таких алгоритмов отправляются «пустые» сообщения, не содержащие какой-то реальной информации; такие сообщения передают причинно-следственную обусловленность, а не информацию. В алгоритмах 6.8, 6.10, 6.11, и 6.17 сообщения используются для передачи содержательной информации. В алгоритмах 6.14 и 6.15/6.16 задействованы сообщения различных типов; здесь требуется, чтобы каждое сообщение содержало один или два бита для различения типов сообщений.

Когда применяется волновой алгоритм, то, вообще говоря, в нем используется большее число переменных, а в сообщениях может быть включена и другая информация. Во многих приложениях происходит совместное или последовательное распространение нескольких волн; в таких случаях в сообщение должна быть включена информация о той волне, к которой это сообщение относится. Кроме того, во всяком процессе могут быть задействованы дополнительные переменные для управления волной или волнами, в которых этот процесс принимает активное участие.

Важный подкласс волновых алгоритмов представляют централизованные волновые алгоритмы, наделенные следующими двумя дополнительными свойствами: единственным процессом, принимающим решения, является инициатор, и все события линейно упорядочены по отношению причинно-следственной зависимости. Волновые алгоритмы такого рода называются *алгоритмами обхода* и рассматриваются в § 6.6.3.

6.1.2. Основные свойства волновых алгоритмов

В этом параграфе мы докажем несколько лемм, которые позволят лучше понять устройство волнового вычисления, а также установим две простые нижние оценки сложности волновых алгоритмов по числу обменов сообщениями.

Структурные свойства волн. Во-первых, каждому событию в вычислении предшествует некоторое событие в инициаторе.

Лемма 6.2. *Для каждого события $e \in C$ существуют такой инициатор p и такое событие f в C_p , что $f \preceq e$.*

Доказательство. Выберем в качестве f минимальный элемент в истории события e , т.е. такой элемент, для которого $f \preceq e$ и нет события $f' \prec f$. Указанный элемент f существует, поскольку история каждого события конечна. Остается показать, что процесс p , в котором происходит событие f , является инициатором. Прежде всего отметим, что f — это первое событие процесса p , ибо иначе более ранние события p предшествовали бы f . Первое событие неинициатора — это событие приема, которому должно предшествовать соответствующее событие отправления, что противоречит минимальности f . Значит, p — инициатор. $\square$

Волна с одним инициатором определяет остовное дерево сети, если для каждого неинициатора выделить канал, по которому он получает первое сообщение.

Лемма 6.3. *Пусть C — волна с одним инициатором p . Для каждого неинициатора q обозначим символом $father_q$ того соседа процесса q , от которого q получил первое сообщение. Тогда граф $T = (P, E_T)$, где $E_T = \{qr : q \neq p \wedge r = father_q\}$, является остовным деревом, направленным к p .*

Доказательство. Поскольку число вершин в T превосходит число дуг на единицу, достаточно показать, что T не содержит циклов. И это действительно так, поскольку для e_q , первого события в q , наличие дуги $qr \in E_T$ влечет $e_r \preceq e_q$, а отношение $\preceq$ является частичным порядком. $\square$

В качестве события f , упомянутого в третьем пункте определения 6.1, можно выбрать событие отправления сообщения для всех процессов q за исключением того процесса, в котором происходит событие *decide*.

Лемма 6.4. *Пусть C — волна и $d_p \in C$ — событие решения в процессе p . Тогда*

$$\forall q \neq p : \exists f \in C_q : (f \preceq d_p \wedge f \text{ — событие отправления сообщения}).$$

Доказательство. Так как C — волна, существует событие $f \in C_q$, которое является причинно-следственным предшественником события d_p . Выберем в качестве f последнее событие в C_q , предшествующее d_p . Чтобы показать, что f — это событие отправления сообщения, заметим, что из определения причинно-следственной зависимости (определение 2.20) вытекает существование такой последовательности (причинно-следственной цепочки)

$$f = e_0, e_1, \dots, e_k = d_p,$$

что для каждого $i < k$ события e_i и e_{i+1} являются либо последовательными событиями в одном и том же процессе, либо парой соответствующих событий отправления-приема сообщения. Поскольку f — последнее событие в процессе q , предшествующее событию d_p , событие e_1 происходит в процессе, отличном от q . Значит, f — событие отправления сообщения. $\square$

Нижние оценки сложности волн. Из леммы 6.4 немедленно вытекает нижняя оценка $N - 1$ для числа сообщений, которыми обмениваются процессы при прохождении волны. Если событие *decide* происходит только в инициаторе волны (что имеет место для алгоритмов обхода), то оценка увеличивается до N сообщений и волновые алгоритмы для произвольных сетей используют по меньшей мере $|E|$ сообщений.

Теорема 6.5. *Пусть C — такая волна с одним инициатором p , что в p происходит событие решения d_p . Тогда в вычислении C происходит обмен по крайней мере N сообщениями.*

Доказательство. По лемме 6.2 каждому событию в C предшествует некоторое событие в процессе p , и, воспользовавшись причинно-следственной

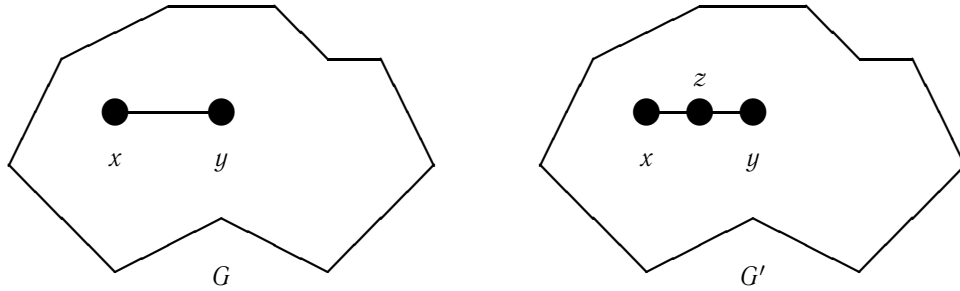


Рис. 6.1. Вставка процесса в неиспользуемый канал

цепочкой наподобие той, которая фигурировала в доказательстве леммы 6.4, нетрудно показать, что в p произойдет хотя бы одно событие отправления сообщения. По лемме 6.4 событие отправления сообщения также произойдет и в каждом другом процессе, а это означает, что произойдет как минимум N обменов сообщениями. $\square$

Теорема 6.6. Пусть A — волновой алгоритм для произвольной сети, в котором не используются первоначальные сведения об отличительных признаках соседей. Тогда A в каждом вычислении проводит по меньшей мере $|E|$ обменов сообщениями.

Доказательство. Предположим, что A имеет вычисление C , в котором проводится менее $|E|$ обменов сообщениями, т.е. существует канал, например, xy , по которому не передаются сообщения в вычислении C (см. рис. 6.1.2). Рассмотрим сеть G' , полученную за счет добавления одной точки z в канале между x и y . Поскольку точки не идентифицируют своих соседей, начальное состояние x и y в G' будет тем же самым, что и в сети G . Это будет, конечно, справедливо и для всех остальных точек G . Следовательно, все события в C произойдут в том же самом порядке начиная с исходной конфигурации G' , но теперь событию *decide* не предшествует никакое событие в z . $\square$

В гл. 7 будет доказана уточненная нижняя оценка сложности по числу сообщений для *децентрализованных* волновых алгоритмов на кольцах и произвольных сетях без предварительных сведений о соседях (см. следствия 7.14 и 7.16).

6.1.3. Распространение информации с обратной связью

В этом параграфе будет показано, что волновые алгоритмы — это как раз те самые алгоритмы, которые необходимы для широковещательного распространения информации всем процессам с тем условием, что некоторые выделенные процессы должны получить подтверждение о завершении выполнения широковещательной связи. Это требование применительно к распространению информации с обратной связью (Propagation of Information with Feedback, PIF) приводит к следующей задаче (см. [174]). Сформировано подмножество процессов, обла-

дающих некоторым сообщением M (одним и тем же для всех процессов этого подмножества), которое должно быть распространено широковещательно, т.е. все процессы должны получить и принять M . Некоторые выделенные процессы должны быть уведомлены о завершении широковещательной связи; это означает, что в них должно быть выработано специальное событие *notify* с тем условием, что событие *notify* произойдет только в том случае, когда все процессы уже получают сообщение M . Алгоритм будет использовать конечное число сообщений.

Уведомление в алгоритме PIF рассматривается как событие решения.

Теорема 6.7. Каждый PIF-алгоритм — это волновой алгоритм.

Доказательство. Пусть P — это некоторый PIF-алгоритм. Тогда каждое вычисление алгоритма P конечно и в каждом вычислении происходит уведомляющее событие (*decide*). Если в некотором вычислении алгоритма P происходит уведомление d_p , которому не предшествует никакое событие в процессе q , то по теореме 2.21 и утверждению 2.23 существует такое выполнение алгоритма P , в котором уведомление происходит, до того как q получит какое-либо сообщение, что противоречит нашим требованиям. $\square$

Необходимо иметь в виду, что теорема 2.21 справедлива только для систем с асинхронной передачей сообщений (см. упражнение 6.1).

Теорема 6.8. Каждый волновой алгоритм может быть использован в качестве PIF-алгоритма.

Доказательство. Пусть A — волновой алгоритм. Чтобы представить A как PIF-алгоритм, процессы, первоначально хранящие M , следует считать стартовыми. Информацию M нужно добавить к каждому сообщению, передаваемому в алгоритме A . Это возможно, поскольку по построению стартовые процессы алгоритма A первоначально обладают информацией M , а последователи не отправляют сообщений, до тех пор пока сами они не получают хоть одно сообщение, а значит так же станут осведомлены о M . Событию *decide* в волне предшествует по крайней мере одно событие в каждом процессе. Следовательно, когда событие решения произойдет, каждый процесс будет уже осведомлен о M , и это можно расценивать как требуемое PIF-алгоритмом уведомление *notify*. $\square$

Построенный PIF-алгоритм имеет такую же сложность, как и A , и он обладает всеми теми же свойствами, упомянутыми в § 6.1.1, что и A , за исключением битовой сложности. Битовая сложность может быть понижена, если присоединять M только к самому первому сообщению, отправляемому по каждому каналу. Если w — число битов сообщения M , то битовая сложность окончательного алгоритма превзойдет битовую сложность алгоритма A на $w|E|$.

6.1.4. Синхронизация

В этом параграфе мы покажем, что волновые алгоритмы — это как раз те алгоритмы, которые необходимы для того, чтобы добиться глобальной синхронизации между процессами. Требование синхронизации (SYN) выдвинуто в следующей задаче; см. [79]. В каждом процессе q должно быть выполнено событие a_q ,

и в некоторых процессах событие b_p должно быть выполнено так, чтобы выполнение a_q произошло по времени ранее, нежели выполнение любого из событий b_p . Алгоритм должен использовать конечное число сообщений.

В SYN-алгоритме события b_p рассматриваются как события *decide*.

Теорема 6.9. *Каждый SYN-алгоритм является волновым алгоритмом.*

Доказательство. Пусть S — это некоторый SYN-алгоритм. Тогда каждое вычисление алгоритма S конечно и в каждом вычислении происходит событие b_p (*decide*). Если в некотором вычислении алгоритма S происходит событие b_p , у которого нет причинно-следственного предшественника a_q , то по теореме 2.21 и утверждению 2.23, существует вычисление алгоритма S , в котором b_p происходит ранее a_q . $\square$

Теорема 6.10. *Каждый волновой алгоритм можно использовать в качестве SYN-алгоритма.*

Доказательство. Пусть A — волновой алгоритм. Чтобы использовать A в качестве SYN-алгоритма, каждому процессу q требуется выполнить a_q , до того как q в рамках алгоритма A отправит некоторое сообщение или примет решение. Событие b_p произойдет в p после события *decide*. По лемме 6.4 всякому событию *decide* причинно-следственно предшествует событие a_q в каждом процессе q . $\square$

Построенный SYN-алгоритм имеет ту же самую сложность по числу сообщений, что и A , а также обладает всеми прочими свойствами алгоритма A , упомянутыми в § 6.1.1.

6.1.5. Вычисление функций точной нижней грани

В этом параграфе будет показано, что волновые алгоритмы — это в точности алгоритмы, необходимые для вычисления функции, значение которой существенно зависит от входных данных каждого процесса. Представителем такого класса функций могут служить функции *точной нижней грани* по всем входам, если это возможно для частично упорядоченного множества.

Пусть $(X, \leq)$ — частично упорядоченное множество. Тогда элемент c называется *точной нижней гранью* элементов a и b , если $c \leq a$, $c \leq b$ и $\forall d: (d \leq a \wedge d \leq b \implies d \leq c)$. Допустим, что X обладает тем свойством, что точная нижняя грань всегда существует. В таком случае эта точная нижняя грань определяется однозначно и обозначается $a \sqcap b$. Двуместная операция $\sqcap$ является коммутативной ($a \sqcap b = b \sqcap a$) и ассоциативной ($a \sqcap (b \sqcap c) = (a \sqcap b) \sqcap c$), и поэтому ее можно распространить на конечные множества:

$$\inf\{j_1, \dots, j_k\} = j_1 \sqcap \dots \sqcap j_k.$$

Проблема вычисления точной нижней грани (Infimum Computation, INF) такова. Все процессы q наделены входными данными j_q , которые являются элементами частично упорядоченного множества X . Требуется, чтобы некоторые выделенные процессы вычислили значение $\inf\{j_p : q \in \mathbb{P}\}$ и чтобы эти процессы были

осведомлены о завершении вычисления. Они записывают результат вычисления в переменную *out*, и им не разрешается впоследствии изменять значение этой переменной.

Запись значения в переменную *out*, рассматривается как событие *decide* в INF-алгоритме.

Теорема 6.11. *Каждый INF-алгоритм является волновым алгоритмом.*

Доказательство. Пусть I — это некоторый INF-алгоритм. Предположим, что существует вычисление C алгоритма I с начальной конфигурацией γ , в котором процесс p записывает значение J в переменной *out* _{p} , и этому событию записи не предшествует никакое событие в процессе q . Тогда рассмотрим начальную конфигурацию γ' , отличающуюся от γ только тем, что q имеет другие входные данные j'_q , которые выбраны так, чтобы выполнялось соотношение $J \not\leq j'_q$. Поскольку никакое использование входных данных процесса q в вычислении C не является причинно-следственным предшественником события записи в p , все события C , предшествующие этому событию записи, происходят в том же самом порядке и для вычисления, начинающегося с γ' . В образовавшемся вычислении p записывает такой же результат J , как и в вычислении C , но теперь этот результат ошибочный. $\square$

Теорема 6.12. *Каждый волновой алгоритм можно использовать для вычисления точной нижней грани.*

Доказательство. Пусть задан алгоритм A . Придадим каждому процессу q дополнительную переменную v_q , которая инициализирована элементом j_q . По ходу волны значения этой переменной претерпевают следующие изменения. Всякий раз, когда процесс q отправляет сообщение, предусмотренное алгоритмом A , текущее значение v_q включается в сообщение. Всякий раз, когда процесс q получает сообщение, предусмотренное алгоритмом A и содержащее значение v , переменная v_q полагается равной $v_q \sqcap v$. Когда в процессе p происходит событие *decide*, текущее значение v_p записывается в *out* _{p} .

Покажем, что алгоритм выдает правильное значение. Правильным называется такой ответ J , что $J = \inf\{j_q : q \in \mathbb{P}\}$. Для всякого события a в процессе q обозначим символом $v^{(a)}$ значение v_q сразу после осуществления события a . Поскольку вначале значение v_q было равно j_q и по ходу волны могло только уменьшиться, соотношение $v^{(a)} \leq j_q$ соблюдается для каждого события a в q . Присваивания переменной v устроены так, что для любых событий a и b имеет место соотношение $a \preceq b \implies v^{(a)} \geq v^{(b)}$. Кроме того, поскольку v вычисляется как точная нижняя грань уже существующих значений, соотношение $J \leq v$ соблюдается для всех значений по ходу волны. Таким образом, если d — это событие *decide* в p , то значение $v^{(d)}$ удовлетворяет соотношению $J \leq v^{(d)}$, а поскольку событию d предшествует хотя бы одно событие в каждом процессе q , мы имеем $v^{(d)} \leq j_q$ для всех q . Отсюда следует, что $J = v^{(d)}$. $\square$

Построенный INF-алгоритм обладает всеми теми же свойствами, что и A , за исключением битовой сложности, поскольку к каждому сообщению A присоединялся элемент из X . Понятие функции точной нижней грани может показаться

довольно-таки абстрактным, но на самом деле многие функции могут быть представлены как точные нижние грани, о чем свидетельствует работа [187, теорема 4.1.1.2].

Теорема 6.13 (о точной нижней грани). Пусть $\star$ — это такая двухместная операция на множестве X , которая обладает следующими свойствами:

- 1) операция $\star$ коммутативна, т.е. $a \star b = b \star a$,
- 2) операция $\star$ ассоциативна, т.е. $(a \star b) \star c = a \star (b \star c)$,
- 3) операция $\star$ идемпотентна, т.е. $a \star a = a$.

Тогда существует частичный порядок $\leq$ на X , для которого $\star$ является функцией точной нижней грани.

К числу операций, удовлетворяющих трем перечисленным выше требованиям, относятся логические конъюнкция и дизъюнкция, функции минимума и максимума на множестве целых чисел, наибольший общий делитель и наименьшее общее кратное на множестве целых чисел, а также операции пересечения и объединения множеств.

Следствие 6.14. Функции $\wedge$, $\vee$, $\min$, $\max$, НОК, НОД, $\cap$, и $\cup$, зависящие от значений, локализованных во всех процессах, можно вычислить по ходу одной единственной волны.

Вычисление операций, которые являются коммутативными, ассоциативными, но не идемпотентными, рассматривается в § 6.5.2.

6.2. Семейство волновых алгоритмов

В следующих трех параграфах представлено семейство волновых алгоритмов и алгоритмов обхода. Во всех случаях описание алгоритма приводится для выделенного процесса p .

6.2.1. Кольцевой алгоритм

В этом параграфе будет представлен волновой алгоритм для кольцевой сети. Этот же самый алгоритм можно использовать и для гамильтоновой сети, в которой информация о некотором фиксированном гамильтоновом цикле заложена во всех процессах. Предполагается, что каждому процессу p предписан сосед $Next_p$ так, чтобы каналы связи между выделенными таким образом соседями образовывали гамильтонов цикл.

Алгоритм является централизованным: инициатор отправляет сообщение **tok** (оно называется *маркером*) по циклу, каждый процесс передает его далее, и, когда оно возвращается инициатору, тот принимает решение (см. алгоритм 6.1).

Теорема 6.15. Кольцевой алгоритм (алгоритм 6.1) является волновым алгоритмом.

Доказательство. Пусть инициатором является процесс p_0 . Поскольку каждый процесс отправляет не более одного сообщения, всего в алгоритме происходит обмен не более N сообщениями.

Для инициатора:

begin send **tok** to $Next_p$; receive **tok**; *decide* **end**

Для неинициатора:

begin receive **tok**; send **tok** to $Next_p$ **end**

Алгоритм 6.1. Кольцевой алгоритм

Через конечное число шагов алгоритм достигает заключительной конфигурации. В этой конфигурации процесс p_0 уже отправил маркер, т.е. выполнил оператор send своей программы. При этом никакое сообщение **tok** не передается ни по одному каналу, ибо иначе оно бы могло быть получено и конфигурация нельзя было бы считать заключительной. Кроме того, ни один процесс, отличный от p_0 , не «хранит» маркер (т.е. уже получил, но еще не отправил **tok**), ибо иначе этот процесс мог бы отправить **tok** и конфигурация была бы не заключительной. Приходим к следующему заключению:

- 1) процесс p_0 уже отправил маркер,
- 2) для каждого процесса p , отправившего маркер, процесс $Next_p$ уже получил этот маркер,
- 3) каждый процесс $p \neq p_0$, получивший маркер, уже отправил маркер.

Отсюда, а также из свойства $Next$ вытекает, что каждый процесс уже получил и уже отправил маркер. Поскольку процесс p_0 уже получил маркер, и конфигурация является заключительной, процесс p_0 выполнил оператор *decide*.

Получение и отправление сообщения **tok** каждым процессом $p \neq p_0$ предшествует событию получения маркера процессом p_0 . Значит, условие зависимости соблюдено. $\square$

6.2.2. Древесный алгоритм

В этом параграфе будет представлен волновой алгоритм для древесной сети. Этот же алгоритм можно использовать и для произвольной сети, если иметь доступ к остовному дереву этой сети. Предполагается, что все листья сети запускают наш алгоритм. Каждый процесс отправляет в точности одно сообщение на протяжении работы алгоритма. Если процесс уже получил сообщение по каждому из инцидентных ему каналов, кроме одного (это условие первоначально верно для листьев), то он отправляет сообщение по оставшемуся каналу. Если процесс уже получил сообщение по всем инцидентным ему каналам, то он принимает решение (см. алгоритм 6.2).

Для доказательства того, что этот алгоритм является волновым, потребуются некоторые обозначения. Будем использовать запись f_{pq} для обозначения события отправления сообщения процессом p процессу q , а g_{pq} — для обозначения события приема процессом q сообщения от процесса p . Обозначим символом T_{pq}

множество процессов, которые достижимы из p без прохождения по ребру pq (процессы, лежащие в дереве «по ту же сторону» этого ребра, что и вершина p). Из связности сети следуют соотношения (см. рис. 6.3)

$$T_{pq} = \bigcup_{r \in \text{Neigh}_p \setminus \{q\}} T_{rp} \cup \{p\} \quad \text{и} \quad \mathbb{P} = \{p\} \cup \bigcup_{r \in \text{Neigh}_p} T_{rp}.$$

```

var  $\text{rec}_p[q]$  for each  $q \in \text{Neigh}_p$  : boolean init false ;
    (*  $\text{rec}_p[q]$  принимает значение true, если  $p$  уже получил сообщение от  $q$  *)

begin while  $\#\{q : \text{rec}_p[q] = \text{false}\} > 1$  do
    begin receive (tok) from  $q$  ;  $\text{rec}_p[q] := \text{true}$  end;
    (* Теперь есть единственный  $q_0$ , для которого  $\text{rec}_p[q_0]$  имеет значение false *)
    send (tok) to  $q_0$  with  $\text{rec}_p[q_0] = \text{false}$  ;
     $x$ : receive (tok) from  $q_0$  ;  $\text{rec}_p[q_0] := \text{true}$  ;
    decide
    (* Оповестить другие процессы о решении:
    forall  $q \in \text{Neigh}_p, q \neq q_0$  do send (tok) to  $q$  *)

end

```

Алгоритм 6.2. Древетный алгоритм

Смысл «закомментированного» оператора **forall** в алгоритме 6.2 мы обсудим в конце этого параграфа. Следующая теорема касается того варианта указанного алгоритма, в котором не содержится этого оператора.

Теорема 6.16. *Древетный алгоритм (алгоритм 6.2) является волновым алгоритмом.*

Доказательство. Поскольку каждый процесс отправляет не более одного сообщения, всего в алгоритме используется не более N сообщений. Отсюда следует, что алгоритм достигает заключительной конфигурации γ спустя конечное число шагов. Мы покажем, что в γ хотя бы в одном из процессов уже произошло событие *decide*.

Пусть F — это число битов *rec* со значением **false** в конфигурации γ , а K — количество процессов, которые уже отправили сообщения в конфигурации γ . Поскольку в γ никаких сообщений не пересылается (иначе конфигурация γ не была бы заключительной), $F = (2N - 2) - K$. Общее число битов *rec* равно $2N - 2$, и K из них имеют значение **true**.

Допустим, что ни один из процессов не принял решения в γ . Тогда $N - K$ процессов, которые в γ еще не отправляли сообщения, имеют в *rec* по меньшей мере два бита со значением **false**; в противном случае они могли бы отправить сообщение, вопреки тому что γ — заключительная конфигурация. Кроме того, K процессов, которые в γ уже отправили сообщение, имеют в *rec* хотя бы по одному биту со значением **false**; в противном случае они могли бы принять решение вопреки сделанному предположению о γ . Значит, $F \geq 2(N - K) + K$, и из неравен-

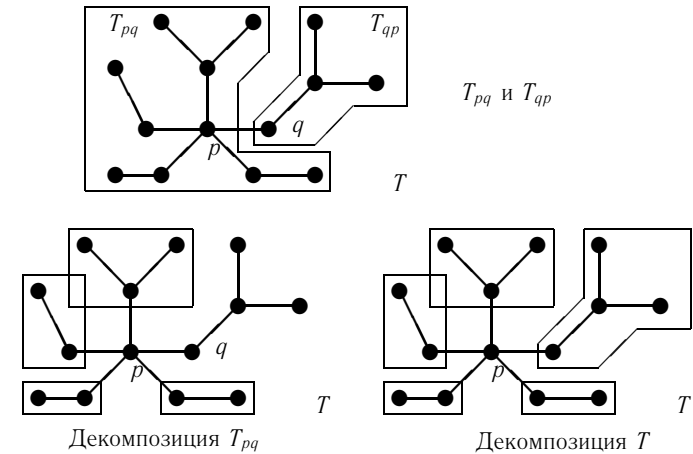


Рис. 6.3. Поддерева T_{pq}

ства $(2N - 2) - K \geq 2(N - K) + K$ следует, что $-2 \geq 0$. Полученное противоречие означает, что хотя бы одно решение в γ уже принято; см. также упражнение 6.5.

Наконец, необходимо показать, что решение предшествует какому-либо событию в каждом из процессов. Напомним, что f_{pq} обозначает событие отправления процессом p сообщения процессу q , а g_{pq} — событие приема процессом q сообщения от процесса p . Воспользуемся индукцией по числу событий приема, чтобы показать, что

$$\forall s \in T_{pq} \exists e \in C_s : e \preceq g_{pq}.$$

Допустим, это верно для всех событий приема, предшествующих событию g_{pq} . Тогда событию g_{pq} предшествует событие f_{pq} (в процессе p) и из устройства программы p следует, что для всех $r \in \text{Neigh}_p$ при $r \neq q$ событию f_{pq} предшествует событие g_{rp} . Из индуктивной гипотезы вытекает, что для всех таких r и для всех $s \in T_{rp}$ найдется такое событие $e \in C_s$, что $e \preceq g_{rp}$. Значит, $e \preceq g_{pq}$.

Решению d_p в p предшествует событие g_{rp} для всех $r \in \text{Neigh}_p$. И отсюда следует, что $\forall s \in \mathbb{P} \exists e \in C_s : e \preceq d_p$. $\square$

Читатель может промоделировать вычисление алгоритма на небольшом дереве (например, на дереве, изображенном на рис. 6.3), и удостовериться в справедливости следующих замечаний. В алгоритме 6.2 есть *два* процесса, которые получают сообщение по каждому из своих каналов и принимают решение; все остальные процессы еще ожидают сообщения, пребывая в заключительной конфигурации в точке x своей программы. Если к нашей программе добавить оператор **forall** (тот самый, который «закомментирован» в алгоритме 6.2), то решение будут принимать все процессы, и в заключительной конфигурации каждый процесс будет пребывать в заключительном состоянии. Всего в модифицированной программе используется $2N - 2$ сообщений.

6.2.3. Алгоритм эха

Алгоритм эха — это централизованный волновой алгоритм для сетей с произвольной топологией. Впервые как отдельный алгоритм он был представлен Ченем в работе [41] и поэтому иногда называется алгоритмом Ченя. Несколько более эффективная версия была предложена Сигаллом [174], и мы рассмотрим здесь именно ее.

Алгоритм наводняет сообщениями **tok** все процессы, выделяя таким образом остовное дерево, как это было определено в лемме 6.3. Маркеры возвращаются «эхом» обратно по ребрам этого дерева, напоминая поток сообщений в древесном алгоритме (см. алгоритм 6.4). Инициатор отправляет сообщение всем своим соседям. После получения первого сообщения всякий неинициатор переправляет сообщения всем своим соседям, за исключением того, от которого было получено это сообщение. Как только неинициатор получит сообщения от всех своих соседей, он отправляет эхо родительскому процессу. Как только инициатор получит сообщения от всех своих соседей, он принимает решение.

```

var  $rec_p$  : integer init 0 ; (* Подсчет числа полученных сообщений *)
       $father_p$  :  $\mathbb{P}$  init undef ;

```

Для инициатора:

```

begin forall  $q \in Neigh_p$  do send  $\langle tok \rangle$  to  $q$  ;
      while  $rec_p < \#Neigh_p$  do
        begin receive  $\langle tok \rangle$  ;  $rec_p := rec_p + 1$  end ;
      decide
end

```

Для неинициатора:

```

begin receive  $\langle tok \rangle$  from neighbor  $q$  ;  $father_p := q$  ;  $rec_p := rec_p + 1$  ;
      forall  $q \in Neigh_p, q \neq father_p$  do send  $\langle tok \rangle$  to  $q$  ;
      while  $rec_p < \#Neigh_p$  do
        begin receive  $\langle tok \rangle$  ;  $rec_p := rec_p + 1$  end ;
      send  $\langle tok \rangle$  to  $father_p$ 
end

```

Алгоритм 6.4. Алгоритм эха

Теорема 6.17. *Алгоритм эха (алгоритм 6.4) является волновым алгоритмом.*

Доказательство. Поскольку каждый процесс отправляет не более одного сообщения по инцидентному каналу, число сообщений, которыми обмениваются процессы на протяжении одного вычисления, конечно. Пусть γ — заключительная конфигурация, которая достигается при вычислении C с инициатором p_0 .

Для этой конфигурации определим (так же, как в лемме 6.3) граф $T = (\mathbb{P}, E_T)$, полагая $pq \in E_T \iff father_p = q$. Чтобы показать, что этот граф является деревом, необходимо установить, что число ребер меньше числа вершин на единицу (лемма 6.3 утверждает, что T — дерево, но исходя из предположения о том, что используемый алгоритм — волновой, а это как раз то, что нам здесь нужно доказать). Заметим, что каждый процесс, участвующий в C , отправляет сообщения всем своим соседям, за исключением (если только этот процесс не является инициатором) того соседа, от которого было получено первое сообщение. Значит, каждый его сосед получит хотя бы одно сообщение в C и также примет участие в C . Отсюда следует, что $father_p \neq undef$ для всех $p \neq p_0$. То, что T не содержит циклов, было показано в доказательстве леммы 6.3.

Корнем дерева служит вершина p_0 . Обозначим символом T_p множество вершин поддеревя, растущего из вершины p . Ребра сети, не принадлежащие T , назовем стягивающими. В конфигурации γ каждый процесс p обязательно должен отправить сообщения всем своим соседям, за исключением родительской вершины $father_p$, и, следовательно, в ходе вычисления C по каждому стягивающему ребру должно передаваться сообщение в обе стороны. Обозначим символом f_p событие отправления процессом p сообщения родительской вершине (если таковое случается в C), а символом g_p — событие получения процессом $father_p$ сообщения от p (если это происходит). Индукцией по числу вершин дерева можно показать, что

- 1) C содержит событие f_p для каждого $p \neq p_0$;
- 2) для всех $s \in T_p$ есть такое $e \in C_s$, что $e \preceq g_p$.

Мы рассмотрим следующие два случая.

Процесс p — это листовая вершина. Процесс p в вычислении C уже получил сообщение от своей родительской вершины и от всех своих соседей (поскольку все прочие каналы являются стягивающими ребрами). Значит, он мог совершить отправление сообщения **tok** процессу $father_p$. Коль скоро γ является заключительной конфигурацией, это сообщение было отправлено. Дерево T_p содержит одну лишь вершину p , и, очевидно, $f_p \preceq g_p$.

Процесс p не является листовой вершиной. Процесс p в C уже получил сообщение от родительской вершины, а также по всем стягивающим ребрам. По индуктивной гипотезе в вычислении C для каждой сыновней вершины p' процесса p представлено событие $f_{p'}$. Коль скоро γ — это заключительная конфигурация, вычисление C содержит также и событие $g_{p'}$. Следовательно, событие отправления сообщения **tok** процессу $father_p$ имеет все предпосылки, для того чтобы осуществиться. И поскольку конфигурация γ является заключительной, это сообщение было отправлено. Дерево T_p содержит объединение деревьев $T_{p'}$ по всем сыновним вершинам процесса p и саму вершину p . Применяя индуктивную гипотезу, можно показать, что в каждом процессе из этого множества есть событие, предшествующее g_p .

Отсюда также следует, что процесс p_0 уже получил сообщение от каждого своего соседа и что p_0 осуществил событие *decide*, которому предшествует хотя бы одно событие в каждом из процессов. $\square$

Остовное дерево, построенное в ходе вычисления алгоритма 6.4, иногда используется последующими алгоритмами. Например, алгоритм Мерлина—Сигалла для вычисления таблицы маршрутизации по кратчайшим путям исходит из предположения о том, что остовное дерево с корнем в v_0 уже задано; см. § 4.2.3. Это исходное остовное дерево может быть вычислено при помощи алгоритма эха. В последней конфигурации алгоритма каждый процесс (отличный от p_0) запоминает, кто из соседей является его родительской вершиной, но своих сыновних вершин среди соседей он в этом дереве не различает. В нашем алгоритме одно и то же сообщение поступает от родительской вершины по стягивающим ребрам, а также от сыновних вершин. Если требуются сведения о сыновних вершинах в построенном дереве, алгоритм нужно чуть-чуть видоизменить так, чтобы сообщение, отправляемое родительской вершине (последняя операция отправления сообщения для неинициаторов), было *особым*. Сыновними вершинами процесса будут признаны те соседи, от которых поступило особое сообщение.

6.2.4. Алгоритм опроса

В сети, образующей клику, каждая пара процессов соединена каналом. Всякий процесс может судить о том, получил ли он сообщение от каждого своего соседа. В *алгоритме опроса* (см. 6.5) инициатор обращается к каждому соседу с просьбой отправить ему сообщение и принимает решение после получения всех сообщений.

```
var  $rec_p$  : integer  init 0 ; (* только для инициатора *)
```

Для инициатора:

```
begin forall  $q \in Neigh_p$  do send  $\langle tok \rangle$  to  $q$  ;
  while  $rec_p < \#Neigh_p$  do
    begin receive  $\langle tok \rangle$  ;  $rec_p := rec_p + 1$  end;
  decide
end
```

Для неинициатора:

```
begin receive  $\langle tok \rangle$  from  $q$  ; send  $tok$  to  $q$  end
```

Алгоритм 6.5. Алгоритм опроса

Теорема 6.18. *Алгоритм опроса (алгоритм 6.5) является волновым алгоритмом.*

Доказательство. В этом алгоритме по каждому каналу, инцидентному инициатору, отправляются два сообщения. Каждый сосед инициатора отвечает один раз на полученный запрос, и, следовательно, инициатор получает $N - 1$ откликов. Это как раз то число, которое необходимо для принятия решения, и от-

сюда следует, что инициатор примет решение, и этому решению предшествует некоторое событие в каждом из процессов. $\square$

Опрос можно проводить также и в звездных сетях, если инициатор будет располагаться в центре.

6.2.5. Фазовый алгоритм

В этом параграфе будет представлен фазовый алгоритм, который является децентрализованным алгоритмом, пригодным для сетей с произвольной топологией. Этот алгоритм был предложен в работе [187, § 4.2.3]. Его можно использовать в качестве волнового алгоритма для ориентированных сетей.

В этом алгоритме требуется, чтобы все процессы располагали сведениями о диаметре сети, который обозначен константой D в тексте описания алгоритма. Алгоритм будет оставаться корректным (хотя и менее эффективным), если все процессы будут использовать вместо D константу D' , превышающую диаметр сети. Таким образом, для применения фазового алгоритма не нужно знать точного значения диаметра; достаточно, если будет известна верхняя оценка (например, $N - 1$) диаметра сети. Во всех процессах должна использоваться *одна и та же* константа D' . Пелег в работе [158] расширил этот алгоритм так, чтобы диаметр вычислялся по ходу выполнения алгоритма, но в этом случае требуется однозначная идентифицируемость процессов.

Общий случай. Алгоритм можно применять для всякой *ориентированной* сети, по каналам которой осуществляется односторонняя передача сообщений. В этом случае соседями вершины p будут *соседи на входе* (т. е. процессы, которые могут отправлять сообщения процессу p) и *соседи на выходе* (т. е. процессы, которым p может отправлять сообщения). Соседи процесса p на входе образуют множество In_p , а соседи на выходе — множество Out_p .

В фазовом алгоритме всякий процесс отправляет в точности D сообщений каждому соседу на выходе. При этом $(i + 1)$ -е сообщение отправляется каждому соседу на выходе только после того, как были получены i сообщений от каждого соседа на входе (см. алгоритм 6.6).

И в самом деле, из описания алгоритма сразу видно, что по каждому каналу передается не более D сообщений (то, что по каждому каналу передается не менее D сообщений, мы покажем ниже). Для каждого ребра pq выражение $f_{pq}^{(i)}$ будет обозначать i -е событие отправления процессом p сообщения процессу q , а выражение $g_{pq}^{(i)}$ будет обозначать i -е событие приема процессом q сообщения от процесса p . Если канал связи между процессами p и q имеет тип очереди (first in — first out), эти события соответствуют друг другу, и соотношение $f_{pq}^{(i)} \preceq g_{pq}^{(i)}$, очевидно, выполняется. Причинно-следственные отношения между $f_{pq}^{(i)}$ и $g_{pq}^{(i)}$ соблюдаются и в тех случаях, когда канал не является очередью, о чем свидетельствует следующая лемма.

Лемма 6.19. *Соотношение $f_{pq}^{(i)} \preceq g_{pq}^{(i)}$ соблюдается и тогда, когда канал не является очередью.*

```

cons  $D$       : integer      = диаметр сети ;
var   $Rec_p[q]$  :  $0..D$       init 0 для каждого  $q \in In_p$  ;
      (* Число сообщений, полученных от  $q$  *)
       $Sent_p$     :  $0..D$       init 0 ;
      (* Число сообщений, отправленных каждому соседу на выходе *)

begin if  $p$  is initiator then
  begin forall  $r \in Out_p$  do send  $\langle \text{tok} \rangle$  to  $r$  ;
     $Sent_p := Sent_p + 1$ 
  end;
  while  $\min_q Rec_p[q] < D$  do
    begin receive  $\langle \text{tok} \rangle$  (from neighbor  $q_0$ ) ;
       $Rec_p[q_0] := Rec_p[q_0] + 1$  ;
      if  $\min_q Rec_p[q] \geq Sent_p$  and  $Sent_p < D$  then
        begin forall  $r \in Out_p$  do send  $\langle \text{tok} \rangle$  to  $r$  ;
           $Sent_p := Sent_p + 1$ 
        end
      end;
    decide
  end

```

Алгоритм 6.6. Фазовый алгоритм

Доказательство. Допустим, что число m_h таково, что $f_{pq}^{(m_h)}$ — это событие отправления сообщения, которое соответствует $g_{pq}^{(h)}$, т.е. при выполнении h -го события приема процесс q получает m_h -е сообщение от p . По определению причинно-следственной зависимости мы имеем $f_{pq}^{(m_h)} \preceq g_{pq}^{(h)}$.

Поскольку каждое сообщение на протяжении вычисления S принимается однократно, все m_h различны, откуда следует, что хотя бы одно из чисел $m_1, \dots, m_i$ будет не меньше чем i . Выберем такое $j \leq i$, что $m_j \geq i$. Тогда выполняются соотношения $f_{pq}^{(i)} \preceq f_{pq}^{(m_j)} \preceq g_{pq}^{(j)} \preceq g_{pq}^{(i)}$.

□

Теорема 6.20. *Фазовый алгоритм (алгоритм 6.6) — это волновой алгоритм.*

Доказательство. Поскольку каждый процесс отправляет не более D сообщений по каждому каналу, алгоритм завершает работу после конечного числа шагов. Обозначим символом γ заключительную конфигурацию вычисления S нашего алгоритма и будем предполагать, что в S есть хотя бы один инициатор (их может оказаться и несколько).

Чтобы убедиться в том, что в γ каждый процесс принял решение, мы сначала покажем, что каждый процесс хотя бы один раз отправлял сообщения. Ввиду того что в конфигурации γ в каналах нет сообщений, для каждого канала qr справедливо равенство $Rec_p[q] = Sent_q$. Кроме того, поскольку каждый процесс отправляет сообщения самое позднее после того, как некоторое сообщение было

им получено, имеем $Rec_p[q] > 0 \implies Sent_p > 0$. Тогда исходя из предположения о существовании хотя бы одного инициатора p_0 , для которого $Sent_{p_0} > 0$, приходим к выводу о том, что $Sent_p > 0$ для каждого p .

Далее будет показано, что каждый процесс принял решение. Рассмотрим процесс p , у которого в конфигурации γ переменная $Sent$ имеет наименьшее значение, т.е. в γ для всех q выполняется неравенство $Sent_q \geq Sent_p$. В частности, это неравенство справедливо для процесса q , являющегося соседом процесса p на входе, и тогда из равенства $Rec_p[q] = Sent_q$ вытекает неравенство $\min_q Rec_p[q] \geq Sent_p$. Но это означает, что $Sent_p = D$; в противном случае процесс p пришлось бы отправлять дополнительное сообщение, как только он в последний раз получил некоторое сообщение. Значит, $Sent_p = D$ для всех p , и, следовательно, для всех каналов qr имеет место равенство $Rec_p[q] = D$. Отсюда заключаем, что каждый процесс принял решение.

Остается показать, что всякому решению предшествует хотя бы одно событие в каждом из процессов. Рассмотрим путь в сети $P = p_0, p_1, \dots, p_l$, где $l \leq D$. Тогда по лемме 6.19 соотношение

$$f_{p_i p_{i+1}}^{(i+1)} \preceq g_{p_i p_{i+1}}^{(i+1)}$$

справедливо для любого i , $0 \leq i < l$, и в соответствии с описанием алгоритма соотношение

$$g_{p_i p_{i+1}}^{(i+1)} \preceq f_{p_{i+1} p_{i+2}}^{(i+2)}$$

имеет место для любого i , $0 \leq i < l - 1$. Следовательно, $f_{p_0 p_1}^{(1)} \preceq g_{p_{l-1} p_l}^{(l)}$. Так как диаметр сети равен D , для каждой пары процессов q и p существует путь $q = p_0, p_1, \dots, p_l = p$, длина которого не превосходит D . Таким образом, для каждого q найдется такое $l \leq D$ и такой сосед r на входе процесса p , что имеет место соотношение $f_{qq'}^{(1)} \preceq g_{rp}^{(l)}$. А в силу устройства нашего алгоритма событие $g_{rp}^{(l)}$ предшествует d_p . □

В алгоритме по каждому каналу передается D сообщений, и поэтому его сложность по числу обменов сообщениями равна $|E|D$. Нужно иметь в виду, однако, что здесь $|E|$ обозначает количество *односторонних* каналов. Если наш алгоритм используется для неориентированной сети, то каждый ее канал следует рассматривать как два односторонних канала, и поэтому сложность по числу сообщений будет равна $2|E|D$.

Фазовый алгоритм для клик. Если сеть является кликой, то ее диаметр равен 1. В таком случае от каждого соседа должно быть получено в точности одно сообщение и каждому процессу достаточно подсчитать общее число полученных сообщений, вместо того чтобы подсчитывать по отдельности сообщения, полученные от каждого соседа на выходе (см. алгоритм 6.7). При этом сложность по числу сообщений будет равна $N(N - 1)$ и алгоритму будет хватать всего лишь $O(\log N)$ битов внутренней памяти.

```

var  $Rec_p$  :  $0..N - 1$  init 0 ;
      (* Число полученных сообщений *)
   $Sent_p$  :  $0..1$  init 0 ;
      (* Число сообщений, отправленных каждому соседу *)

begin if  $p$  is initiator then
  begin forall  $r \in Neigh_p$  do send  $\langle tok \rangle$  to  $r$  ;
     $Sent_p := Sent_p + 1$ 
  end;
  while  $Rec_p < \#Neigh_p$  do
    begin receive  $\langle tok \rangle$ 
       $Rec_p := Rec_p + 1$  ;
      if  $Sent_p = 0$  then
        begin forall  $r \in Neigh_p$  do send  $\langle tok \rangle$  to  $r$  ;
           $Sent_p := Sent_p + 1$ 
        end
      end;
    decide
  end

```

Алгоритм 6.7. Фазовый алгоритм для клик

6.2.6. Алгоритм Финна

Алгоритм Финна (см. [79]) — это еще один волновой алгоритм, который можно использовать для произвольных ориентированных сетей. Здесь не требуется, чтобы диаметр сети был известен заранее, но зато этот алгоритм опирается на однозначную идентифицируемость процессов. В сообщениях процессы обмениваются отличительными признаками, и это приводит к тому, что битовая сложность алгоритма становится достаточно большой.

В процессе p формируются два множества отличительных признаков Inc_p и $NInc_p$. Если говорить неформально, то Inc_p обозначает множество таких процессов q , что некоторое событие в q предшествует самому последнему событию, случившемуся в p , а $NInc_p$ обозначает множество таких процессов q , что у каждого соседа r процесса q какое-нибудь событие в r предшествует самому последнему событию, случившемуся в p . Формирование этих множеств осуществляется так. Вначале $Inc_p = \{p\}$ и $NInc_p = \emptyset$. Процесс p отправляет сообщения, в которых содержатся Inc_p и $NInc_p$, всякий раз, когда одно из этих множеств расширяется. Когда p получает сообщение, содержащее множество Inc и множество $NInc$, полученные отличительные признаки добавляются к соответствующим множествам, которые формирует процесс p . Если процесс p получил сообщения от всех своих соседей на входе, то отличительный признак p вставляется в $NInc_p$. Когда эти два множества станут одинаковыми, p принимает решение (см. алгоритм 6.8). Исходя из содержательного смысла, который приписан этим множествам, это будет означать, что, каков бы ни был процесс q , если некоторое событие в q предшествует d_p , то у всякого соседа r процесса q также произошло какое-нибудь

событие, предшествующее d_p . Отсюда следует, что для нашего алгоритма требование зависимости соблюдено.

По ходу доказательства корректности алгоритма будет показано, что все это свойственно каждому процессу p , и равенство двух указанных множеств означает, что решению предшествует какое-нибудь событие в каждом процессе.

```

var  $Inc_p$  : set of processes init  $\{p\}$  ;
       $NInc_p$  : set of processes init  $\emptyset$  ;
       $rec_p[q]$  : bool for  $q \in In_p$  init false ;
      (* индикаторы получения процессом  $p$  сообщения от  $q$  *)

begin if  $p$  is initiator then
  forall  $r \in Out_p$  do send  $\langle sets, Inc_p, NInc_p \rangle$  to  $r$  ;
  while  $Inc_p \neq NInc_p$  do
    begin receive  $\langle sets, Inc, NInc \rangle$  from  $q_0$  ;
       $Inc_p := Inc_p \cup Inc$  ;  $NInc_p := NInc_p \cup NInc$  ;
       $rec_p[q_0] := true$  ;
      if  $\forall q \in In_p : rec_p[q]$  then  $NInc_p := NInc_p \cup \{p\}$  ;
      if  $Inc_p$  or  $NInc_p$  has changed then
        forall  $r \in Out_p$  do send  $\langle sets, Inc_p, NInc_p \rangle$  to  $r$ 
      end;
    decide
  end

```

Алгоритм 6.8. Алгоритм Финна

Теорема 6.21. Алгоритм Финна (алгоритм 6.8) является волновым алгоритмом.

До к а з а т е л ь с т в о . Заметим, что два множества, формируемые каждым процессом, могут только увеличиваться. Поскольку суммарный размер этих двух множеств варьируется от 1 в первом сообщении, передаваемом по каждому каналу, до не более чем $2N$ в последнем сообщении, общее количество сообщений ограничено величиной $2N|E|$. Рассмотрим вычисление C , в котором есть хотя бы один инициатор, и его последнюю конфигурацию γ , которая является заключительной. Так же как и в доказательстве теоремы 6.20, можно показать, что если процесс p сумел отправить хотя бы одно сообщение (каждому соседу) и q — сосед процесса p на выходе, то q также сумел отправить по меньшей мере одно сообщение. Отсюда следует, что каждый процесс сумел отправить хотя бы одно сообщение (по каждому каналу).

Теперь будет показано, что в γ каждый процесс принял решение. Во-первых, для каждого ребра pq в конфигурации γ выполняется включение $Inc_p \subseteq Inc_q$. И в самом деле, совершив последнее изменение множества Inc_p , процесс p отправил сообщение $\langle sets, Inc_p, NInc_p \rangle$. Как только оно было получено, процесс q выполнил оператор $Inc_q := Inc_q \cup Inc_p$. Сильная связность сети приводит к тому, что $Inc_p = Inc_q$ для всех p и q . Коль скоро $p \in Inc_p$ и каждое множество Inc

содержит только отличительные признаки процессов, для каждого процесса p в конфигурации γ выполняется равенство $Inc_p = \mathbb{P}$.

Во-вторых, аналогичным образом можно показать, что для каждой пары процессов p и q выполняется равенство $NInc_p = NInc_q$. Поскольку каждый процесс успел отправить хотя бы одно сообщение по каждому каналу, для любого процесса p выполняется требование $\forall q \in In_p : rec_p[q]$. Следовательно, для каждого p верно включение $p \in NInc_p$. А так как множества $NInc$ содержат только отличительные признаки процессов, отсюда следует, что $NInc_p = \mathbb{P}$ для каждого процесса p . Из равенств $Inc_p = \mathbb{P}$ и $NInc_p = \mathbb{P}$ вытекает, что $Inc_p = NInc_p$, и это означает, что в конфигурации γ каждый процесс p принял решение.

Теперь нужно показать, что решению d_p в процессе p предшествует какое-нибудь событие в каждом из процессов. Для всякого события e в процессе p обозначим $Inc^{(e)}$ (соответственно $NInc^{(e)}$) значение Inc_p (соответственно $NInc_p$) сразу после осуществления e (см. доказательство теоремы 6.12). Следующие два утверждения формально обосновывают тот содержательный смысл, который был придан этим множествам в начале этого параграфа.

Утверждение 6.22. Если существует событие $e \in C_q : e \preceq f$, то $q \in Inc^{(f)}$.

Доказательство. Как и в доказательстве теоремы 6.12, можно показать, что соотношение $e \preceq f$ влечет $Inc^{(e)} \subseteq Inc^{(f)}$. Учитывая соотношение $e \in C_q \implies q \in Inc^{(e)}$, отсюда получаем наше утверждение. $\square$

Утверждение 6.23. Если $q \in NInc^{(f)}$, то для всех $r \in In_q$ существует такое событие $e \in C_r$, что $e \preceq f$.

Доказательство. Пусть a_q обозначает внутреннее событие в процессе q , связанное с первым выполнением оператора $NInc_q := NInc_q \cup \{q\}$ процессом q . Событие a_q — это единственное событие, для которого выполняется включение $q \in NInc^{(a_q)}$ и которому не предшествует другое событие a' , удовлетворяющее условию $q \in NInc^{(a')}$. Значит, $q \in NInc^{(f)} \implies a_q \preceq f$. Из описания алгоритма видно, что для каждого $r \in In_q$ найдется событие $e \in C_r$, предшествующее a_q . Отсюда следует доказываемое утверждение. $\square$

Процесс p принимает решение только тогда, когда $Inc_p = NInc_p$; в наших обозначениях $Inc^{(d_p)} = NInc^{(d_p)}$. Теперь мы видим, что

- 1) $p \in Inc^{(d_p)}$;
- 2) из $q \in Inc^{(d_p)}$ следует, что $q \in NInc^{(d_p)}$, и отсюда получаем $In_q \subseteq Inc^{(d_p)}$.

Поскольку сеть сильно связанная, мы получаем искомое равенство $Inc^{(d_p)} = \mathbb{P}$. $\square$

6.3. Алгоритмы обхода

В этом параграфе мы обсудим особый класс волновых алгоритмов, в которых все события по ходу волны линейно упорядочены по отношению причинно-следственной зависимости, и при этом последнее событие происходит в том же самом процессе, что и первое.

Определение 6.24. Алгоритмом обхода называется алгоритм, обладающий следующими тремя свойствами.

1. В каждом вычислении есть единственный инициатор, который запускает алгоритм, отправляя одно-единственное сообщение.
2. Всякий процесс после получения сообщения либо отправляет ровно одно сообщение, либо принимает решение.
3. Как следует из первых двух свойств, в каждом конечном вычислении в точности один процесс принимает решение. Говорят, что алгоритм *завершает работу* в том единственном процессе, который принимает решение.
3. Алгоритм завершает работу в инициаторе, и к тому моменту, когда это происходит, каждому процессу хотя бы один раз удалось отправить сообщение.

В каждой достижимой конфигурации алгоритма обхода есть либо в точности одно сообщение, находящееся на этапе пересылки, либо в точности один процесс, который только что принял сообщение и еще не успел отправить ответное сообщение. С более абстрактной точки зрения все вместе взятые сообщения в вычислении можно рассматривать как единственный объект (*маркер*), который переходит от одного процесса к другому и «посещает», таким образом, все процессы. В параграфе 7.5.4 алгоритмы обхода применяются для построения алгоритмов голосования, а для такого построения важно знать не только общее число переходов маркера по ходу одной волны, но и сколько переходов необходимо сделать маркеру, чтобы посетить первые x процессов.

Определение 6.25. Алгоритм называется *алгоритмом f -обхода* (для некоторого класса сетей), если выполнены следующие условия:

- 1) он является алгоритмом обхода (для указанного класса);
- 2) в каждом вычислении после совершения $f(x)$ переходов маркер побывал не менее чем у $\min(N, x + 1)$ процессов.

var rec_p : integer init 0; (* только для инициатора *)

Для инициатора:

```
(* Записать  $Neigh_p = \{q_1, q_2, \dots, q_{N-1}\}$  *)
begin while  $rec_p < \#Neigh_p$  do
  begin send tok to  $q_{rec_p+1}$ ;
    receive tok;  $rec_p := rec_p + 1$ 
  end;
  decide
end
```

Для неинициатора:

```
begin receive tok from  $q$ ; send tok to  $q$  end
```

Алгоритм 6.9. Алгоритм последовательного опроса

Кольцевой алгоритм (алгоритм 6.1) является алгоритмом обхода, и, поскольку после x переходов маркер побывал у $x + 1$ процессов и посетил все процессы по окончании N -го перехода, этот алгоритм является также алгоритмом x -обхода для кольцевых сетей.

6.3.1. Обход клик

Всякую клику можно обойти, воспользовавшись *последовательным опросом*; этот алгоритм очень похож на алгоритм 6.5, однако соседи инициатора опрашиваются здесь поочередно, и как только поступает отклик от одного соседа, проводится опрос следующего (см. алгоритм 6.9).

Теорема 6.26. *Алгоритм последовательного опроса (алгоритм 6.9) — это алгоритм $2x$ -обхода для клик.*

Доказательство. Легко видеть, что если алгоритм завершается в инициаторе, то каждому процессу удалось прислать отклик. Запросом к процессу q_x служит $(2x - 1)$ -е сообщение, а откликом на него является $2x$ -е сообщение. Следовательно, если произошел обмен $2x$ сообщениями, то посетить удалось $x + 1$ процессов $p, q_1, \dots, q_x$. $\square$

6.3.2. Обход торов

Торический граф граф размера $n \times n$ — это граф $G = (V, E)$, у которого

$$V = \mathbb{Z}_n \times \mathbb{Z}_n = \{(i, j) : 0 \leq i, j < n\}$$

и

$$E = \{(i, j)(i', j') : (i = i' \wedge j = j' \pm 1) \vee (i = i' \pm 1 \wedge j = j')\}$$

(см. § Б.2.4). Сложение и вычитание здесь проводится по модулю n . Предполагается, что в торе имеется восприятие направления (см. § Б.4.3), т. е. в вершине (i, j) канал, соединяющий эту вершину с $(i, j + 1)$, помечен *Up*, канал, ведущий в вершину $(i, j - 1)$, помечен *Down*, канал, ведущий в вершину $(i + 1, j)$, помечен *Right*, а канал, ведущий в вершину $(i - 1, j)$, помечен *Left*. Пары координат (i, j) служат удобным средством задания топологии сети и ориентации, однако мы будем считать, что процессы не располагают сведениями об этих координатах; их представление о топологии ограничено метками каналов.

Тор является гамильтоновым графом, т. е. в торе произвольного размера существует гамильтонов цикл, и маркер перемещается по этому циклу, руководствуясь алгоритмом 6.10. После k -го перехода маркер отправляется вверх, если $n|k$ (n является делителем k); в противном случае он отправляется направо.

Теорема 6.27. *Алгоритм тора (алгоритм 6.10) является алгоритмом x -обхода для тора.*

Доказательство. Как видно из описания алгоритма, решение принимается, как только маркер был передан n^2 раз. Если маркер переместился от

Для инициатора, выполнить один раз:

send $\langle \text{num}, 1 \rangle$ to Up

Для каждого процесса после приема маркера $\langle \text{num}, k \rangle$:

begin if $k = n^2$ **then decide**

else if $n|k$ **then** send $\langle \text{num}, k + 1 \rangle$ to Up

else send $\langle \text{num}, k + 1 \rangle$ to $Right$

end

Алгоритм 6.10. Алгоритм обхода для тора

процесса p к процессу q , совершив U переходов Up и R переходов $Right$, то $p = q$ тогда и только тогда, когда $(n|U \wedge n|R)$. Обозначим процесс-инициатор символом p_0 , а процесс, получивший маркер $\langle \text{num}, k \rangle$, обозначим символом p_k .

Из общего числа n^2 перемещений маркера n ходов были сделаны вверх, а остальные $n^2 - n$ ходов были сделаны направо. Поскольку оба числа n и $n^2 - n$ делятся на n , получаем, что $p_{n^2} = p_0$, и это означает, что алгоритм завершается в инициаторе.

Далее будет показано, что все процессы, начиная от p_0 и оканчивая p_{n^2-1} , попарно различны. А так как у нас всего n^2 процессов, отсюда следует, что посетить удалось каждый процесс. Допустим, $p_a = p_b$ для некоторых $a, b, 0 \leq a < b < n^2$. На участке от p_a до p_b маркер совершил некоторое количество переходов Up и некоторое количество переходов $Right$, и, поскольку $p_a = p_b$, оба числа делятся на n . Обратившись к описанию алгоритма, можно заметить, что это влечет за собой два соотношения

$$\#\{k : a \leq k < b \wedge n|k\} \text{ кратно } n \quad \text{и} \quad \#\{k : a \leq k < b \wedge n \nmid k\} \text{ кратно } n.$$

Суммарный размер этих множеств равен $b - a$, откуда следует, что $n|(b - a)$. Представим это отношение в виде $(b - a) = ln$. Тогда множество $\{k : a \leq k < b\}$ содержит l чисел, делящихся на n . Отсюда следует, что $n|l$, и поэтому $n^2|(b - a)$, что является противоречием. Так как все процессы, начиная с p_0 и оканчивая p_{n^2-1} , попарно различны, маркеру после x переходов удастся посетить $x + 1$ процессов. $\square$

6.3.3. Обход гиперкубов

Граф $G = (V, E)$, в котором

$$V = \{(b_0, \dots, b_{n-1}) : b_i = 0, 1\}$$

и

$$E = \{(b_0, \dots, b_{n-1}), (c_0, \dots, c_{n-1}) : \text{наборы } b \text{ и } c \text{ отличаются в одном бите}\}$$

называется n -мерным *гиперкубом* (см. § Б.2.5). Предполагается, что в гиперкубе имеется восприятие направления (см. § Б.4.3), т. е. канал между вершинами

b и c , отличающимися в i -м бите, помечен « i » в обеих вершинах. Предполагается также, что процессы не располагают сведениями о пометках вершин; их представление о топологии ограничено знанием меток каналов.

Как и в случае тора, гиперкуб является гамильтоновым графом, и гамильтонов цикл можно обойти, воспользовавшись алгоритмом 6.11. Доказательство корректности этого алгоритма почти такое же, как для алгоритма 6.10.

Для инициатора выполнить один раз:
send $\langle \text{num}, 1 \rangle$ по каналу $n - 1$.

Для каждого процесса после приема маркера $\langle \text{num}, k \rangle$:
begin if $k = 2^n$ **then** *decide*
 else begin let l — наибольшее такое число, что $2^l | k$;
 send $\langle \text{num}, k + 1 \rangle$ по каналу l
 end
end

Алгоритм 6.11. Алгоритм обхода для гиперкуба

Теорема 6.28. Алгоритм гиперкуба (алгоритм 6.11) является алгоритмом x -обхода для гиперкуба.

Доказательство. Как видно из описания алгоритма, решение принимается после 2^n переходов маркера. Обозначим символом p_0 инициатор, и пусть p_k будет обозначать процесс, к которому поступил маркер $\langle \text{num}, k \rangle$. Для всякого $k < 2^n$ наборы, обозначающие вершины p_k и p_{k+1} , отличаются в одном бите, индекс которого обозначим $l(k)$. Этот индекс удовлетворяет соотношению

$$l(k) = \begin{cases} n - 1, & \text{если } k = 0, \\ \text{наибольшее такое } l, \text{ что } 2^l | k, & \text{если } k > 0. \end{cases}$$

Так как для каждого $i < n$ существует четное число таких индексов $k \in \{0, \dots, 2^n\}$, что $l(k) = i$, мы заключаем, что $p_0 = p_{2^n}$ и решение принимается инициатором. Проводя рассуждения, сходные с теми, которые применялись для доказательства теоремы 6.27, приходим к выводу о том, что равенство $p_a = p_b$ влечет условие $2^n | (b - a)$. Отсюда следует, что все процессы $p_0, \dots, p_{2^n-1}$ попарно различны.

Таким образом, к моменту завершения вычисления маркеру удалось посетить все процессы. При этом, совершив x переходов, маркеру удастся посетить $x + 1$ процессов. $\square$

6.3.4. Обход связных сетей

Алгоритм обхода произвольных связных сетей был предложен Тарри в 1895; см. [183]. Этот алгоритм определяется двумя следующими правилами (см. алгоритм 6.12).

R1. Процесс не передает маркер по одному и тому же каналу дважды.

R2. Неинициатор передает маркер своей *родительской вершине* (соседу, от которого он впервые получил маркер) только в том случае, когда его невозможно передать по другим каналам согласно правилу R1.

```

var  $used_p[q]$  : bool    init false for each  $q \in Neigh_p$  ;
                                (* Показывает, отправлял ли  $p$  сообщение процессу  $q$  *)
     $father_p$  : process init undef ;

```

Только для инициатора, выполнить один раз:
begin $father_p := p$; choose $q \in Neigh_p$;
 $used_p[q] := true$; send **tok** to q
end

Для каждого процесса после приема **tok** от q_0 :
begin if $father_p = undef$ **then** $father_p := q_0$;
 if $\forall q \in Neigh_p : used_p[q]$ **then** *decide*
 else if $\exists q \in Neigh_p : (q \neq father_p \wedge \neg used_p[q])$
 then begin choose $q \in Neigh_p \setminus \{father_p\}$
 with $\neg used_p[q]$;
 $used_p[q] := true$; send **tok** to q
 end
 else begin $used_p[father_p] := true$;
 send **tok** to $father_p$
 end
end

Алгоритм 6.12. Алгоритм обхода Тарри

Теорема 6.29. Алгоритм Тарри (алгоритм 6.12) является алгоритмом обхода.

Доказательство. Так как маркер передается не более чем по одному разу в каждом из двух направлений по любому каналу, он совершает не более $2|E|$ переходов, пока алгоритм не завершится. Поскольку каждый процесс отправляет маркер по каждому каналу не более одного раза, он его и получает по каждому каналу не более одного раза. Всякий раз, когда маркером владеет неинициатор p , это означает, что процессу p пришлось получать маркер на один раз больше, чем отправлять его. Отсюда следует, что число каналов, инцидентных p , превосходит число каналов, использованных этим процессом, по крайней мере на 1, и поэтому p не принимает решения, а передает маркер. Значит, решение может принять только инициатор.

Далее мы обоснуем три тезиса, из которых следует, что всякий раз, когда алгоритм завершает работу, каждому процессу пришлось участвовать в передаче маркера.

1. Все каналы, инцидентные инициатору, были использованы по одному разу в каждом направлении. Каждый канал был использован инициатором

для передачи маркера, ибо в противном случае алгоритм не завершается. Инициатору пришлось получать маркер столько же раз, сколько и отправлять его. А поскольку получать его приходилось по разным каналам, маркер был получен по одному разу с использованием каждого канала.

2. Для каждого посещенного маркером процесса p все каналы, инцидентные p , были использованы для передачи маркера в каждом направлении по одному разу. Предположим, что это не так. Пусть p — самый первый из посещенных маркером процессов, для которого не выполняется это предположение. Заметим, что согласно тезису 1) таким процессом не может быть инициатор. Согласно выбору p все каналы, инцидентные $father_p$, были использованы по одному разу в каждом направлении. Отсюда следует, что p уже отправил маркер в родительскую вершину. Но это означает, что p использовал все инцидентные каналы для отправления маркера. А поскольку маркер вернулся к инициатору, процессу p пришлось получать его столь же часто, сколь и отправлять. Но в таком случае по каждому инцидентному каналу процесс p получал маркер по одному разу. А это противоречит сделанному предположению.

3. Маркер посетил все процессы и передавался по каждому каналу в обоих направлениях. Если бы нашлись процессы, которые маркеру не удалось посетить, то это означало бы, что для некоторой пары соседних процессов p и q маркеру удалось побывать в p , но не удалось посетить q . Но это противоречит тому, что процессу p пришлось использовать каждый канал в обоих направлениях. Значит, маркер побывал во всех процессах, и все каналы были задействованы по одному разу в каждом направлении согласно тезису 2). $\square$

Каждое вычисление алгоритма Тарри задает в сети остовное дерево, о чем свидетельствует лемма 6.3. Корнем дерева служит инициатор, и каждый неинициатор p к моменту завершения вычисления уже заносит в переменную $father_p$ ссылку на родительскую вершину в этом дереве. Если требуется, чтобы каждый процесс обладал (к моменту завершения вычисления) сведениями о том, какие из его соседей являются сыновними вершинами в этом дереве, то этого можно достичь, отправляя специальное сообщение процессу $father_p$.

6.4. Сложность по времени: поиск в глубину

Процессам в алгоритме Тарри предоставлена достаточная свобода выбора соседей, которым нужно передать маркер, и в результате может образоваться большой класс остовных деревьев. В этом параграфе мы рассмотрим алгоритмы построения таких остовных деревьев, что в них каждое стягивающее ребро соединяет две вершины, одна из которых выступает в роли предшественника другой. Стягивающее ребро — это ребро, не принадлежащее остовному дереву. Для заданного остовного дерева T сети G и для каждой его вершины p мы будем использовать запись $T[p]$ для обозначения множества процессов поддерева, растущего из p , а запись $A[p]$ — для обозначения множества предшественников p , т. е. вершин, лежащих в дереве T на пути из корня в p . Заметим, что $q \in T[p] \iff p \in A[q]$.

Определение 6.30. Корневое остовное дерево T сети G называется *деревом поиска в глубину*, если для каждого стягивающего ребра pq выполняется соотношение $q \in T[p] \vee q \in A[p]$.

Деревья поиска в глубину применяются во многих графовых алгоритмах, например проверки планарности, проверки двусвязности, построения интервальных схем разметки (см. § 4.4.2). В § 6.4.1 будет показано, что незначительная модификация алгоритма Тарри (связанная с ограничением свободы выбора процессов) позволяет применять его для построения деревьев поиска в глубину. Полученный таким образом алгоритм мы назовем *классическим алгоритмом поиска в глубину*. В § 6.4.2 будут рассмотрены два алгоритма, позволяющие строить деревья поиска в глубину за меньшее время по сравнению с классическим алгоритмом. Для распределенных алгоритмов понятие сложности по времени будет определено ниже. В § 6.4.3 будет предложен алгоритм поиска в глубину для сетей при наличии предварительных сведений об отличительных признаках соседей. В этом случае теорема 6.6 уже будет не применима, и по сути дела можно построить алгоритм, использующий всего лишь $2N - 2$ сообщений.

Сложность по времени для распределенных алгоритмов. Время, которое требуется для передачи сообщения в асинхронных сетях, заранее не известно, и нам не хотелось бы ставить наши методы анализа в зависимость от параметров отдельных систем. Нужно вспомнить, что в последовательных вычислениях сложность по времени не измеряется в единицах физического времени, а определяется числом выполненных команд. Здесь мы сделаем то же самое. В качестве единицы измерения продолжительности распределенного вычисления возьмем самую большую задержку передачи сообщения в ходе вычисления. Такая единица времени будет задаваться аксиомой, гласящей, что задержка передачи всякого сообщения ограничена этой единицей времени. Поэтому изложенный ниже тезис T2 нужно в действительности рассматривать не как допущение о вычислении, а как утверждение об этой единице времени.

Определение 6.31. Сложностью по времени для распределенного алгоритма называется наибольшее время, которое требуется для выполнения произвольного вычисления этого алгоритма с учетом следующих допущений.

T1. Во всяком процессе любое конечное число событий может осуществиться мгновенно (за время 0).

T2. Период времени между отправлением и получением одного и того же сообщения не превышает одной единицы времени.

Сложность по времени для всех волновых алгоритмов, описанных в этой главе, приведена в таблице 6.18. Мы не приводим обоснования тех значений, которые указаны в таблице; читателю предлагается в качестве упражнения проверить эти значения самостоятельно. Альтернативные определения сложности по времени будут обсуждаются в § 6.5.3.

Лемма 6.32. Для алгоритмов обхода сложность по времени равна сложности по числу сообщений.

Доказательство. Обмен сообщениями осуществляется последовательно, и для каждого обмена может потребоваться одна единица времени. $\square$

6.4.1. Распределенный поиск в глубину

Классический алгоритм поиска в глубину получается из алгоритма Тарри, в котором свобода выбора очередного соседа для передачи ему маркера ограничена третьим правилом следующего вида (см. алгоритм 6.13).

R3. Всякий раз, когда процесс получает маркер, он возвращает его назад по тому же каналу, если это не противоречит правилам R1 и R2.

```

var  $used_p[q]$  : bool   init false for each  $q \in Neigh_p$  ;
                        (* Указывает, отправлял ли  $p$  маркер процессу  $q$  *)
   $father_p$       : process init undef ;

```

Только для инициатора, выполнить один раз:

```

begin  $father_p := p$  ; choose  $q \in Neigh_p$  ;
       $used_p[q] := true$  ; send tok to  $q$ 
end

```

Для каждого процесса, при получении **tok** от q_0 :

```

begin if  $father_p = undef$  then  $father_p := q_0$  ;
      if  $\forall q \in Neigh_p : used_p[q]$  then decide
      else if  $\exists q \in Neigh_p : (q \neq father_p \wedge \neg used_p[q])$ 
        then begin if  $father_p \neq q_0 \wedge \neg used_p[q_0]$  then  $q := q_0$ 
                  else choose  $q \in Neigh_p \setminus \{father_p\}$ 
                        with  $\neg used_p[q]$  ;
                   $used_p[q] := true$  ; send tok to  $q$ 
                end
      else begin  $used_p[father_p] := true$  ;
                send tok to  $father_p$ 
              end
end

```

Алгоритм 6.13. Классический алгоритм поиска в глубину

Теорема 6.33. *Классический алгоритм поиска в глубину (алгоритм 6.13) строит остовное дерево поиска в глубину за $2|E|$ единиц времени, используя при этом $2|E|$ обменов сообщениями.*

Доказательство. Поскольку в основу нашего алгоритма положен алгоритм Тарри, мы имеем дело с алгоритмом обхода, который вычисляет остовное дерево T . Как было уже установлено, по каждому каналу передаются два сообщения (по одному в каждом направлении), и это служит обоснованием сложности по числу сообщений. А сложность по времени равна $2|E|$, поскольку обмен сообщениями проводится поочередно и каждый обмен требует не более одной еди-

ницы времени. Остается убедиться в том, что дерево, построенное в результате применения правила R3, будет деревом поиска в глубину.

Во-первых, согласно правилу R3 вслед за первым прохождением по стягивающему ребру немедленно следует повторное прохождение в обратном направлении. Предположим, что pq — это стягивающее ребро и p является первым процессом, воспользовавшимся этим стягивающим ребром. К тому моменту, когда процесс q получает маркер от p , этот маркер уже побывал в q (в противном случае процесс q занес бы в переменную $father_q$ ссылку на p и наше ребро не могло бы считаться стягивающим) и $used_q[p]$ имеет значение false (поскольку по предположению процесс p был первым из этих двух процессов, воспользовавшимся указанным ребром). Следовательно, согласно правилу R3, процесс q немедленно возвращает маркер процессу p .

Теперь можно показать, что если ребро pq является стягивающим и его впервые использовал процесс p , то $q \in A[p]$. Рассмотрим путь, по которому следовал маркер, до тех пор пока он не был отправлен по ребру pq . Коль скоро pq — стягивающее ребро, маркер уже побывал в вершине q до того момента, когда он достиг q по указанному ребру:

$$\dots, q, \dots, p, q.$$

Образум из рассматриваемого пути как можно более короткий путь, заменяя все фрагменты вида r_1, r_2, r_1 , где $r_1 r_2$ — стягивающее ребро, фрагментом r_1 . Согласно приведенным выше соображениям все стягивающие ребра будут таким образом удалены. Отсюда следует, что полученный в результате путь будет путем в T , состоящим только из ребер, использованных прежде первого обращения к pq . Если q не относится к числу предшественников процесса p , то это означает, что ребро из q в $father_q$ было задействовано ранее, нежели было использовано ребро pq , вопреки правилу R2 нашего алгоритма. $\square$

Сложность по числу сообщений классического распределенного поиска в глубину равна $2|E|$, и, как следует из теоремы 6.6, эта оценка является оптимальной (с точностью до постоянного множителя 2), если только отличительные признаки соседей заранее неизвестны. Сложность по времени также равна $2|E|$, и это наилучшее из того, что могут в этом случае дать алгоритмы обхода (см. лемму 6.32). Распределенный вариант поиска в глубину был впервые предложен Чуном [51].

В § 6.4.2 будут рассмотрены два алгоритма, позволяющие строить в сети дерево поиска в глубину без предварительных сведений об отличительных признаках соседей за $O(N)$ единиц времени. Ясно, что эти алгоритмы не являются алгоритмами обхода. В § 6.4.3 мы покажем, как можно воспользоваться сведениями об отличительных признаках соседей, чтобы построить алгоритм, у которого сложность по числу обменов сообщениями и по времени равна $O(N)$.

6.4.2. Алгоритмы поиска в глубину, требующие линейного времени

Классический алгоритм поиска в глубину имеет большую сложность по времени, потому что все ребра сети, как стягивающие ребра, так и ребра, входя-

щие в дерево, обходятся последовательно. Как показано в доказательстве теоремы 6.33, маркер-сообщение **tok** обходит все стягивающие ребра и немедленно возвращается. Всякое решение, приводящее к более низкой сложности по времени, основывается на том соображении, что маркер-сообщение проходит только по ребрам, входящим в дерево. Ясно, что для этого потребуется линейное время, ибо в дереве имеется всего $N - 1$ ребер.

Решение Авербаха. В наш алгоритм теперь будет включен механизм, предотвращающий передачу маркера по стягивающим ребрам. В алгоритме Авербаха [13] это обеспечивается тем, что каждый процесс в тот момент, когда он должен передать маркер, обладает сведениями о том, у какого из его соседей уже побывал маркер. Тогда процесс либо выбирает какого-нибудь соседа, не получавшего до сих пор маркер, либо возвращает маркер в родительскую вершину, если такого соседа нет.

Когда маркер впервые достается процессу p (для инициатора это происходит в момент запуска алгоритма), p оповещает об этом событии каждого своего соседа r , за исключением родительской вершины, отправляя ему сообщение $\langle \text{vis} \rangle$. Передача маркера приостанавливается, до тех пор пока p не получит сообщение $\langle \text{ack} \rangle$ от всех своих соседей. Это служит гарантией того, что каждый сосед r процесса p осведомлен к моменту отправления маркера процессом p о том, что этот маркер уже побывал у p . Когда позднее маркер достигнет r , процесс r уже не будет отправлять маркер процессу p , если только p не является родительской вершиной для r (см. алгоритм 6.14).

```

var  $used_p[q]$  : bool   init false for each  $q \in Neigh_p$  ;
                        (*Указывает, отправлял ли  $p$  маркер процессу  $q$ *)
 $father_p$       : process init undef ;

```

Только для инициатора, выполнить один раз:

```

begin  $father_p := p$  ; choose  $q \in Neigh_p$  ;
      forall  $r \in Neigh_p$  do send  $\langle \text{vis} \rangle$  to  $r$  ;
      forall  $r \in Neigh_p$  do receive  $\langle \text{ack} \rangle$  from  $r$  ;
       $used_p[q] := true$  ; send  $\langle \text{tok} \rangle$  to  $q$ 
end

```

Для каждого процесса после получения $\langle \text{tok} \rangle$ от q_0 :

```

begin if  $father_p = undef$  then
  begin  $father_p := q_0$  ;
        forall  $r \in Neigh_p \setminus \{father_p\}$  do send  $\langle \text{vis} \rangle$  to  $r$  ;
        forall  $r \in Neigh_p \setminus \{father_p\}$  do receive  $\langle \text{ack} \rangle$  from  $r$ 
      end ;
  if  $p$  is the initiator and  $\forall q \in Neigh_p : used_p[q]$ 
    then decide
  else if  $\exists q \in Neigh_p : (q \neq father_p \wedge \neg used_p[q])$ 
    then begin if  $father_p \neq q_0 \wedge \neg used_p[q_0]$ 
      then  $q := q_0$ 
      else choose  $q \in Neigh_p \setminus \{father_p\}$  with  $\neg used_p[q]$  ;
             $used_p[q] := true$  ; send  $\langle \text{tok} \rangle$  to  $q$ 
      end
    else begin  $used_p[father_p] := true$  ; send  $\langle \text{tok} \rangle$  to  $father_p$  end
  end
end

```

Для каждого процесса после получения $\langle \text{vis} \rangle$ от q_0 :

```

begin  $used_p[q_0] := true$  ; send  $\langle \text{ack} \rangle$  to  $q_0$  end

```

Алгоритм 6.14. Алгоритм Авербаха поиска в глубину

Обмен сообщениями часто приводит к тому, что элементу массива $used_p[father_p]$ присваивается значение **true**, даже когда процесс p еще не отправлял маркер в свою родительскую вершину. Поэтому в алгоритме должно быть явно указано, что решение может принимать только инициатор; неинициатор p , у которого $used_p[q]$ имеет значение **true** для всех соседей q , отправляет маркер в свою родительскую вершину.

Теорема 6.34. *Алгоритм Авербаха (алгоритм 6.14) строит дерево поиска в глубину за $4N - 2$ единиц времени, проводя при этом $4|E|$ обменов сообщениями.*

Доказательство. Маркер пересылается в точности по тем же каналам, что и в алгоритме 6.13, за исключением стягивающих ребер — по ним передачи маркера не происходит. Поскольку передачи по стягивающим ребрам не влияют на окончательное значение $father_p$ для каждого процесса p , полученное

в результате дерево всегда будет совпадать с одним из возможных результатов алгоритма 6.13. Маркер проходит последовательно в обе стороны по $N - 1$ ребрам дерева, и для этого требуется $2N - 2$ единиц времени. В каждой вершине движение маркера приостанавливается не более одного раза для обмена сообщениями **vis/ack** и это приводит к задержке в каждой вершине на две единицы времени.

По каждому стягивающему ребру проходят два сообщения **vis** и два сообщения **ack**. По каждому ребру дерева передаются два сообщения **tok**, одно сообщение **vis** (от родительской вершины к ее преемнику) и одно сообщение **ack** (к родительской вершине от ее преемника). Следовательно, происходит обмен $4|E|$ сообщениями. $\square$

Отправление сообщения **vis** тем соседям, которым процесс собирается передать маркер, может быть опущено. Это улучшение (оно не приведено в алгоритме 6.14) позволяет сэкономить по два сообщения на каждом ребре дерева и, следовательно, сокращает сложность по числу обменов сообщениями на $2N - 2$ сообщения.

```

var  $used_p[q]$  : bool   init false for each  $q \in Neigh_p$  ;
                        (* Указывает, отправлял ли  $p$  маркер процессу  $q$  *)
   $father_p$       : process init udef ;
   $mrs_p$          : process init udef ;

```

Только для инициатора, выполнить один раз:

```

begin  $father_p := p$  ; choose  $q \in Neigh_p$  ;
      forall  $r \in Neigh_p$  do send vis to  $r$  ;
       $used_p[q] := true$  ;  $mrs_p := q$  ; send tok to  $q$ 
end

```

Для каждого процесса после получения **vis** от q_0 :

```

begin  $used_p[q_0] := true$  ;
      if  $q_0 = mrs_p$  then (* Истолковывать как сообщение tok *)
        forward tok message as upon receipt of tok message
      end

```

Алгоритм 6.15. Алгоритм Сидон поиска в глубину (Часть 1)

Решение Сидона. В алгоритме Сидона [53], в отличие от алгоритма Авербаха, не отправляются сообщения **ack**, и за счет этого улучшается сложность по времени. В модифицированном алгоритме Сидона маркер отправляется немедленно, т.е. без той задержки на две единицы времени, которая введена в алгоритме Авербаха для ожидания подтверждений. Тот же самый алгоритм был предложен Лакшмананом и др. [123]. В алгоритме Сидона возможно возникновение следующей ситуации. В процессе p уже побывал маркер, и этот процесс уже отправил сообщение **vis** своему соседу r . Позднее маркер достается процессу r , но в тот момент, когда r получает маркер, сообщение **vis** от p еще не

Для каждого процесса после получения **tok** от q_0 :

```

begin if  $mrs_p \neq udef$  and  $mrs_p \neq q_0$ 
  (* Это стягивающее ребро, истолковывать как сообщение vis *)
  then  $used_p[q_0] := true$ 
else (* Поступать, как в предыдущем алгоритме *)
  begin if  $father_p = udef$  then
    begin  $father_p := q_0$  ;
        forall  $r \in Neigh_p \setminus \{father_p\}$  do
          send vis to  $r$ 
        end ;
    if  $p$  — инициатор and  $\forall q \in Neigh_p : used_p[q]$ 
      then decide
    else if  $\exists q \in Neigh_p : (q \neq father_p \wedge \neg used_p[q])$ 
      then begin if  $father_p \neq q_0 \wedge \neg used_p[q_0]$ 
        then  $q := q_0$ 
        else choose  $q \in Neigh_p \setminus \{father_p\}$ 
              with  $\neg used_p[q]$  ;
           $used_p[q] := true$  ;  $mrs_p := q$  ;
          send tok to  $q$ 
        end
      else begin  $used_p[father_p] := true$  ;
          send tok to  $father_p$ 
        end
      end
    end
  end

```

Алгоритм 6.16. Алгоритм Сидона поиск в глубину (Часть 2)

достигло r . В таком случае r может передать маркер процессу p , отправляя его, таким образом, по стягивающему ребру. (Обратите внимание на то, как сообщения **ack** в алгоритме Авербаха предотвращают подобный сценарий развития событий.)

Чтобы справиться с такой ситуацией, процесс p записывает (в переменную mrs_p) тех соседей, которым он в последнюю очередь отправлял маркер. Когда маркер проходит только по ребрам дерева, процесс p получает в очередной раз маркер от того самого соседа mrs_p . А в том сценарии, который был описан выше, p получает сообщение **tok** от другого соседа, а именно от r ; в таком случае сам маркер просто игнорируется, но p помечает ребро rp как использованное, точно так же, как если бы было получено сообщение **vis** от r . Процесс r получает от p сообщение **vis**, после того как он передал маркер процессу p , т.е. r получает сообщение **vis** от своего соседа mrs_r . В этом случае процесс r ведет себя так, как если бы он вообще не передавал маркер процессу p , а именно, r выбирает следующего соседа и передает ему маркер (см. алгоритм 6.15/6.16).

Теорема 6.35. Алгоритм Сидона (алгоритм 6.15/6.16) строит DFS дерево за $2N - 2$ единицы времени, используя $4|E|$ обменов сообщениями.

Доказательство. И вновь перемещение маркера будет подобно тому движению, которое задается алгоритмом 6.13. Либо его передача по стягивающим ребрам отменяется (как и в алгоритме 6.14), либо маркер идет навстречу сообщению **vis** по стягивающему ребру. В последнем случае процесс, получающий сообщение **vis**, продолжает передавать маркер, что вызывает такой же эффект, как если бы маркер немедленно вернулся обратно по стягивающему ребру.

Промежуток времени между двумя последовательными передачами маркера по ребру дерева ограничен одной единицей. Если маркер отправлен по ребру дерева процессу p в момент времени t , то к моменту времени t все сообщения **vis** от тех соседей q процесса p , в которых маркер побывал ранее, были отправлены, и, следовательно, эти сообщения получены самое позднее в момент времени $t + 1$. Так что, хотя процесс p мог к моменту времени $t + 1$ передавать маркер по стягивающему ребру несколько раз, самое позднее в момент $t + 1$ он уже исправил все эти ошибки и передал маркер по ребру дерева. Коль скоро необходимо совершить $2N - 2$ проходов по ребрам дерева, наш алгоритм завершает работу спустя $2N - 2$ единиц времени.

По каждому каналу передаются не более двух сообщений **vis** и двух сообщений **tok**, и это служит обоснованием тому, что верхняя оценка числа обменов сообщениями равна $4|E|$. $\square$

Во многих случаях наш алгоритм будет отправлять меньшее количество сообщений, нежели алгоритм Авербаха. Проведенный анализ числа сообщений в алгоритме Сидона проведен в предположении о наихудшем случае, а именно когда маркер передается по стягивающим ребрам в обоих направлениях. Можно ожидать, что сообщения **vis** позволят с успехом предотвратить многие нежелательные передачи, и в таком случае по каждому каналу может передаваться только два или три сообщения.

Сидон отмечает, что, хотя в алгоритме маркер может передаваться тем вершинам, в которых он уже побывал, сам алгоритм имеет лучшую сложность по времени (равно как и по числу обменов сообщениями), нежели алгоритм 6.14, и это приводит к предотвращению таких нежелательных передач. Это наводит на мысль о том, что для исправления ненужных действий требуется меньше времени и сообщений, нежели для предотвращения этих действий. Сидон оставил открытым вопрос о том, существует ли алгоритм поиска в глубину, который достигает той же сложности по числу обменов сообщениями, что и классический алгоритм, т. е. $2|E|$, и при этом укладывается в $O(N)$ единиц времени.

6.4.3. Поиск в глубину с учетом сведений о соседях

Если процессы осведомлены об отличительных признаках своих соседей, то передачу маркера по стягивающим ребрам можно предотвратить, включив список пройденных вершин в состав маркера. Процесс p , получив маркер, в который вложен список L , не будет передавать маркер процессам из L . Переменную $used_p[q]$ можно изъять, потому что если процесс p ранее уже передавал маркер процессу q , то $q \in L$ (см. алгоритм 6.17).

```
var  $father_p$  : процесс  initundef ;
```

Только для инициатора, выполнить один раз:

```
begin  $father_p := p$  ; choose  $q \in Neigh_p$  ;
    send  $\langle tlist, \{p\} \rangle$  to  $q$ 
end
```

Для каждого процесса после получения $\langle tlist, L \rangle$ от q_0 :

```
begin if  $father_p = undef$  then  $father_p := q_0$  ;
    if  $\exists q \in Neigh_p \setminus L$ 
    then begin choose  $q \in Neigh_p \setminus L$  ;
        send  $\langle tlist, L \cup \{p\} \rangle$  to  $q$ 
    end
    else if  $p$  is initiator
    then decide
    else send  $\langle tlist, L \cup \{p\} \rangle$  to  $father_p$ 
end
```

Алгоритм 6.17. Поиск в глубину с учетом сведений о соседях.

Теорема 6.36. *DFS-алгоритм с учетом сведений о соседях является алгоритмом обхода, вычисляющим дерево обхода в глубину с использованием $2N - 2$ обменов сообщениями за $2N - 2$ единиц времени.*

Битовая сложность этого алгоритма велика: если w — это число битов, необходимых для задания отличительного признака, то для списка L может потребоваться до Nw битов (см. упражнение 6.14).

6.5. Прочие вопросы

6.5.1. Обзор волновых алгоритмов

В таблице 6.18 приведен список волновых алгоритмов, рассмотренных в этой главе.

В колонке, озаглавленной «номер», приведен тот номер алгоритма, который используется для его обозначения в этой главе; в колонке, озаглавленной «Ц/Д» указано, является ли алгоритм централизованным (Ц) или децентрализованным (Д); в колонке, помеченной «О», указано, является ли данный алгоритм алгоритмом обхода; в колонке, озаглавленной «С», приведена сложность алгоритма по числу обменов сообщениями; в колонке, озаглавленной «В», приведена сложность алгоритма по времени. Здесь N обозначает количество процессов, $|E|$ — количество каналов, а D — диаметр сети (измеряемый по числу переходов).

Сложность образования волны в сетях с наиболее распространенной топологией существенно зависит от того, требуется ли, чтобы алгоритм был централизованным или децентрализованным. В таблице 6.19 перечислены оценки сложности по числу сообщений для централизованных и децентрализованных волновых алгоритмов для колец, произвольных сетей и деревьев. Подобным же образом

Алгоритм	Номер	Топология	Ц/Д	О	С	В
Section 6.6.2. Общие алгоритмы						
Кольцевой	6.1	кольцо	Ц	да	N	N
Древесный	6.2	дерево	Д	нет	N	$O(D)$
Эха	6.4	произвольная	Ц	нет	$2 E $	$O(N)$
Опроса	6.5	клика	Ц	нет	$2N - 2$	2
Фазовый	6.6	произвольная	Д	нет	$2D E $	$2D$
Фазовый	6.7	клика	Д	нет	$N(N - 1)$	2
на кликах						
Финна	6.8	произвольная	Д	нет	$\leq 4N E $	$O(D)$
§ 6.6.3. Алгоритмы обхода						
Последовательного опроса	6.9	клика	Ц	да	$2N - 2$	$2N - 2$
Тора	6.10	тор	Ц	да	N	N
Гиперкуба	6.11	гиперкуб	Ц	да	N	N
Тарри	6.12	произвольная	Ц	да	$2 E $	$2 E $
§ 6.6.4. Алгоритмы поиска в глубину						
Классический	6.13	произвольная	Ц	да	$2 E $	$2 E $
Авербаха	6.14	произвольная	Ц	нет	$4 E $	$4N - 2$
Сидона	6.15/6.16	произвольная	Ц	нет	$4 E $	$2N - 2$
с учетом соседей	6.17	произвольная	Ц	да	$2N - 2$	$2N - 2$

Примечание: фазовый алгоритм (6.6) и алгоритм Финна (6.8) пригодны для ориентированных сетей.

Таблица 6.18. Волновые алгоритмы, рассмотренные в настоящей главе

можно проанализировать зависимость сложности от других параметров, таких как осведомленность о соседях или восприятие направления (см. § Б.4.3).

6.5.2. Вычисление сумм

Как было показано в § 6.1.5, при помощи одиночной волны можно вычислять точную нижнюю грань по входным данным всех процессов. Вычисление точной нижней грани можно применять для выполнения коммутативных, ассоциативных и идемпотентных операций над входными данными, наподобие вычисления минимума, максимума, и т. п. (см. следствие 6.14). Однако большое число функций, наподобие суммирования, так вычислять нельзя, потому что они не идемпотентны. Суммирование можно применять для подсчета количества процессов, обладающих определенными свойствами (полагая, что на вход подается 1, если процесс обладает нужным свойством, и 0 в противном случае), но результаты этого параграфа можно также применять и для других операций, являющихся коммутативными и ассоциативными, таких как произведение целых чисел или объединение множеств.

Топология	Ц/Д	Сложность	Ссылка
Кольцо	Ц	N	алгоритм 6.1
	Д	$O(N \log N)$	алгоритм 7.7
Произвольная	Ц	$2 E $	алгоритм 6.4
	Д	$O(N \log N + E)$	раздел 7.5.3
Дерево	Ц	$2(N - 1)$	алгоритм 6.4
	Д	$O(N)$	алгоритм 6.2

Таблица 6.19. Влияние централизации на сложность по числу сообщений

Оказывается, общего метода для вычисления сумм при помощи волнового алгоритма не существует, но в отдельных случаях вычисление суммы возможно. Это те случаи, когда алгоритм является алгоритмом обхода, когда процессы наделены отличительными признаками, или когда алгоритм порождает остовное дерево, которое может быть использовано.

Невозможность общего построения. Нельзя предложить общую конструкцию для вычисления сумм на основе произвольного волнового алгоритма, подобную той, которая была использована в теореме 6.12 для вычисления точных нижних граней. И вот почему. Есть волновой алгоритм, а именно фазовый алгоритм (с параметром $D = 2$), для определенного класса сетей, включающего все неориентированные безымянные сети диаметра два. Но нет алгоритма, способного вычислять сумму всех входных данных, который делал бы это корректно на всех неориентированных безымянных сетях диаметра два. К этому классу сетей относятся две сети, изображенные на рис. 6.20. Предположим, что на вход каждого процесса поступила 1. Тогда искомым результатом равен 6 для левой сети и 8 для правой сети. Применяя метод, предложенный в гл. 9, можно показать, что всякий алгоритм будет выдавать одинаковый результат на каждой из этих двух сетей, и, значит, он не является одновременно корректным для обеих сетей. Детальную разработку этой аргументации оставляем читателю в качестве упражнения 9.7.

Вычисление суммы с помощью алгоритмов обхода. Если A является алгоритмом обхода, то сумму всех входных данных можно вычислить так. Пусть процесс p обладает переменной j_p , первоначальное значение которой — это входные данные процесса p . Наделим маркер дополнительным полем s . Всякий раз, когда процесс p передает маркер, он выполняет оператор

$$s := s + j_p ; \quad j_p := 0.$$

И тогда можно показать, что каждый раз, когда маркер покидает процесс p , будет выполняться соотношение $j_p = 0$, а s будет иметь значение, равное сумме входных данных для всех ранее пройденных процессов. Следовательно, как только наш алгоритм завершит свое выполнение, s будет равно сумме всех входных данных.

Вычисление суммы при помощи остовного дерева. Некоторые волновые алгоритмы при выполнении процессом p события решения d_p открывают ему до-

ступ к остовному дереву с корнем p , по которому сообщения направляются к p . На самом деле в каждом вычислении всякого волнового алгоритма содержится такое остовное дерево; однако может случиться так, что процесс q отправляет несколько сообщений и не располагает сведениями о том, какие исходящие из него ребра принадлежат одному из таких деревьев. Если процесс осведомлен о том, какое исходящее ребро ведет к родительской вершине в подобном дереве, то само дерево можно использовать для вычисления сумм. Каждый процесс отправляет своей родительской вершине в этом дереве сумму всех входных данных, вычисленных в своем поддереве.

Этот принцип можно применять для древесного алгоритма, алгоритма эха, фазового алгоритма на кликах. Древесный алгоритм можно легко адаптировать так, чтобы в сообщении, отправляемое из p в q , включать сумму всех входных данных в дереве T_{pq} . Процесс, принимающий решение, вычисляет окончательный результат, суммируя значения, содержащиеся в двух сообщениях, следующих навстречу друг другу по одному и тому же ребру. Фазовый алгоритм на кликах адаптируется за счет того, что в каждое сообщение из q в p включаются входные данные q . Процесс p суммирует все полученные значения и свои собственные входные данные, и в результате будет получен правильный ответ, как только p примет решение. В алгоритме эха входные данные суммируются при помощи остовного дерева T , которое явно строится по ходу вычисления (см. упражнение 6.15).

Вычисление сумм с использованием отличительных признаков. Предположим, что каждый процесс наделен индивидуальным отличительным признаком. Сумма всех входных данных может быть вычислена так. Каждый процесс помещает свои входные данные своим индивидуальным ярлыком, образуя пару (p, j_p) . Отметим, что никакие два процесса не могут сформировать одну и ту же пару. Алгоритм гарантирует, что всякий раз, когда процесс принимает решение, он осведомлен о всех входных данных отдельных процессов; множество $S = \{(p, j_p) : p \in \mathbb{P}\}$ является объединением множеств $S_p = \{(p, j_p)\}$ по всем p , и его можно вычислить по ходу одной волны. На основе этого множества желаемый результат вычисляется с помощью локальных операций.

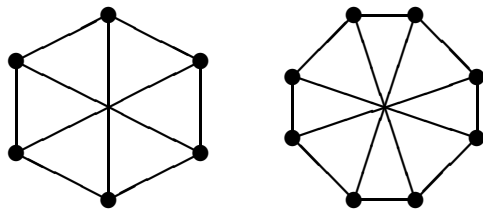


Рис. 6.20. Две сети диаметра два и степени три

Для такого решения задачи требуется, чтобы индивидуальные отличительные признаки всех процессов стали доступны, а это весьма значительно увеличивает

битовую сложность. Каждое сообщение волнового алгоритма включает подмножество S , для представления которого требуется до Nw битов, если считать, что один ярлык и входные данные одного процесса требуют в совокупности w битов (см. упражнение 6.16).

6.5.3. Альтернативные определения сложности по времени

Сложность по времени для распределенных алгоритмов можно определять несколькими разными способами. В этой книге всякий раз, когда речь заходит о сложности по времени, мы обращаемся к определению 6.31, но здесь мы обсудим и другие возможные определения¹⁾.

Определения, основанные на более ограниченных допущениях о времени. Время, расходуемое распределенными вычислениями, можно оценивать, воспользовавшись более ограниченными допущениями о хронометрировании событий в системе.

Определение 6.37. Для заданного алгоритма его *равномерная сложность по времени* — это наибольшее время вычисления алгоритма при следующих допущениях.

O1. Процесс может выполнять любой конечный набор действий за нулевое время.

O2. Время между отправлением и приемом сообщения в точности равно одной единице времени.

Сравнивая это определение с определением 6.31, заметим, что допущение O1 точно такое же, как и T1. Поскольку длительность передачи сообщений согласно допущению T2 *не превосходит* длительности передачи согласно допущению O2, может возникнуть мысль о том, что равномерная сложность по времени не меньше обычной сложности по времени. Аргументацию можно было бы основывать на том, что каждое вычисление проводится в соответствии с допущением T2 не медленнее, нежели в соответствии с допущением O2, и, следовательно, вычисление с максимальной задержкой также будет выполняться при допущении T2 не хуже, чем при допущении O2. Изъян в этом рассуждении состоит в том, что изменчивость длительности передачи сообщения, допускаемая T2, приводит к более разнообразному классу вычислений, в который, быть может, придется включить вычисления с неудовлетворительным поведением во времени. Напомним, что T2 — это не ограничение, налагаемое на выполнения алгоритмов, которые нужно хронометрировать, а определение единицы времени. В таком случае допущение O2 в действительности сужает множество выполнений, ограничивая его теми, в которых все сообщения передаются с одной и той же задержкой. Суть дела видна ниже на примере алгоритма эха.

На самом деле верно обратное: для всякого алгоритма его обычная сложность по времени не меньше, чем равномерная сложность по времени того же алгорит-

¹⁾В дальнейшем в этой главе, говоря о сложности алгоритмов по времени согласно определению 6.31, мы будем называть эту сложность *обычной сложностью по времени*. — Прим. перев.

ма. Каждое вычисление, согласованное с допущениями О1 и О2, соответствует также и допущениям Т1 и Т2, и при этом оно укладывается точно в такой же отрезок времени. Следовательно, наихудший возможный случай поведения алгоритма согласно допущениям О1 и О2 охвачен определением 6.31 и дает нам нижнюю оценку сложности по времени.

Теорема 6.38. *Равномерная сложность по времени алгоритма эха равна $O(D)$. Обычная сложность по времени алгоритма эха равна $\Theta(N)$ даже для сетей диаметра 1.*

Доказательство. Чтобы проанализировать равномерную сложность по времени, воспользуемся допущениями О1 и О2. Всякий процесс, отстоящий на d переходов от инициатора, получает первое сообщение **tok** спустя в точности d единиц времени после начала вычисления, и он будет расположен на уровне d в построенном дереве T . (Это можно показать, применив индукцию по d .) Пусть D_T обозначает высоту дерева T . Тогда $D_T \leq D$ и процесс, расположенный на уровне d в дереве T , отправит сообщение **tok** в свою родительскую вершину не позднее, чем через $(2D_T + 1) - d$ единиц времени после начала вычисления. (Это можно показать при помощи обратной индукции по d .) Значит, инициатор примет решение не позднее, чем через $2D_T + 1$ единиц времени после начала вычисления.

Чтобы оценить обычную сложность по времени, обратимся к допущениям Т1 и Т2. Всякий процесс, отстоящий на d переходов от инициатора, получает первое сообщение **tok** спустя не более d единиц времени после начала вычисления (это можно обосновать, воспользовавшись индукцией по d). Предположим, что построение остова дерева завершилось спустя F единиц времени после начала вычисления, и тогда $F \leq D$. Высота остова дерева D_T совсем не обязательно будет ограничена его диаметром (это показано ниже на примере одного вычисления), но, поскольку есть всего N процессов, высота дерева не будет превосходить $N - 1$. Процесс, расположенный на уровне d в дереве T , отправляет сообщение **tok** в свою родительскую вершину не позднее, чем через $(F + 1) + (D_T - d)$ единиц времени после начала вычисления (это можно показать при помощи обратной индукции по d). Отсюда следует, что инициатор примет решение спустя не более $(F + 1) + D_T$ единиц времени после начала вычисления, а это есть величина $O(N)$.

Чтобы убедиться в том, что $\Omega(N)$ является нижней оценкой сложности по времени, построим вычисление на клике, для которого требуется N единиц времени. Выделим в клике остова дерево высоты $N - 1$ (представляющее собой не что иное, как линейную цепочку вершин). Условимся считать, что все сообщения **tok**, передаваемые по ребрам дерева от родительских вершин к их последователям, поступают к адресату $1/N$ единиц времени спустя после отправления, а сообщения **tok** по стягивающим ребрам поступают спустя одну единицу времени. Такие задержки не противоречат допущению Т2, и в этом вычислении все дерево строится за одну единицу времени, но имеет высоту $N - 1$. А теперь представим себе, что все сообщения, передаваемые процессами по ребрам дерева в их родительские вершины, также претерпевают задержку на одну единицу времени. В таком случае решение будет принято спустя в точности N единиц времени после начала вычисления. $\square$

Читатель может задаться вопросом: которое из этих двух определений предпочтительнее, когда речь заходит о сложности распределенных алгоритмов по времени. Равномерная сложность по времени обладает тем недостатком, что некоторые вычисления выпадают из рассмотрения, хотя они и возможны для заданного алгоритма. Среди неучтенных вычислений могут оказаться такие, для которых требуется особенно много времени. А вот допущения, представленные в определении 6.31, не упускают ни одного вычисления: ведь в этом определении всего лишь вводится единица измерения времени для каждого вычисления. Недостатком обычной сложности по времени является то, что окончательный итог подводится теми вычислениями (наподобие приведенного в доказательстве теоремы 6.38), которые, хотя и будучи осуществимыми в принципе, выглядят крайне неправдоподобно. И в самом деле, в этом вычислении одно-единственное сообщение передается по одному каналу медленнее, чем $N - 1$ сообщений по последовательной цепочке каналов.

В качестве компромисса между этими двумя определениями можно рассмотреть α -сложность по времени, которая определяется допущением о том, что задержка передачи каждого сообщения располагается между α и 1 (α — это константа, не превосходящая 1). К сожалению, такому компромиссному варианту присущи недостатки обоих альтернативных определений. Читателю предоставляется возможность самостоятельно убедиться в том, что для алгоритма эха его α -сложность по времени равна $O(\min(N, D/\alpha))$.

Наиболее точная мера сложности по времени получается при введении распределения вероятностей времени задержки сообщения, позволяющего определять математическое ожидание времени вычисления заданного алгоритма. У этого подхода есть два главных недостатка. Во-первых, анализ алгоритмов становится зависящим от свойств системы, поскольку для каждой распределенной системы имеется свое распределение вероятностей времени задержки сообщения. А во-вторых, в подавляющем большинстве случаев сам анализ оказался бы слишком сложной задачей.

Определения на основе цепочек сообщений. Вместо того чтобы обращаться к идеализированным допущениям, мы можем определить время, расходуемое распределенным вычислением, при помощи структурных свойств самого вычисления. Рассмотрим вычисление C .

Определение 6.39. *Цепочкой сообщений в C называется такая последовательность сообщений $m_1, m_2, \dots, m_k$, что для каждого i , $0 \leq i < k$, прием сообщения m_i предшествует в причинно-следственной связи отправлению сообщения m_{i+1} .*

Цепной сложностью по времени распределенного алгоритма называется длина самой протяженной цепочки сообщений в произвольном вычислении указанного алгоритма.

В этом определении, как и в определении 6.31, для оценки сложности алгоритма по времени рассматриваются все возможные выполнения алгоритма, но используются при этом разные меры сложности вычислений. Рассмотрим слу-

чай такого вычисления (описанного в доказательстве теоремы 6.38), когда единственное сообщение передается медленнее, чем цепочка из k сообщений. Обычная сложность по времени такого фрагмента вычисления равна 1, тогда как цепная сложность того же самого фрагмента вычисления равна k . В системах, где гарантирована верхняя оценка длительности задержки сообщения (как это подразумевается в определении обычной сложности по времени), обычная сложность по времени служит хорошей мерой. А в системах, где большинство сообщений поступают по истечении некоторой «средней» задержки и лишь малая их доля задерживается на гораздо больший срок, предпочтение отдается цепной сложности по времени.

6.6. Упражнения к главе 6

6.6.1

Упражнение 6.1. Приведите пример PIF-алгоритма для систем с *синхронным* обменом сообщениями, который не позволяет проводить вычисление точных нижних граней (так, как это указано в теоремах 6.7 и 6.12). Этот пример может быть пригоден только для топологии специального вида.

Упражнение 6.2. Элемент b частично упорядоченного множества $(X, \leq)$ называется нижней точкой, если для всякого $c \in X$ выполняется неравенство $b \leq c$. Доказательство теоремы 6.11 опирается на тот факт, что в частично упорядоченном множестве $(X, \leq)$ отсутствует нижняя точка. В каком месте доказательства используется этот факт?

Можно ли привести пример алгоритма, который не является волновым и при этом вычисляет точную нижнюю грань в частично упорядоченном множестве, *обладающим* нижней точкой?

Упражнение 6.3. Приведите пример двух отношений частичного порядка на множестве натуральных чисел, для которых в качестве функции точной нижней грани используется (1) наибольший общий делитель, и (2) наименьшее общее кратное.

Приведите пример отношений частичного порядка на семействе подмножеств универсума U , для которого в качестве функции точной нижней грани используется (1) пересечение множеств, и (2) объединение множеств.

Упражнение 6.4. Докажите теорему о точной нижней грани (теорему 6.13).

6.6.2

Упражнение 6.5. Покажите, что в каждом вычислении древесного алгоритма (см. алгоритм 6.2) в точности два процесса принимают решение.

Упражнение 6.6. Воспользуйтесь алгоритмом эха (алгоритм 6.4) для построения алгоритма, вычисляющего схему разметки префикса (см. § 4.4.3) для произвольной сети за $O(N)$ единиц времени с использованием $2|E|$ сообщений. Можно ли построить алгоритм, который вычислял бы схему разметки за время $O(D)$? (Здесь D обозначает диаметр сети.)

Упражнение 6.7. Покажите, что взаимосвязь, которая проявляется в лемме 6.19, сохраняется и в том случае, когда сообщения могут утеряться в канале pq , но не сохраняются, когда сообщения могут дублироваться. Какой из этапов доказательства утратит силу, если сообщения могут дублироваться?

Упражнение 6.8. Примените конструкцию из теоремы 6.12 к фазовому алгоритму для построения алгоритма вычисления максимума на множестве целочисленных входных данных всех процессов.

Какова битовая сложность, а также сложность по времени и числу обменов сообщениями построенного алгоритма?

Упражнение 6.9. Предположим, что есть желание использовать волновой алгоритм в сети, в которой возможно *дублирование* сообщений.

- 1) Какие изменения нужно внести в алгоритм эха?
- 2) Какие изменения нужно внести в алгоритм Финна?

6.6.3

Упражнение 6.10. *Полным двудольным графом* называется граф $G = (V, E)$, в котором $V = V_1 \cup V_2$, причем $V_1 \cap V_2 = \emptyset$ и $E = V_1 \times V_2$.

Приведите пример алгоритма $2x$ -обхода для полного двудольного графа.

Упражнение 6.11. В работе [67] было показано, что обход и широковещательное распространение сообщений возможны в гиперкубе и при *отсутствии* восприятия направления. При каком значении f такой алгоритм будет алгоритмом f -обхода?

6.6.4

Упражнение 6.12. Приведите пример вычисления алгоритма Тарри, в котором построенное дерево T не является DFS-деревом.

Упражнение 6.13. Напишите алгоритм, реализующий поиск в глубину с разметкой интервалов (см. § 4.4.2) для произвольной связной сети.

Может ли этот алгоритм провести поиск за $O(N)$ единиц времени? Может ли этот алгоритм провести поиск с использованием $O(N)$ обменов сообщениями?

Упражнение 6.14. Предположим, что алгоритм поиска в глубину с учетом сведений о соседях используется в системе, в которой каждый процесс не только располагает сведениями об отличительных признаках соседей, но имеет также доступ к множеству отличительных признаков всех процессов системы $(\mathbb{P})$. Покажите, что в этом случае можно обойтись сообщениями, состоящими из N битов каждое.

6.6.5

Упражнение 6.15. Адаптируйте алгоритм эха (алгоритм 6.4) для вычисления суммы входных данных всех процессов.

Упражнение 6.16. Допустим, что все процессы сетей, изображенных на рис. 6.20, обладают индивидуальными отличительными признаками и на вход каждого процесса подано целое число. Промоделируйте на обеих сетях вычисление фазового алгоритма, вычисляющего множество $S = \{(p, i_p) : p \in \mathbb{P}\}$, а также сумму всех входных данных.

Упражнение 6.17. Какова цепная сложность по времени фазового алгоритма на кликах (алгоритм 6.7)?

Г Л А В А 7

АЛГОРИТМЫ ИЗБРАНИЯ ЛИДЕРА

В этой главе будет обсуждаться задача *о выборах*, которую называют также задачей *избрания лидера*. Впервые задачу о выборах сформулировал Ле-Ланн в работе [131], и он же предложил первое ее решение (см. § 7.2.1). Проблема состоит в том, чтобы исходя из конфигурации, в которой все процессы пребывают в одном и том же состоянии, достичь такой конфигурации, в которой ровно один процесс будет находиться в состоянии *leader*, тогда как все остальные процессы будут пребывать в состоянии *lost*.

Выборы лидера среди процессов должны быть проведены, когда требуется выполнить какой-нибудь централизованный алгоритм и при этом нет заранее заданного кандидата, который мог бы выступить инициатором этого алгоритма. Такое может произойти, например, в том случае, когда процедура инициализации выполняется в начале работы или после выхода системы из строя. Поскольку множество активных процессов может быть заранее неизвестно, нельзя назначить раз и навсегда один из процессов на роль лидера.

Существует великое множество результатов (алгоритмов и более общих теорем), касающихся задачи о выборах. При отборе результатов, включенных в настоящую главу, мы придерживались следующих критериев.

1. Синхронные системы, анонимные процессы и отказоустойчивые алгоритмы рассматриваются в других главах. В этой главе всегда подразумевается, что процессы и каналы надежны, система вполне асинхронна и все процессы можно опознать по индивидуальным отличительным признакам.

2. Задаче о выборах была отведена роль «тестовой задачи» для сопоставления эффективности различных моделей вычислений. Поэтому мы затронем некоторые результаты, которые необходимы для проведения такого сравнения, и будем часто возвращаться к этой задаче в последующих главах (см. гл. 9, 12, 11).

3. Мы сосредоточим внимание на результатах, касающихся сложности по числу обменов сообщениями; алгоритмы с улучшенной сложностью по времени, а также результаты, устанавливающие взаимосвязь между сложностью по времени и сложностью по числу обменов сообщениями, здесь не обсуждаются.

7.1. Введение

В задаче о выборах требуется, начав вычисление из конфигурации, в которой все процессы пребывают в одном и том же состоянии, достичь такой конфигурации, в которой ровно один процесс находится в особом состоянии *leader*, тогда как все остальные процессы пребывают в состоянии *lost*. Тот процесс, который

находится в состоянии *leader* в конце вычисления, называется *лидером*, и о нем говорится, что он был избран посредством заданного алгоритма.

Определение 7.1. *Алгоритмом избрания лидера* называется алгоритм, который обладает следующими свойствами.

1. Каждый процесс наделен одним и тем же локальным алгоритмом.
2. Алгоритм является децентрализованным, т. е. вычисление может быть инициировано произвольным непустым подмножеством процессов.
3. Алгоритм достигает заключительной конфигурации в каждом вычислении, и в каждой заключительной конфигурации существует ровно один процесс, который находится в состоянии *leader*, а все остальные процессы при этом пребывают в состоянии *lost*.

Последнее свойство иногда бывает ослаблено и ограничено лишь требованием того, чтобы ровно один процесс находился в состоянии *leader*. Тогда может случиться так, что избранный процесс будет осведомлен о своей победе на выборах, а проигравшие все еще не будут подозревать о своем поражении. Если задан алгоритм, удовлетворяющий этому ослабленному требованию, то его можно легко дополнить потоком сообщений, инициированным лидером, благодаря которому все процессы будут оповещены о результатах выборов. Это дополнительное замечание опущено в некоторых алгоритмах настоящей главы.

Во всех алгоритмах этой главы процесс p наделен переменной $state_p$, множество значений которой включает в себя состояния *leader* и *lost*. Иногда мы будем полагать, что значением $state_p$ является состояние *sleep*, до того как процесс p выполнит хоть один шаг нашего алгоритма, и состояние *candidate*, как только p присоединился к вычислению, но еще не был осведомлен о том, выиграл или проиграл он эти выборы. В некоторых алгоритмах используются дополнительные состояния, наподобие *active*, *passive* и пр., которые особо отмечаются при описании алгоритма.

7.1.1. Допущения, которых придерживаются в этой главе

Задача о выборах изучается в этой главе в рамках тех допущений, которые мы здесь рассмотрим.

1. *Система вполне асинхронна.* Уже говорилось о том, что процессы не имеют доступа к общим часам и передача сообщений может длиться сколь угодно долго или коротко.

Оказывается, допущение о синхронной передаче сообщений (подразумевающее, что отправление и прием сообщения рассматриваются как единый переход) почти не влияет на результаты, относящиеся к задаче о выборах. Читателю предоставляется возможность убедиться самостоятельно, что все алгоритмы, приведенные в этой главе, можно применять в системах с синхронной передачей сообщений и все результаты, касающиеся нижних оценок сложности, можно также переложить на этот случай.

Допущение о глобальном отсчете времени, наподобие предположения о том, что процессы могут отслеживать реальное время и что задержка передачи сооб-

щения ограничена, действительно оказывает важное влияние на решение задачи о выборах.

2. *Каждый процесс опознается по уникальному имени — своему отличительному признаку, — и это имя изначально известно самому процессу.* Для простоты предполагалось, что отличительный признак процесса p — это само имя p . Отличительные признаки заимствованы из линейно упорядоченного множества $\mathcal{P}$, т. е. на множестве отличительных признаков действует отношение $\leq$. Число битов, необходимых для представления отличительного признака, обозначим w .

Важное значение уникальных отличительных признаков состоит в том, что их можно использовать не только для адресации сообщений, но и для того, чтобы нарушить симметрию на множестве процессов. При проектировании алгоритма избрания лидера можно, например, постулировать, что на выборах победит процесс с наименьшим (или, напротив, наибольшим) по порядку отличительным признаком. Наша проблема, таким образом, превращается в задачу отыскания наименьшего отличительного признака посредством децентрализованного алгоритма. В этом случае задачу о выборах называют задачей *отыскания экстремумов*.

Хотя некоторые из алгоритмов, которые обсуждаются в этой главе, были первоначально сформулированы так, чтобы избирался наибольший процесс, на самом деле для большинства из них мы будем пользоваться формулировкой, при которой избирается наименьший процесс; в каждом таком случае алгоритм, предназначенный для избрания наибольшего процесса, можно легко получить, обратив порядок, на основе которого проводится сравнение отличительных признаков.

3. *Некоторые результаты в этой главе затрагивают алгоритмы сравнения.* Алгоритмами сравнения называются алгоритмы, в которых используется единственная операция — сравнение отличительных признаков. Как можно увидеть из описания самих алгоритмов, все алгоритмы, рассматриваемые в этой главе, — это алгоритмы сравнения. Всякий раз, представляя результат о нижней оценке, мы явно указываем, касается ли она алгоритмов сравнения.

Как было установлено (например, Бодлаэрдом в работе [33] для случая кольцевой сети), в асинхронных сетях произвольные алгоритмы не могут иметь сложность, меньшую, чем алгоритмы сравнения. Но, как будет показано в гл. 12, для синхронных систем это уже не так; в таких системах произвольные алгоритмы способны иметь сложность, меньшую, нежели алгоритмы сравнения.

4. *Каждое сообщение может содержать $O(w)$ битов.* В каждом сообщении может содержаться лишь постоянное число отличительных признаков процессов. Это допущение сделано для того, чтобы справедливо сопоставлять сложность по числу обменов сообщениями для различных алгоритмов.

7.1.2. Выборы и волны

Как уже отмечалось, отличительные признаки процессов можно использовать для нарушения симметрии между процессами; алгоритм избрания можно спроектировать так, чтобы выделять процесс с наименьшим отличительным признаком.

Согласно результатам, приведенным в §6.1.5, наименьший отличительный признак может быть вычислен процессами по ходу одной волны. Это значит, что выборы можно провести, запустив волну вычисления наименьшего отличительного признака, после чего процесс с наименьшим признаком становится лидером. Так как алгоритм избрания должен быть децентрализованным, этот принцип можно применять только к децентрализованным волновым алгоритмам (см. таблицу 6.18).

```

var  $ws_p$  : bool           init false ;
     $wr_p$  : integer         init 0 ;
     $rec_p[q]$  : bool для каждого  $q \in Neigh_p$  init false ;
     $v_p$  :  $\mathcal{P}$               init  $p$  ;
     $state_p$  : (sleep, leader, lost)         init sleep ;

begin if  $p$  is initiator then
    begin  $ws_p := true$  ;
        forall  $q \in Neigh_p$  do send  $\langle wakeup \rangle$  to  $q$ 
    end;
    while  $wr_p < \#Neigh_p$  do
        begin receive  $\langle wakeup \rangle$  ;  $wr_p := wr_p + 1$  ;
            if not  $ws_p$  then
                begin  $ws_p := true$  ;
                    forall  $q \in Neigh_p$  do send  $\langle wakeup \rangle$  to  $q$ 
                end
            end;
        (* Здесь начинает работать древесный алгоритм *)
        while  $\#\{q : \neg rec_p[q]\} > 1$  do
            begin receive  $\langle tok, r \rangle$  from  $q$  ;  $rec_p[q] := true$  ;
                 $v_p := \min(v_p, r)$ 
            end;
            send  $\langle tok, v_p \rangle$  to  $q_0$  with  $\neg rec_p[q_0]$  ;
            receive  $\langle tok, r \rangle$  from  $q_0$  ;
             $v_p := \min(v_p, r)$ ; (* принять решение с ответом  $v_p$  *)
            if  $v_p = p$  then  $state_p := leader$  else  $state_p := lost$  ;
            forall  $q \in Neigh_p, q \neq q_0$  do send  $\langle tok, v_p \rangle$  to  $q$ 
        end
    end

```

Алгоритм 7.1. Алгоритм избрания лидера на деревьях

Избрание посредством древесного алгоритма. Если топология сети имеет вид дерева, или у нас в распоряжении есть остовное дерево, выборы можно провести, воспользовавшись древесным алгоритмом (см. §6.2.2). В древесном алгоритме требуется, чтобы среди его инициаторов были все листовые вершины. Чтобы запустить алгоритм и в том случае, когда лишь часть процессов являются инициаторами, к нему добавлен этап *побудки*. Процессы, которые хотят затеять выборы, наводняют сеть сообщениями $\langle wakeup \rangle$. Булева перемен-

ная ws используется для того, чтобы каждый процесс не отправлял сообщение $\langle wakeup \rangle$ более одного раза, а переменная wr применяется для подсчета числа сообщений $\langle wakeup \rangle$, которые получил процесс. Когда процесс получил сообщение $\langle wakeup \rangle$ по всем каналам, он запускает алгоритм 6.2, который снабжен приставкой (как указано в теореме 6.12) для вычисления наименьшего отличительного признака, позволяющей, кроме того, вынудить каждый процесс принять решение. Когда процесс принимает решение, он располагает сведениями об отличительном признаке лидера; если этот признак совпадает с отличительным признаком самого процесса, то он становится лидером, а в противном случае он получает статус *lost* (см. алгоритм 7.1).

Теорема 7.2. Алгоритм 7.1 решает задачу о выборах на древесных сетях с использованием $O(N)$ обменов сообщениями, затрачивая при этом $O(D)$ единиц времени.

Доказательство. Когда хоть один процесс запускает данный алгоритм, все процессы отправляют сообщение $\langle wakeup \rangle$ всем своим соседям, и каждый процесс запускает выполнение древесного алгоритма после получения сообщения $\langle wakeup \rangle$ от каждого соседа. Все процессы завершают выполнение древесного алгоритма с одним и тем же значением переменной v , а именно с наименьшим отличительным признаком среди всех процессов. И (единственный) процесс с таким признаком завершит работу в состоянии *leader*, а все прочие процессы — в состоянии *lost*.

По каждому каналу передаются два сообщения $\langle wakeup \rangle$ и два сообщения $\langle tok, r \rangle$, и поэтому сложность по числу обменов сообщениями становится равной $4N - 4$. По истечении D единиц времени, после того как первый процесс запустил наш алгоритм, каждый процесс уже отправит сообщения $\langle wakeup \rangle$, и, следовательно, спустя $D + 1$ единиц времени каждый процесс уже запустит волну. И в самом деле, нетрудно увидеть, что первое решение будет принято спустя D единиц времени после начала движения волны, а последнее решение будет принято спустя D единиц времени после первого, и поэтому общее затраченное время становится равным $3D + 1$. Более подробный анализ позволяет обнаружить, что алгоритм всегда завершается спустя $2D$ единиц времени, но это оставлено читателю в качестве упражнения (см. упражнение 7.2). $\square$

Если сообщения в канале могут быть переупорядочены (т. е. канал не является очередью), то процесс может получить сообщение $\langle tok, r \rangle$ от соседа, *прежде чем* он получит сообщение $\langle wakeup \rangle$ от того же соседа. В таком случае сообщение $\langle tok, r \rangle$ можно временно сохранить или обработать, подобно тому как это делается для более поздних сообщений $\langle tok, r \rangle$.

Число обменов сообщениями можно сократить, воспользовавшись двумя модификациями. Во-первых, можно сделать так, чтобы неинициатор не отправлял сообщение $\langle wakeup \rangle$ тому процессу, от которого он получил первое сообщение $\langle wakeup \rangle$. Во-вторых, сообщение $\langle wakeup \rangle$, отправляемое из листового узла, можно сочетать с сообщением $\langle tok, r \rangle$, отправляемым из того же листового узла. При помощи этих улучшений число обменов сообщениями, которые требуются

нашему алгоритму, можно сократить до $3N - 4 + k$, где k обозначает число нелитовых вершин-стартеров; см. [187, р. 139].

Избрание посредством фазового алгоритма. Фазовый алгоритм можно приспособить для выборов, позволив ему вычислять наименьший отличительный признак по ходу одной волны, как указано в теореме 6.12.

Теорема 7.3. *При помощи фазового алгоритма (алгоритм 6.6) выборы можно провести в произвольных сетях, используя $O(D|E|)$ обменов сообщениями и затрачивая $O(D)$ единиц времени.*

В основу алгоритма Пелега (см. [158]) положен фазовый алгоритм; ему требуется $O(D|E|)$ обменов сообщениями и $O(D)$ единиц времени, но знание D для него не нужно, так как в нем предусмотрено вычисление диаметра в режиме *online*.

Избрание посредством алгоритма Финна. Алгоритму Финна (алгоритм 6.8) не требуется предварительных сведений о диаметре сети. В алгоритме Финна проводится $O(N|E|)$ обменов сообщениями, но сами сообщения гораздо длиннее того, что позволяют нам допущения, сделанные в этой главе. Поэтому каждое сообщение в алгоритме Финна следует рассматривать как $O(N)$ сообщений; таким образом, сложность по числу обменов сообщениями становится равной $O(N^2|E|)$.

7.2. Кольцевые сети

В этом параграфе рассматриваются некоторые алгоритмы избрания для *неориентированных* колец. Задача о выборах применительно к кольцевым сетям была впервые поставлена Ле-Ланном в работе [131], ему же удалось найти ее решение со сложностью $O(N^2)$ по числу обменов сообщениями. Это решение было улучшено Ченем и Робертсом (см. [40]); они предложили алгоритм, который имеет сложность $O(N^2)$ в наихудшем случае, но зато его сложность в среднем составляет всего лишь $O(N \log N)$. Решения Ле-Ланна и решения Ченя—Робертса обсуждаются в § 7.2.1. Вопрос о существовании алгоритма со сложностью $O(N \log N)$ в наихудшем случае оставался открытым до 1980 г., когда такой алгоритм был предложен Хиршбергом и Синклейром в работе [105]. В отличие от более ранних решений, в алгоритме Хиршберга—Синклейра требуется, чтобы каналы были двунаправленными. Какое-то время предполагалось, что $\Omega(N^2)$ обменов сообщениями составляют нижнюю оценку для однонаправленных колец, но Петерсон, а также Долев, Клейв и Роде (см. [159] и [69]) независимо предложили решение сложности $O(N \log N)$ для однонаправленного кольца. Их решение исследуется в § 7.2.2.

Почти в то же самое время указанные алгоритмы были дополнены согласованными нижними оценками. Нижняя оценка $\approx 0.34N \log N$ сложности по числу обменов сообщениями в наихудшем случае для двунаправленных колец была обоснована Бодлаендэром; см. [31]. Пачль, Корах и Ротем в работе [156] доказали нижнюю оценку $\Omega(N \log N)$ сложности в среднем как для двунаправленных,

так и для однонаправленных колец. Эти результаты о нижних оценках будут изучены в § 7.2.3.

7.2.1. Алгоритмы Ле-Ланна и Ченя—Робертса

В алгоритме Ле-Ланна (см. [131]) каждый инициатор вычисляет список отличительных признаков всех инициаторов, после чего лидером избирается инициатор с наименьшим признаком. Каждый инициатор отправляет по кольцу маркер, в который вложен отличительный признак инициатора, и все процессы передают далее этот маркер. Предполагается, что в каналах соблюдается очередность сообщений и всякий инициатор должен создать свой маркер, прежде чем получит маркер от всякого другого инициатора. (Как только процесс получает маркер, он уже не может впоследствии инициировать выполнение алгоритма.) Когда инициатор p получает свой собственный маркер обратно, маркеры всех инициаторов уже прошли через p , и p будет избран лидером в том и только том случае, если p — наименьший процесс среди всех инициаторов (см. алгоритм 7.2).

```

var Listp      : подмножество  $\mathcal{P}$       init{ $p$ };
    statep ;

begin if  $p$  is initiator then
    begin statep := cand ; send  $\langle \text{tok}, p \rangle$  to Nextp ; receive  $\langle \text{tok}, q \rangle$  ;
        while  $q \neq p$  do
            begin Listp := Listp  $\cup$  { $q$ } ;
                send  $\langle \text{tok}, q \rangle$  to Nextp ; receive  $\langle \text{tok}, q \rangle$ 
            end;
            if  $p = \min(\text{List}_p)$  then statep := leader
                else statep := lost
            end
        else while true do
            begin receive  $\langle \text{tok}, q \rangle$  ; send  $\langle \text{tok}, q \rangle$  to Nextp ;
                if statep = sleep then statep := lost
            end
        end
end

```

Алгоритм 7.2. Алгоритм Ле-Ланна избрания лидера

Теорема 7.4. *Алгоритм Ле-Ланна (алгоритм 7.2) решает задачу о выборах на кольцах с использованием $O(N^2)$ обменов сообщениями за $O(N)$ единиц времени.*

Доказательство. Так как порядок следования маркеров по кольцу остается неизменным (вследствие допущения очередности в каналах) и инициатор q отправляет $\langle \text{tok}, q \rangle$ *ранее*, нежели q получает $\langle \text{tok}, p \rangle$, инициатор p получает $\langle \text{tok}, q \rangle$, до того как p получит $\langle \text{tok}, p \rangle$ обратно. Отсюда следует, что каждый инициатор p завершает работу со списком List_p, представляющим множество

всех инициаторов, и инициатор с наименьшим отличительным признаком становится единственным процессом, избранным на роль лидера. Всего оказываются задействованными не более N различных маркеров, и каждый из них совершает N шагов, что приводит к сложности $O(N^2)$ по числу обменов сообщениями. Спустя самое позднее $N - 1$ единиц времени, после того как первый инициатор отправил свой маркер, каждый инициатор проделает то же самое. При этом каждый инициатор получит свой маркер обратно в течение N единиц времени после создания этого маркера. Значит, наш алгоритм завершит работу не позднее, чем спустя $2N - 1$ единиц времени. $\square$

Все неинициаторы переходят в состояние *lost*, но будут пребывать сколь угодно долго в ожидании сообщений $\langle \text{tok}, r \rangle$. Ожидание можно прервать, если заставить лидера отправить специальный маркер по кольцу для оповещения о том, что выборы завершены.

Алгоритм Ченя—Робертса [40] улучшает алгоритм Ле-Ланна за счет того, что из кольца изымаются все маркеры тех процессов, относительно которых уже становится ясно, что они проиграют выборы. А именно, инициатор p изымает маркер $\langle \text{tok}, q \rangle$ из кольца, если $q > p$. Инициатор p обретает статус *lost*, едва лишь получает маркер с отличительным признаком $q < p$, и статус *leader*, как только получает маркер с отличительным признаком p (см. алгоритм 7.3).

```
var statep ;
```

```
begin if p is initiator then
```

```
    begin statep := cand ; send  $\langle \text{tok}, p \rangle$  to nextp ;
```

```
    while statep  $\neq$  leader do
```

```
        begin receive  $\langle \text{tok}, q \rangle$  ;
```

```
            if  $q = p$  then statep := leader
```

```
            else if  $q < p$  then
```

```
                begin if statep = cand then statep := lost ;
```

```
                    send  $\langle \text{tok}, q \rangle$  to nextp
```

```
                end
```

```
            end
```

```
        end
```

```
    else while true do
```

```
        begin receive  $\langle \text{tok}, q \rangle$  ; send  $\langle \text{tok}, q \rangle$  to nextp ;
```

```
        if statep = sleep then statep := lost
```

```
        end
```

```
end
```

(* Только лидер завершает выполнение программы. Он отправляет сообщение всем процессам, информируя их об отличительных признаках лидера и о завершении выборов *)

Алгоритм 7.3. Алгоритм Ченя—Робертса избрания лидера

Теорема 7.5. Алгоритм Ченя—Робертса (алгоритм 7.3) решает задачу о выборах для колец, используя в худшем случае $\Theta(N^2)$ обменов сообщениями за $O(N)$ единиц времени.

Доказательство. Обозначим символом p_0 инициатор с наименьшим отличительным признаком. Всякий другой процесс может быть либо неинициатором, либо инициатором с отличительным признаком, превышающим p_0 , и поэтому все процессы передадут далее маркер $\langle \text{tok}, p_0 \rangle$, выпущенный p_0 . Значит, p_0 получит свой маркер обратно и будет избран лидером.

Неинициаторы не будут избраны, но все они перейдут в состояние *lost* самое позднее к моменту передачи маркера, который выпустил процесс p_0 . Инициатор p , для которого $p > p_0$, не будет избран лидером; p_0 не передаст далее маркер $\langle \text{tok}, p \rangle$, и поэтому p никогда не получит свой собственный маркер. Такой инициатор p перейдет в состояние *lost* самое позднее в тот момент, когда будет передавать далее $\langle \text{tok}, p_0 \rangle$. Это и служит обоснованием того, что наш алгоритм решает задачу о выборах.

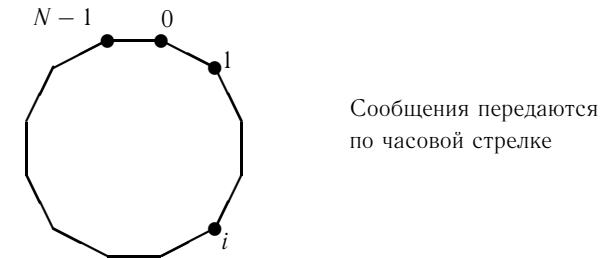


Рис. 7.4. Наихудший возможный сценарий для алгоритма Ченя—Робертса

В алгоритме задействовано не более N различных маркеров, и каждый маркер передается не более N раз; этим и обосновывается оценка $O(N^2)$ сложности по числу обменов сообщениями. Чтобы убедиться в том, что иногда может понадобиться передать $\Omega(N^2)$ сообщений, рассмотрим начальную конфигурацию, в которой отличительные признаки расположены в кольце по возрастанию (см. рис. 7.4), и каждый процесс является инициатором. Маркер каждого процесса изымается из кольца процессом 0, и поэтому маркер процесса i совершает $N - i$ переходов; поэтому число передач сообщений будет равно $\sum_{i=0}^{N-1} (N - i) = \frac{1}{2}N(N + 1)$. $\square$

Если речь идет о сложности по времени или о сложности по числу обменов сообщениями в худшем случае, то алгоритм Ченя—Робертса не улучшает указанные параметры по сравнению с алгоритмом Ле-Ланна. Зато достигается улучшение *в среднем*, если рассматривать все возможные распределения отличительных признаков по кольцу.

Теорема 7.6. Алгоритму Ченя—Робертса в среднем требуется всего лишь $O(N \log N)$ обменов сообщениями, когда все процессы являются инициаторами.

Доказательство. (В основу этого доказательства положен замысел Фридмана Маттерна.)

Исходя из предположения о том, что все процессы являются инициаторами, мы подсчитаем среднее число передач маркеров по всем возможным циклическим перестановкам N различных отличительных признаков. Рассмотрим некоторое фиксированное множество, состоящее из N отличительных признаков, и обозначим символом s наименьший признак. Тогда существуют $(N-1)!$ различных распределений отличительных признаков по кругу; для заданного распределения обозначим символом p_i отличительный признак процесса, отстоящего от s на i шагов *против хода часовой стрелки* (см. рис. 7.5).

Чтобы подсчитать общее число передач маркеров для всех возможных перестановок, мы сначала подсчитаем, сколько раз для всех возможных перестановок передается маркер $\langle \text{tok}, p_i \rangle$, а затем проведем суммирование по всем i . Для каждой перестановки маркер $\langle \text{tok}, s \rangle$ передается N раз; значит, всего он передается $N(N-1)!$ раз. Маркер $\langle \text{tok}, p_i \rangle$ передается не более i раз, поскольку он будет удален из кольца, как только достигнет s . Будем использовать запись $A_{i,k}$ для обозначения числа циклических перестановок, при которых $\langle \text{tok}, p_i \rangle$ передается в точности k раз. Тогда общее число передач $\langle \text{tok}, p_i \rangle$ равно $\sum_{k=1}^i (k \cdot A_{i,k})$.

Маркер $\langle \text{tok}, p_i \rangle$ передается в точности i раз, если p_i — наименьший отличительный признак среди всех признаков от p_1 до p_i , что имеет место для $(1/i) \cdot (N-1)!$ перестановок; таким образом,

$$A_{i,i} = \frac{1}{i} (N-1)!.$$

Маркер $\langle \text{tok}, p_i \rangle$ передается *не менее* k раз (где $k \leq i$), если вслед за процессом p_i расположены $k-1$ процессов, отличительные признаки которых превосходят p_i . Очевидно, что число перестановок, в которых p_i является наименьшим из k отличительных признаков $p_{i-k+1}, \dots, p_i$, составляет $1/k$ -ю долю всех перестановок, т. е. $(1/k)(N-1)!$. И, наконец, для $k < i$ маркер $\langle \text{tok}, p_i \rangle$ передается *в точности* k раз, если он передается не менее k раз, но и не более того, т. е. передается не менее k раз, но *не* передается не менее $k+1$ раз. Следовательно, число перестановок, для которых это происходит, равно

$$\frac{1}{k} (N-1)! - \frac{1}{k+1} (N-1)!,$$

и поэтому

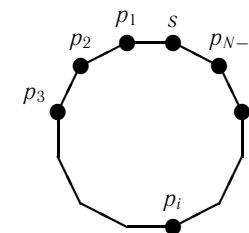
$$A_{i,k} = \frac{1}{k(k+1)} (N-1)! \quad (\text{для } k < i).$$

Получается, что для всех возможных перестановок маркер $\langle \text{tok}, p_i \rangle$ передается всего

$$\sum_{k=1}^{i-1} k \left(\frac{1}{k(k+1)} (N-1)! \right) + i \frac{1}{i} (N-1)!$$

раз, а это число равно $\left(\sum_{k=1}^i \frac{1}{k} \right) (N-1)!$. Сумма $\left(\sum_{k=1}^i \frac{1}{k} \right)$ известна под названием i -го *гармонического числа* и обозначается H_i . В качестве упражнения 7.3 предлагается доказать тождество

$$\sum_{i=1}^m H_i = (m+1)H_m - m.$$



Сообщения передаются по часовой стрелке

Рис. 7.5. Распределение отличительных признаков по кольцу

Далее мы суммируем количество прохождений маркеров через процесс i , чтобы получить общее количество передач маркеров (за исключением $\langle \text{tok}, s \rangle$) для всех перестановок. Оно равно

$$\sum_{i=1}^{N-1} [H_i (N-1)!] = (NH_{N-1} - (N-1))(N-1)!.$$

Добавив к нему $N(N-1)!$ передач маркера $\langle \text{tok}, s \rangle$, мы получаем суммарное количество передач маркеров

$$(NH_{N-1} + 1)(N-1)! = (NH_N)(N-1)!.$$

Так как мы имеем $(N-1)!$ различных перестановок, среднее число передач для всех возможных распределений, очевидно, равно NH_N и может быть оценено величиной $\approx 0.69N \log N$ (см. упражнение 7.4). $\square$

7.2.2. Алгоритм Петерсона/Долева—Клейва—Роде

Сложность по числу обменов сообщениями алгоритма Ченя—Робертса составляет $O(N \log N)$ в среднем, но не в наихудшем случае. Алгоритм сложности

$O(N \log N)$ в наихудшем случае был предложен Франклином в работе [90], однако в этом алгоритме требуется, чтобы каналы были двунаправленными, а также Долев, Клейв и Роде (см. Петерсон [159] и [69] независимо разработали весьма похожий алгоритм решения рассматриваемой задачи для однонаправленных колец, и в этом алгоритме в худшем случае проводится всего лишь $O(N \log N)$ обменов сообщениями. При этом требуется, чтобы в каналах связи поддерживалась очередность сообщений.

Вначале алгоритм вычисляет наименьший отличительный признак и доводит его до сведения всех процессов; затем процесс с указанным отличительным признаком становится лидером, а все остальные процессы признают свое поражение на выборах. Суть этого алгоритма будет проще понять, если взглянуть на него так, как будто алгоритм выполняют не сами процессы, а их *отличительные признаки*. Первоначально каждый отличительный признак является *активным*, но, как мы увидим далее, в каждом туре некоторые отличительные признаки становятся *пассивными*. В каждом туре всякий *активный* отличительный признак сравнивается с двумя соседними *активными* отличительными признаками, расположенными по ходу и против хода часовой стрелки. Если этот признак будет минимальным из трех, то он переходит в следующий тур, а иначе он становится *пассивным*. Так как все отличительные признаки попарно различны, отличительные признаки, расположенные по обе стороны от локального минимума, уже не будут образовывать такой локальный минимум, и поэтому по крайней мере половина отличительных признаков не перейдет в следующий тур. Следовательно, спустя самое большее $\log N$ туров останется только один *активный* отличительный признак, который и будет признан победителем.

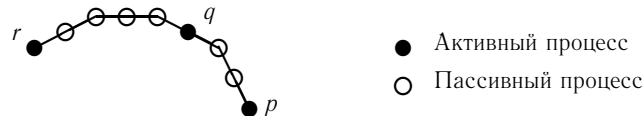


Рис. 7.6. Процесс p получает текущие отличительные признаки от процессов q и r

Этот принцип очень просто приспособить для двунаправленных сетей, как это сделано в алгоритме Франклина; см. [90]. В ориентированных сетях сообщения можно передавать только по часовой стрелке, и это затрудняет получение первого соседнего по ходу часовой стрелки отличительного признака, см. рис. 7.6. Отличительный признак q необходимо сравнить с r и p ; но если признак r может быть легко передан q , то признак p вынужден был бы двигаться в направлении, противоположном ориентации каналов, чтобы достичь q . Для того чтобы сделать возможным проведение сравнения с обоими признаками r и p , отличительный признак q передается (по направлению каналов в кольце) тому процессу, который в этот момент является хранителем признака p , а r переправляется не только процессу, хранящему q , но также и далее процессу, хранящему p . Если q остается единственным *активным* отличительным признаком в начале неко-

торого тура, то первый же признак, который будет передан q в этом туре, будет равен q (т. е. в этом случае $p = q$). Как только сложится такая ситуация, данный отличительный признак становится победителем на выборах.

```

var  $ci_p$  :  $\mathcal{P}$    init  $p$  ;      (* Текущий признак процесса  $p$  *)
     $acn_p$  :  $\mathcal{P}$    init undef ;  (* Признак активного соседа против хода часовой стрелки *)
     $win_p$  :  $\mathcal{P}$    init undef ;  (* Признак победителя *)
     $state_p$  : (active, passive, leader, lost) init active ;

begin if  $p$  is initiator then  $state_p := active$  else  $state_p := passive$  ;
    while  $win_p = undef$  do
        begin if  $state_p = active$  then
            begin send  $\langle one, ci_p \rangle$  ; receive  $\langle one, q \rangle$  ;  $acn_p := q$  ;
                if  $acn_p = ci_p$  then (*  $acn_p$  минимальный признак *)
                    begin send  $\langle small, acn_p \rangle$  ;  $win_p := acn_p$  ;
                        receive  $\langle small, q \rangle$ 
                    end
                else (*  $acn_p$  — текущий признак соседа *)
                    begin send  $\langle two, acn_p \rangle$  ; receive  $\langle two, q \rangle$  ;
                        if  $acn_p < ci_p$  and  $acn_p < q$ 
                            then  $ci_p := acn_p$ 
                            else  $state_p := passive$ 
                        end
                    end
                end
            end
        else (*  $state_p = passive$  *)
            begin receive  $\langle one, q \rangle$  ; send  $\langle one, q \rangle$  ;
                receive  $m$  ; send  $m$  ;
                (*  $m$  — это либо  $\langle two, q \rangle$ , либо  $\langle small, q \rangle$  *)
                if  $m$  — это сообщение  $\langle small, q \rangle$  then  $win_p := q$ 
            end
        end
    end;
    if  $p = win_p$  then  $state_p := leader$  else  $state_p := lost$ 
end

```

Алгоритм 7.7. Алгоритм Петерсона/Долева—Клейва—Роде

Рассмотрим алгоритм избрания лидера в однонаправленном кольце (алгоритм 7.7). Процесс p является *активным* в некотором туре, если в начале тура он хранит *активный* отличительный признак ci_p . В противном случае p является *пассивным* и просто ретранслирует все сообщения, которые он получает. Всякий *активный* процесс отправляет свой текущий отличительный признак следующему по порядку *активному* процессу и получает текущий отличительный признак от предшествующего *активного* процесса, используя сообщения типа $\langle one \rangle$. Полученный признак сохраняется в памяти (в качестве значения переменной acn_p), и если данный признак переживет этот тур, то он станет текущим отличительным признаком процесса p в следующем туре. Чтобы оценить, переживет ли отличительный признак acn_p данный тур, этот признак нужно срав-

нить как с ci_p , так и с *активным* признаком, полученным в сообщении типа $\langle \mathbf{two} \rangle$. Процесс p отправляет сообщение $\langle \mathbf{two}, asnp_p \rangle$, чтобы предоставить такую возможность следующему по порядку *активному* процессу. Исключительным является тот случай, когда выполняется равенство $asnp_p = ci_p$; тогда данный отличительный признак остается единственным *активным* признаком и сообщение $\langle \mathbf{small}, asnp_p \rangle$ оповещает об этом все процессы.

Теорема 7.7. *Алгоритм 7.7 решает задачу о выборах для однонаправленных колец с использованием $O(N \log N)$ обменов сообщениями.*

Доказательство. Будем говорить, что процесс участвует в i -м туре, если основной цикл этого процесса выполняется i -й раз. Туры не синхронизированы глобально; возможна ситуация, при которой один из процессов опережает на несколько туров другой процесс, расположенный в иной части кольца. Но коль скоро каждый процесс отправляет и получает в каждом туре ровно два сообщения и в каналах поддерживается очередность посланий, получатель всякого сообщения будет участвовать в том же туре, в котором находился его отправитель. В первом туре все инициаторы *активны* и каждый *активный* процесс хранит индивидуальный «текущий отличительный признак».

Утверждение 7.8. *Если в начале i -го тура имеется $k > 1$ активных процессов и все «текущие отличительные признаки» ci , хранимые этими процессами, попарно различны, то в следующий тур перейдет не менее одного и не более $k/2$ процессов. По окончании i -го тура текущие отличительные признаки активных процессов будут вновь попарно различны, и среди них будет находиться самый младший признак.*

Доказательство. После обмена сообщениями вида $\langle \mathbf{one}, q \rangle$, которые ретранслируются *пассивными* процессами, каждый *активный* процесс получит текущий отличительный признак своего ближайшего *активного* соседа, расположенного против хода часовой стрелки, и этот признак будет во всяком случае не совпадать с его собственным отличительным признаком. Следовательно, каждый *активный* процесс продолжит этот тур, и произойдет обмен сообщениями вида $\langle \mathbf{two}, q \rangle$, благодаря чему каждый активный процесс получит также текущий отличительный признак второго по близости *активного* соседа, расположенного против хода часовой стрелки. Все *активные* процессы теперь хранят разные значения $asnp$, а это означает, что процессы, выдержавшие этот тур, будут в конце его хранить разные отличительные признаки. Этот тур выдержит по крайней мере тот отличительный признак, который был наименьшим в начале тура, и, значит, хотя бы один процесс перейдет в следующий тур. А поскольку отличительный признак, который расположен вслед за локальным минимумом, не будет локальным минимумом, число процессов, перешедших в следующий тур, не будет превосходить $k/2$. $\square$

Из утверждения 7.8 следует, что наступит и такой тур (его номер не будет превосходить величины $\lfloor \log N \rfloor + 1$), в начале которого будет ровно один *активный* участник, а именно самый младший среди отличительных признаков всех инициаторов.

Утверждение 7.9. *Если в начале тура есть только один активный процесс p с текущим отличительным признаком ci_p , то по окончании этого тура алгоритм завершит свою работу, и при этом для каждого процесса q будет выполнено равенство $win_q = ci_p$.*

Доказательство. Сообщение $\langle \mathbf{one}, ci_p \rangle$, отправленное процессом p , будет ретранслировано всеми процессами и в конце концов будет получено самим процессом p . Процесс p убедится в том, что выполняется равенство $asnp_p = ci_p$, и отправит сообщение $\langle \mathbf{small}, asnp_p \rangle$ по кольцу, вынудив тем самым каждый процесс q выйти из основного цикла, присвоив при этом переменной win_q значение $asnp_p$. $\square$

Рассматриваемый нами алгоритм завершает свое выполнение в каждом из процессов, и при этом все процессы будут согласованно оповещены об отличительном признаке лидера (этот признак является значением переменной win_q). Процесс с указанным признаком будет пребывать в состоянии *leader*, а все остальные процессы — в состоянии *lost*.

Потребуется самое большее $\lfloor \log N \rfloor + 1$ туров, в каждом из которых происходит обмен ровно $2N$ сообщениями; это и служит обоснованием оценки сложности $2N \log N + O(N)$ по числу обменов сообщениями. Доказательство теоремы 7.7 завершено. $\square$

Долеву, Клейву и Роде удалось улучшить их алгоритм так, что верхняя оценка сложности была уменьшена до $1.5N \log N$, после чего Петерсон сумел построить алгоритм, в котором используется всего лишь $1.44N \log N$ обменов сообщениями. Но и эта оценка была впоследствии уменьшена Долевым, Клейвом и Роде до $1.356N \log N$. В течение 10 лет верхняя оценка $1.356N \log N$ для задачи о выборах на кольцах оставалась самой точной, но и она была уменьшена до $1.271N \log N$ Хайемом и Пржитикой в работе [104].

7.2.3. Результат о нижних оценках

В этом параграфе дается обоснование нижней оценке сложности задачи о выборах на однонаправленных кольцах. Так как выборы можно провести по ходу одного-единственного выполнения децентрализованного волнового алгоритма, в качестве следствия будет получена и нижняя оценка сложности децентрализованных волновых алгоритмов на кольцах.

Этот результат был получен Пачлом, Корачем и Ротемом в работе [156] в рамках следующих допущений.

1. Оценка установлена для алгоритмов, вычисляющих самый младший отличительный признак. Если лидер известен, то самый младший отличительный признак можно вычислить с использованием N обменов сообщениями, а если самый младший отличительный признак известен хотя бы одному процессу, то процесс с этим признаком может быть избран лидером также с использованием N обменов сообщениями. Следовательно, сложность задачи о выборах и сложность задачи вычисления самого младшего отличительного признака отстоят друг от друга самое большее на N обменов сообщениями.

2. Рассматриваемое кольцо однонаправленное.
3. Процессы не осведомлены о размерах кольца.
4. Предполагается, что в каналах поддерживается очередность сообщений.

Это допущение не ослабляет результат, поскольку сложность алгоритмов без соблюдения очередности сообщений не может быть лучше, чем сложность алгоритмов, требующих поддержания очередности сообщений.

5. Предполагается, что все процессы являются инициаторами. Это допущение не ослабляет результат, потому что оно описывает ситуацию, свойственную любому децентрализованному алгоритму.

6. Предполагается, что алгоритмы приводятся в действие сообщениями; это означает, что после того, как на этапе инициализации алгоритма были отправлены некоторые сообщения, всякий процесс отправляет последующие сообщения только после получения какого-нибудь сообщения. Если речь идет об *асинхронных* системах, то алгоритмы общего вида не дают выигрыша по сложности по сравнению с алгоритмами, приводимыми в действие сообщениями. И в самом деле, для всякого асинхронного алгоритма A можно построить следующий алгоритм B , который приводится в действие сообщениями. После инициализации и получения какого-нибудь сообщения алгоритм B отправляет максимальную последовательность сообщений, которые разрешается отправить алгоритму A , не проводя приема сообщений, и лишь затем получает очередное сообщение. Алгоритм B приводится в действие сообщениями, и, более того, каждое вычисление алгоритма B является одним из возможных вычислений алгоритма A (хотя, возможно, распределение задержек передач сообщений будет далеко не лучшим). Последние три допущения избавляют систему от недетерминированности. В рамках этих допущений каждое вычисление, начинающееся с заданной конфигурации, содержит одно и то же множество событий.

В этом параграфе записи $s = (s_1, \dots, s_N)$, t , и т. п. будут использоваться для обозначения наборов попарно различных отличительных признаков. Множество всех таких наборов обозначим D , т. е.

$$D = \{(s_1, \dots, s_k) : s_i \in \mathcal{P} \text{ и } i \neq j \implies s_i \neq s_j\}.$$

Длину набора s обозначим $len(s)$, а конкатенацию наборов s и t обозначим st . *Циклическим сдвигом* набора s назовем такой набор $s's''$, что $s = s''s'$; ясно, что этот набор имеет вид $s_i, \dots, s_N, s_1, \dots, s_{i-1}$. Запись $CS(s)$ будет обозначать множество циклических сдвигов набора s . Нетрудно видеть, что размер множества $CS(s)$ равен длине набора s .

Будем говорить, что кольцо помечено набором $(s_1, \dots, s_N)$, если именно в таком порядке в кольце размера N расположены отличительные признаки процессов, начиная с s_1 и оканчивая s_N . Кольцо, помеченное набором s , будем также называть s -кольцом. Если t является циклическим сдвигом набора s , то t -кольцо, естественно, будет тем же самым кольцом, что и s -кольцо.

С каждым сообщением, которое отправляется по ходу работы алгоритма, мы ассоциируем набор отличительных признаков процессов, который будем называть *трассой* данного сообщения. Если сообщение m было отправлено процессом p , до того как p получил некоторое сообщение, то трассой сообщения m служит

набор (p) . Если сообщение m было отправлено процессом p , после того как он получил некоторое сообщение, трассой которого служит набор $s = (s_1, \dots, s_k)$, то трасса сообщения m задается набором $(s_1, \dots, s_k, p)$. Сообщение с трассой s будет называться s -сообщением. Нижняя оценка будет получена на основе анализа свойств множества трасс всех сообщений, которые могут быть отправлены по ходу работы алгоритма.

Рассмотрим подмножество E множества D . Множество E назовем *исчерпывающим*, если выполнены следующие условия:

- 1) E префиксно замкнуто, т. е. $tu \in E \implies t \in E$;
- 2) E является циклическим покрытием множества D , т. е. $\forall s \in D: CS(s) \cap E \neq \emptyset$.

Как будет показано далее, множество всех трасс любого алгоритма¹⁾ является исчерпывающим. Чтобы вывести отсюда нижнюю оценку сложности такого алгоритма, определим две меры множества E . Будем говорить, что набор t является как последовательный набор отличительных признаков в s -кольце, если t — это префикс какого-нибудь набора $r \in CS(s)$. В качестве меры $M(s, E)$ выбирается количество наборов множества E , которые появляются последовательно в s -кольце, а в качестве $M_k(s, E)$ — количество всех таких наборов длины k , т. е.

$$M(s, E) = |\{t \in E : t \text{ является префиксом некоторого набора } r \in CS(s)\}|,$$

$$M_k(s, E) = |\{t \in E : t \text{ является префиксом некоторого набора } r \in CS(s) \text{ и } len(t) = k\}|.$$

Далее мы будем рассматривать произвольный алгоритм A вычисления самого младшего отличительного признака, а запись E_A будем использовать для обозначения множества таких наборов s , что по ходу выполнения алгоритма A на s -кольце отправляется некоторое s -сообщение.

Лемма 7.10. *Если оба набора t и u содержат набор s в качестве подстроки и по ходу работы алгоритма A на t -кольце отправляется некоторое s -сообщение, то какое-нибудь s -сообщение будет также отправлено по ходу работы алгоритма A на u -кольце.*

Доказательство. Отправление s -сообщения процессом s_k , где $s = (s_1, \dots, s_k)$, связано причинно-следственной зависимостью только с процессами s_i , $1 \leq i \leq k$. Их начальное состояние в u -кольце будет тем же самым, что и в t -кольце (здесь уместно напомнить, что размер кольца неизвестен), и, следовательно, все события, предшествующие отправлению рассматриваемого сообщения, также осуществляются и в u -кольце. $\square$

Лемма 7.11. *Множество E_A является исчерпывающим множеством.*

Доказательство. Чтобы убедиться в том, что E_A является префиксно замкнутым, заметим, что если A отправляет какое-нибудь s -сообщение по ходу работы на s -кольце, то это означает, что для всякого префикса t набора s алгоритм A сначала должен был отправить t -сообщение на том же самом s -кольце.

¹⁾Решающего задачу о выборах. — Прим. перев.

По лемме 7.10 алгоритм A , таким образом, отправляет t -сообщения на t -кольце, и, значит, $t \in E_A$.

Чтобы убедиться в том, что E_A циклически покрывает множество D , рассмотрим некоторое вычисление алгоритма A на s -кольце. По крайней мере один процесс должен принять решение относительно самого младшего отличительного признака, и отсюда следует (рассуждения здесь будут подобны тем, которые проводились при доказательстве теоремы 6.11), что указанный процесс получил сообщение, длина трассы которого равна $\text{len}(s)$. Эта трасса является циклическим сдвигом набора s и содержится в E_A . $\square$

Лемма 7.12. *По ходу работы алгоритма A на s -кольце отправляется не менее $M(s, E_A)$ сообщений.*

Доказательство. Рассмотрим произвольный префикс t , $t \in E_A$, циклического сдвига r набора s . По определению E_A некоторое t -сообщение отправляется по ходу работы алгоритма A на t -кольце. Значит, оно так же отправляется и по ходу работы алгоритма на r -кольце, которое совпадает с s -кольцом. Следовательно, для каждого элемента t из множества

$$\{t \in E_A : t \text{ является префиксом некоторого набора } r \in CS(s)\}$$

хотя бы одно t -сообщение отправляется по ходу вычисления на s -кольце, а это свидетельствует о том, что количество обменов сообщениями в таком вычислении будет не меньше, чем $M(s, E_A)$. $\square$

Для всякого множества I отличительных признаков процессов обозначим выражением $\text{Per}(I)$ множество всех перестановок множества I . Введем запись $\text{ave}_A(I)$ для обозначения среднего числа обменов сообщениями, используемых алгоритмом A на всех кольцах, помеченных отличительными признаками из множества I , а запись $\text{wor}_A(I)$ для обозначения числа обменов сообщениями, используемых в наихудшем случае. Из предшествующей леммы следует, что если множество I состоит из N элементов, то выполняются следующие неравенства:

$$1) \text{ave}_A(I) \geq \frac{1}{N!} \sum_{s \in \text{Per}(I)} M(s, E_A);$$

$$2) \text{wor}_A(I) \geq \max_{s \in \text{Per}(I)} M(s, E_A).$$

Нижнюю оценку теперь можно установить путем анализа произвольного исчерпывающего множества.

Теорема 7.13. *Сложность в среднем всякого однонаправленного алгоритма отыскания самого младшего отличительного признака будет не меньше чем $N \cdot \mathbf{H}_N$.*

Доказательство. Проводя усреднение по всем начальным конфигурациям, помеченным множеством I , мы получаем следующие соотношения:

$$\begin{aligned} \text{ave}_A(I) &\geq \frac{1}{N!} \sum_{s \in \text{Per}(I)} M(s, E_A) = \\ &= \frac{1}{N!} \sum_{s \in \text{Per}(I)} \sum_{k=1}^N M_k(s, E_A) = \\ &= \frac{1}{N!} \sum_{k=1}^N \sum_{s \in \text{Per}(I)} M_k(s, E_A). \end{aligned}$$

Зафиксируем k и заметим, что для каждого набора $s \in \text{Per}(I)$ имеется N префиксов длины k , которыми начинаются циклические сдвиги набора s . Так как множество $\text{Per}(I)$ содержит $N!$ перестановок, общее число таких префиксов будет достигать величины $N \cdot N!$. Эти префиксы можно разбить на $N \cdot N!/k$ групп, в каждой из которых содержится k циклических сдвигов одного и того же набора. Поскольку E_A циклически покрывает D , множество E_A пересекается с каждой такой группой, и поэтому справедливо неравенство

$$\sum_{s \in \text{Per}(I)} M_k(s, E_A) \geq \frac{N \cdot N!}{k}.$$

Отсюда и следует оценка

$$\text{ave}_A(I) \geq \frac{1}{N!} \sum_{k=1}^N \frac{N \cdot N!}{k} = N \cdot \mathbf{H}_N. \quad \square$$

Этот результат свидетельствует о том, что алгоритм Ченя—Робертса является оптимальным, если речь идет о сложности в среднем. Оценка сложности в наихудшем случае будет, по крайней мере, не меньше, чем оценка сложности в среднем, а это означает, что самая точная оценка сложности в наихудшем случае расположена между величиной $N \cdot \mathbf{H}_N \approx 0.69N \log N$ и величиной $1.356N \log N$.

Доказательство, приведенное в этом параграфе, существенно опирается на допущения о том, что кольцо является однонаправленным, и что размер кольца неизвестен. Нижняя оценка $\frac{1}{2}N \cdot \mathbf{H}_N$ была получена Бодлаендером в работе [31] для сложности в среднем алгоритмов решения задачи о выборах на *двунаправленных* кольцах при условии, что размер кольца неизвестен. Чтобы устранить недетерминизм в случае двунаправленных колец, пришлось рассматривать вычисления, в которых каждый процесс начинает работу в одно и то же время, а каждое сообщение имеет одну и ту же задержку при передаче. Для случая, когда размер кольца известен, Бодлаендер [32] получил нижнюю оценку $\frac{1}{2}N \log N$ на однонаправленных кольцах и $\left(\frac{1}{4} - \epsilon\right)N\mathbf{H}_N$ на двунаправленных кольцах (в обоих случаях речь идет о сложности в среднем).

Подводя итоги, заметим, что почти все сделанные нами допущения не влияют на сложность выборов на кольце. Независимо от того, известен или неизвестен размер кольца, является ли кольцо однонаправленным или двунаправленным, рассматривается ли средний или наихудший случай, всякий раз сложность составляет величину $\Theta(N \log N)$. Здесь существенно то, что кольцо асинхронно; как будет показано в гл. 12, для сетей, имеющих доступ к глобальному времени, сложность по числу обменов сообщениями будет меньше.

Так как лидера можно избрать по ходу одного-единственного вычисления децентрализованного волнового алгоритма, из нижней оценки сложности выборов следует и нижняя оценка сложности для волновых алгоритмов.

Следствие 7.14. *По ходу работы всякого децентрализованного алгоритма на кольцевых сетях происходит обмен, по меньшей мере, $\Omega(N \log N)$ сообщениями, как в среднем, так и в наихудшем случае.*

7.3. Произвольные сети

Теперь перейдем к изучению задачи о выборах для сетей произвольной, заранее неизвестной топологии при условии отсутствия сведений о соседях. Далее мы установим нижнюю оценку $\Omega(N \log N + |E|)$ числа обменов сообщениями. При обосновании ее мы будем сочетать идеи доказательства теоремы 6.6 и результаты предыдущего параграфа. В § 7.3.1 будет приведен простой алгоритм, имеющий малую сложность по времени, но большую сложность по числу обменов сообщениями в наихудшем случае. В § 7.3.2 будет представлен алгоритм, являющийся оптимальным по сложности в наихудшем случае.

Теорема 7.15. *Всякий алгоритм избрания лидера на основе сравнения для произвольных сетей имеет сложность (и в среднем, и в наихудшем случае), не меньшую чем $\Omega(|E| + N \log N)$.*

Доказательство. Слагаемое $\Omega(N \log N)$ служит нижней оценкой, потому что произвольные сети включают в себя и кольца, для которых соблюдается нижняя оценка $\Omega(N \log N)$. Чтобы убедиться в том, что $|E|$ обменов сообщениями служит нижней оценкой даже для самого лучшего из всех вычислений, допустим, что некоторый алгоритм избрания A допускает вычисление C на сети G , в ходе которого осуществляется обмен менее чем $|E|$ сообщениями (см. рис. 7.8). Построим сеть G' , соединив две копии сети G ребром, связывающим два узла, которые вставлены в каждой копии сети G посреди ребра, не задействованного в вычислении C . Отличительные признаки в обеих частях построенной сети сохраняют тот же относительный порядок, что и в сети G . Вычисление C можно промоделировать совместно в обеих частях сети G' , и, таким образом, мы получим вычисление, в результате которого лидерами будут избраны два процесса. $\square$

Следствие 7.16. *Всякий децентрализованный волновой алгоритм для произвольных сетей без предварительной осведомленности о соседях имеет сложность по числу обменов сообщениями, не меньшую чем $\Omega(|E| + N \log N)$.*

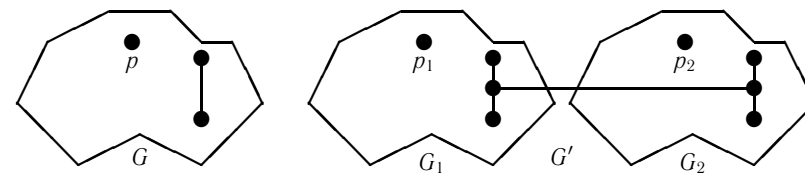


Рис. 7.8. Вычисление, приводящее к двум лидерам

7.3.1. Эффект угасания и быстрый алгоритм

Алгоритм избрания лидера можно получить из произвольного волнового алгоритма при помощи конструкции, которая носит название *угасание*. В полученном таким образом алгоритме каждый инициатор запускает отдельную волну. Чтобы отличаться от сообщений других волн, сообщения, движущиеся по волне, запущенной процессом p , должны быть помечены отличительным признаком процесса p . Данный алгоритм гарантирует, что независимо от количества запущенных волн только одна волна приведет к решению, а именно та волна, которую запустил инициатор с самым младшим отличительным признаком. Все остальные волны будут прерваны еще до того, как будет принято решение.

Алгоритм избрания лидера $Ex(A)$, соответствующий волновому алгоритму A , устроен так. Всякий процесс в каждый момент времени активен по отношению не более чем к одной волне; эта волна называется его *текущей активной волной*, обозначается saw и имеет первоначальное значение $undef$. Инициаторы выборов поступают так, как будто каждый из них запускает свою волну и придает переменной saw свой отличительный признак в качестве значения. Если сообщение некоторой волны, например волны, запущенной процессом q , достигает процесса p , то p проводит следующий анализ этого сообщения. Если $q > saw_p$, то сообщение просто игнорируется, что немедленно приводит к тому, что волна, запущенная процессом q , угасает. Если $q = saw_p$, то указанное сообщение обрабатывается так, как это предписано волновым алгоритмом. Если $q < saw_p$ или saw_p имеет значение $undef$, то p присоединяется к волне, запущенной процессом q , устанавливая в своих переменных первоначальные значения, а также выполняя присваивание $saw_p := q$. Когда волна, запущенная процессом q , приводит к осуществлению события решения (в большинстве алгоритмов решение всегда принимает сам процесс q), процесс q будет считаться избранным. Если же волновой алгоритм устроен так, что процесс, принимающий решение, не обязан быть инициатором, то процесс, осуществивший решение, оповещает об этом инициатора, используя остовное дерево, определенное в лемме 6.3. Для этого потребуется не более $N - 1$ обменов сообщениями; в последующих теоремах мы не уделяем этому внимания.

Теорема 7.17. *Если A является централизованным волновым алгоритмом, использующим M обменов сообщениями в каждой волне, то алгоритм*

```

var  $caw_p$  :  $\mathcal{P}$       init  $undef$  ; (* Текущая активная волна *)
       $rec_p$  : integer init 0 ;   (* Число полученных сообщений  $\langle tok, caw_p \rangle$  *)
       $father_p$  :  $\mathcal{P}$       init  $undef$  ; (* Родоначальник волны  $caw_p$  *)
       $lrec_p$  : integer init 0 ;   (* Число полученных сообщений  $\langle ldr \rangle$ . *)
       $win_p$  :  $\mathcal{P}$       init  $undef$  ; (* Отличительный признак лидера *)

begin if  $p$  is initiator then
  begin  $caw_p := p$ ; forall  $q \in Neigh_p$  do send  $\langle tok, p \rangle$  to  $q$  end;
  while  $lrec_p < \#Neigh_p$  do
    begin receive  $msg$  from  $q$  ;
    if  $msg = \langle ldr, r \rangle$  then
      begin if  $lrec_p = 0$  then
        forall  $q \in Neigh_p$  do send  $\langle ldr, r \rangle$  to  $q$  ;
         $lrec_p := lrec_p + 1$  ;  $win_p := r$ 
      end
    else (* сообщение  $\langle tok, r \rangle$  *)
      begin if  $r < caw_p$  then (* Алгоритм повторной инициализации *)
        begin  $caw_p := r$  ;  $rec_p := 0$  ;  $father_p := q$  ;
        forall  $s \in Neigh_p, s \neq q$  do send  $\langle tok, r \rangle$  to  $s$ 
        end;
        if  $r = caw_p$  then
          begin  $rec_p := rec_p + 1$  ;
          if  $rec_p = \#Neigh_p$  then
            if  $caw_p = p$ 
              then forall  $s \in Neigh_p$  do send  $\langle ldr, p \rangle$  to  $s$ 
              else send  $\langle tok, caw_p \rangle$  to  $father_p$ 
            end (* Если  $r > caw_p$ , то сообщение игнорируется *)
          end
        end
      end
    end; if  $win_p = p$  then  $state_p := leader$  else  $state_p := lost$ 
  end

```

Алгоритм 7.9. Угасание в сочетании с алгоритмом эха

Ex(A) проводит выборы лидера с использованием не более NM обменов сообщениями.

Доказательство. Допустим, что p_0 — инициатор, обладающий самым младшим отличительным признаком. К волне, запущенной процессом p_0 , немедленно начнут присоединяться все процессы по мере получения сообщений этой волны, и каждый процесс продолжит эту волну, поскольку нет волны, несущей более младший отличительный признак, ради которой этот процесс стал бы гасить волну, запущенную p_0 . Следовательно, волна, запущенная p_0 , будет двигаться до самого конца, решение будет принято, и процесс p_0 станет лидером.

Если процесс p не является инициатором, то и волна, несущая отличительный признак p , никогда не зародится, и, следовательно, процессу p не бывать лидером. Если же процесс p , отличный от p_0 , является инициатором, то волна, несущая отличительный признак p , будет запущена, но в этой волне принятию решения предшествовало бы событие отправления сообщения (все еще в преде-

лах этой волны) процессом p_0 , или же само решение принималось бы процессом p_0 (лемма 6.4). А так как p_0 никогда не отправляет сообщений и не выполняет никаких внутренних действий в пределах волны, несущей отличительный признак процесса p , такое решение не может состояться, и процесс p не будет избран.

Запускается не более N волн, и в каждой волне проводится не более M обменов сообщениями. Поэтому общая сложность будет составлять самое большее NM . $\square$

Гораздо более тонкий вопрос — оценить сложность алгоритма $Ex(A)$ по времени. Во многих случаях эта сложность будет иметь такой же порядок, что и сложность по времени алгоритма A , но иногда при неудачном стечении обстоятельств может случиться так, что инициатор с самым младшим отличительным признаком запускает свою волну слишком поздно. В общем случае можно показать, что временная сложность будет составлять $O(Nt)$, где t — сложность по времени базового волнового алгоритма, потому что спустя t единиц времени, после того как инициатор p запустил свою волну, либо запущенная им волна должна привести к решению, либо будет запущена другая волна.

Если принцип угасания применить к кольцевому алгоритму, то будет получен алгоритм Ченя—Робертса (см. упражнение 7.9). Алгоритм 7.9 представляет собой алгоритм избрания лидера, полученный на основе алгоритма эха. Для упрощения описания предполагается, что неравенство $undef > q$ выполняется для всех $q \in \mathcal{P}$. Просматривая текст описания алгоритма, читатель должен обратить внимание на то, что после получения сообщения $\langle tok, r \rangle$, для которого выполняется неравенство $r < caw_p$, оператор ветвления **if** с условием $r = caw_p$ также выполняется, ввиду более раннего присваивания значения переменной caw_p . Когда процесс p будет избран (т. е. получит $\langle tok, p \rangle$ от каждого своего соседа), он разошлет сообщение $\langle ldr, p \rangle$ всем процессам, уведомляя их о том, что он стал лидером, и вынуждая их завершить выполнение алгоритма.

7.3.2. Алгоритм Галладжера—Хамблета—Спиры

Как мы сейчас покажем, задача о выборах в произвольной сети тесно связана с задачей построения остовного дерева децентрализованным алгоритмом. Будем использовать запись C_E для обозначения сложности по числу обменов сообщениями задачи о выборах, а запись C_T — для обозначения сложности построения остовного дерева. Из теоремы 7.2 следует, что $C_E \leq C_T + O(N)$, а если в нашем распоряжении есть лидер, то остовное дерево можно построить с использованием $2|E|$ обменов сообщениями при помощи алгоритма эха; поэтому справедливо неравенство $C_T \leq C_E + 2|E|$. Нижняя оценка величины C_E (см. теорему 7.15) свидетельствует о том, что обе задачи имеют сложность одного и того же порядка, а именно для решения обеих задач требуется использовать по меньшей мере $\Omega(N \log N + E)$ обменов сообщениями.

В этом параграфе представлен алгоритм Галладжера—Хамблета—Спиры (GHS), предназначенный для построения (минимального) остовного дерева с использованием $2|E| + 5N \log N$ обменов сообщениями. Отсюда видно, что порядок роста

обеих величин C_E и C_T составляет $\Theta(N \log N + E)$. Указанный алгоритм был опубликован в [96]. Этот алгоритм можно легко модифицировать (о чем будет упомянуто в конце этого параграфа) и так настроить для проведения выборов лидера по ходу построения, чтобы отпала необходимость в проведении отдельных выборов согласно описанию, приведенному выше.

Алгоритм GHS опирается на следующие допущения.

1. Каждому ребру приписан уникальный вес $\omega(e)$. Здесь предполагается, что вес $\omega(e)$ — это действительное число, но в качестве весов могут выступать и целые числа.

Если уникальные веса ребер не заданы а priori, то каждому ребру можно приписать вес, основываясь на отличительных признаках тех узлов, которые соединены этим ребром. Таким образом, для вычисления весов ребер необходимо, чтобы узлы были осведомлены об отличительных признаках своих соседей; для этого потребуется $2|E|$ дополнительных сообщений на этапе инициализации алгоритма.

2. Все узлы пребывают первоначально в состоянии оцепенения и пробуждаются перед началом выполнения алгоритма. Некоторые узлы пробуждаются самопроизвольно (если выполнение алгоритма запускается в силу особенностей устройства этих узлов), другие могут получить сообщение по ходу работы алгоритма, еще пребывая в оцепенении. В последнем случае узел, получивший сообщение, вначале выполняет процедуру локальной инициализации, а затем приступает к обработке этого сообщения.

Минимальное остовное дерево. Рассмотрим взвешенный граф $G = (V, E)$ и для обозначения веса ребра e будем использовать запись $\omega(e)$. Вес остовного дерева T в графе G полагается равным сумме весов всех $N - 1$ ребер, входящих в состав T , и при этом T называется *минимальным остовным деревом* или сокращенно *MST* (иногда его называют также остовным деревом минимального веса), если ни одно остовное дерево не имеет вес, меньший чем T . В этом параграфе мы будем полагать, что каждое ребро имеет уникальный вес, т. е. различные ребра имеют различные веса; хорошо известно, что в таком случае в графе существует единственное минимальное остовное дерево.

Предложение 7.18. *Если все веса ребер попарно различны, то существует только одно MST.*

Доказательство. Допустим противное и рассмотрим случай, когда оба дерева T_1 и T_2 (причем $T_1 \neq T_2$) являются минимальными остовными деревьями. Рассмотрим ребро e наименьшего веса, которое содержится в одном из этих деревьев, но не содержится в другом; такое ребро существует, ибо $T_1 \neq T_2$. Допустим, не умаляя общности, что e принадлежит T_1 , но не входит в состав T_2 . Тогда граф $T_2 \cup \{e\}$ содержит цикл, и, поскольку в дереве T_1 нет циклов, хотя бы одно ребро этого цикла, скажем ребро e' , не содержится в T_1 . Согласно выбору ребра e , справедливо неравенство $\omega(e) < \omega(e')$. Но тогда дерево $T_2 \cup \{e\} \setminus \{e'\}$ имеет вес, меньший чем T_2 , вопреки тому, что T_2 является MST. $\square$

Утверждение 7.18 существенно облегчает распределенное построение минимального остовного дерева, так как не оставляет возможности (распределенного) выбора ответа из семейства правильных ответов. Напротив, каждый узел, который совершает локальный выбор ребра, принадлежащего хоть какому-нибудь минимальному дереву, вносит тем самым вклад в построение единственного глобального MST.

В основу всех алгоритмов построения минимального остовного дерева положено понятие *фрагмента*. Фрагментом может быть произвольное поддерево MST. Ребро e называется *исходящим ребром* фрагмента F , если один из его концов принадлежит F , а другой нет. Алгоритмы начинают работу, располагая фрагментами, состоящими из единственного узла, и последовательно наращивают фрагменты, до тех пор пока не будет завершено построение MST, основываясь при этом на следующем утверждении.

Предложение 7.19. *Если F является фрагментом и e — ребро наименьшего веса, исходящее из F , то $F \cup \{e\}$ также является фрагментом.*

Доказательство. Допустим, что $F \cup \{e\}$ не является фрагментом MST. Тогда e наряду с некоторыми ребрами MST образует цикл и при этом одно из ребер этого цикла, принадлежащее MST (например f), является ребром, исходящим из фрагмента F . Согласно выбору ребра e справедливо неравенство $\omega(e) < \omega(f)$; но в таком случае после изъятия ребра f из MST и добавления вместо него ребра e , мы получили бы дерево, вес которого был бы меньше веса MST, т. е. пришли бы к противоречию. $\square$

Хорошо известны последовательные алгоритмы построения MST, среди которых можно отметить алгоритмы Прима и Краскала. Алгоритм Прима (см. [55, раздел 24.2]) начинает работу с простейшего фрагмента и на каждом шаге расширяет его, добавляя ребро наименьшего веса, исходящее из построенного к тому времени фрагмента. Алгоритм Краскала (см. [55, раздел 24.2]) начинает работу, располагая семейством фрагментов, состоящих из одного узла, и проводит их соединение, добавляя ребра наименьшего веса, исходящие из того или иного фрагмента. Поскольку в алгоритме Краскала разрешается работать независимо с несколькими фрагментами, он более подходит для реализации в качестве распределенного алгоритма.

7.3.3. Глобальное описание алгоритма GHS.

Вначале мы расскажем о глобальной работе алгоритма, т. е. опишем, как алгоритм обращается с фрагментами. Затем мы приведем локальный алгоритм, описывающий действия, которые должны выполняться в каждом узле сети, чтобы осуществить эти глобальные операции над фрагментами.

Всякое вычисление алгоритма GHS складывается из следующих шагов.

1. Формируется такое семейство фрагментов, что объединение их содержит все узлы сети.

2. Первоначально это семейство состоит из всех узлов сети, каждый из которых рассматривается как граф с одним узлом.

3. Узлы всякого фрагмента вступают во взаимодействие с целью выявления исходящего из фрагмента ребра с наименьшим весом.

4. Как только будет определено исходящее из фрагмента ребро с наименьшим весом, данный фрагмент соединяется с другим фрагментом путем добавления этого исходящего ребра, которое строится в результате взаимодействия этих двух фрагментов.

5. Алгоритм завершает работу, как только останется только один фрагмент. Для эффективной реализации этих шагов потребуется ввести некоторые обозначения и операции.

1. *Имя фрагмента.* Чтобы определить исходящее ребро наименьшего веса, необходимо суметь проверить, выходит ли некоторое ребро за пределы фрагмента или оно ведет в узел того же самого фрагмента. Для этого каждый фрагмент будет наделен именем, которое будет известно всем процессам, входящим в этот фрагмент. Процессы будут проверять, является ли ребро внутренним или внешним ребром, сравнивая имена фрагментов, к которым они относятся.

2. *Соединение старшего и младшего фрагментов.* Когда два фрагмента соединяются, в процессах хотя бы одного из этих фрагментов происходит перемена имени фрагмента; для этого требуется, чтобы изменение имени произошло в каждой точке по крайней мере одного из указанных фрагментов. Чтобы осуществить это изменение эффективно, в основу стратегии соединения фрагментов положен следующий принцип: *младший из двух фрагментов* присоединяется к *старшему из них* и принимает имя старшего фрагмента.

3. *Ранги фрагментов.* После небольшого размышления становится ясно, что решение о том, какой из фрагментов старше, *нельзя* основывать на количестве узлов в этих фрагментах. В таком случае размер фрагмента пришлось бы уточнять в каждом процессе обоих фрагментов-компонент и, таким образом, нарушать взятое обязательство проводить уточнения только в младшем из двух фрагментов. Вместо этого каждому фрагменту сопоставляется *ранг*, который первоначально полагается равным 0 для любого фрагмента, состоящего из одного узла. Некоторый фрагмент F_1 может соединиться с фрагментом F_2 более высокого ранга, после чего новый фрагмент $F_1 \cup F_2$ будет иметь такой же ранг, как и фрагмент F_2 . Этот новый фрагмент унаследует имя фрагмента F_2 , и поэтому в узлах фрагмента F_2 не потребуется проводить никаких уточнений. Неизбежно также и соединение двух фрагментов одного и того же ранга; в этом случае новый фрагмент будет носить новое имя и его ранг будет на единицу превосходить ранг соединенных фрагментов. Именем образовавшегося фрагмента будет вес того ребра, которое соединило два исходных фрагмента; указанное ребро назовем *стержнем* нового фрагмента. Те два узла, которые соединены ребром-стержнем, будут называться *стержневыми узлами*.

Лемма 7.20. *Если соблюдать указанные выше правила соединения фрагментов, то суммарно во всех процессах перемена имени или ранга случится не более $N \log N$ раз.*

Доказательство. Ранг всякого процесса не понижается, и лишь в том случае, когда он повышается, процесс вынужден переменить имя своего фрагмен-

та. Фрагмент ранга L содержит не менее 2^L процессов, и поэтому максимально возможный ранг равен $\log N$. Отсюда следует, что в каждом отдельном процессе ранг содержащего его фрагмента повышается не более $\log N$ раз. Значит, общее число перемен имени и ранга фрагмента ограничено величиной $N \log N$. $\square$

Краткое изложение стратегии соединения. Для обозначения фрагмента F с именем FN и рангом L будем использовать запись $F = (FN, L)$. Обозначим символом e_F исходящее из фрагмента F ребро наименьшего веса.

Правило А. Если e_F ведет в такой фрагмент $F' = (FN', L')$, для которого $L < L'$, то фрагмент F присоединяется к F' и образовавшийся при этом новый фрагмент сохраняет имя FN' и ранг L' . Эти новые значения отправляются всем процессам, входящим в состав F .

Правило В. Если e_F ведет в такой фрагмент $F' = (FN', L')$, для которого $L = L'$ и $e_{F'} = e_F$, то эти два фрагмента соединяются и образуют один новый фрагмент, который будет носить имя $\omega(e_F)$ и иметь ранг $L+1$. Эти новые значения отправляются всем процессам, входящим в состав F и F' .

Правило С. Во всех остальных случаях (т. е. когда либо $L > L'$, либо $L = L'$ и $e_{F'} \neq e_F$) фрагмент F должен ожидать, когда откроется возможность применения правила А или В.

7.3.4. Подробное описание алгоритма GHS

Статус узлов и связей. Как отмечено в описании алгоритма 7.10, в узле p задействовано несколько локальных переменных, среди которых есть переменные статуса каналов $stach_p[q]$ для каждого канала связи pq . Канал может иметь один из следующих трех статусов: *branch*, если известно, что соответствующее ребро входит в MST, *reject*, если известно, что это ребро не входит в MST, и *basic*, если данное ребро еще не было использовано. Обмен сообщениями во фрагменте с целью выявления исходящего ребра наименьшего веса проводится по ребрам типа *branch*. Для всякого процесса p рассматриваемого фрагмента запись $father_p$ будет обозначать то ребро, которое ведет к стержню данного фрагмента. Состоянием $state_p$ узла p может быть либо *find*, если процесс p привлечен к проводимому во фрагменте поиску исходящего ребра наименьшего веса, либо *found* в противном случае. Описание алгоритма GHS состоит из трех частей 7.10/7.11/7.12. Иногда обработка сообщений может быть отложена, до тех пор пока не будет выполнено некоторое локальное условие. В таком случае мы будем подразумевать, что анализируемое сообщение сохраняется в памяти. Впоследствии оно извлекается из памяти и обрабатывается так, как будто оно было только что получено. Если процесс получает сообщение, пребывая все еще в состоянии *sleep*, то перед тем как приступить к обработке этого сообщения, наш алгоритм проходит в этой точке этап инициализации (выполняя действие (1)).

Поиск исходящего ребра наименьшего веса. Все узлы фрагмента участвуют совместно в поиске исходящего из этого фрагмента ребра наименьшего веса, и как только такое ребро будет обнаружено, по нему отправляется сообщение

```

var  $state_p$            : (sleep, find, found) ;
     $stach_p[q]$          : (basic, branch, reject) для каждого  $q \in Neigh_p$  ;
     $name_p, bestwt_p$     : real;
     $level_p$             : integer;
     $testch_p, bestch_p, father_p$  :  $Neigh_p$  ;
     $rec_p$               : integer;

(1) Первым действием каждого процесса является инициализация данного алгоритма:
    begin пусть  $pq$  — это канал связи процесса  $p$ , имеющий наименьший вес ;
         $stach_p[q] := branch$  ;  $level_p := 0$  ;
         $state_p := found$  ;  $rec_p := 0$  ;
        send  $\langle connect, 0 \rangle$  to  $q$ 
    end

(2) После получения сообщения  $\langle connect, L \rangle$  от  $q$ :
    begin if  $L < level_p$  then (* Соединить по правилу А *)
        begin  $stach_p[q] := branch$  ;
            send  $\langle initiate, level_p, name_p, state_p \rangle$  to  $q$ 
        end
    else if  $stach_p[q] = basic$ 
        then (* Правило С *) обработать это сообщение позднее
    else (* Правило В *) send  $\langle initiate, level_p + 1, \omega(pq), find \rangle$  to  $q$ 
    end

(3) После получения сообщения  $\langle initiate, L, F, S \rangle$  от  $q$ :
    begin  $level_p := L$  ;  $name_p := F$  ;  $state_p := S$  ;  $father_p := q$  ;
         $bestch_p := undef$  ;  $bestwt_p := \infty$  ;
        forall  $r \in Neigh_p$  :  $stach_p[r] = branch \wedge r \neq q$  do
            send  $\langle initiate, L, F, S \rangle$  to  $r$  ;
        if  $state_p = find$  then begin  $rec_p := 0$  ;  $test$  end
    end

```

Алгоритм 7.10. Алгоритм Галладжера—Хамблета—Спиры (Часть 1).

$\langle connect, L \rangle$, где L — ранг данного фрагмента. Если фрагмент состоит из единственного узла, как это случается после инициализации такого узла, то нужное ребро — это просто инцидентное данному узлу ребро наименьшего веса (см. действие (1)), и сообщение $\langle connect, 0 \rangle$ отправляется по этому ребру.

Теперь рассмотрим случай, когда новый фрагмент образовался за счет соединения двух фрагментов ребром $e = pq$. Если оба соединенных фрагмента имели одинаковый ранг L , то оба процесса p и q должны были отправить сообщение $\langle connect, L \rangle$ по ребру e и должны были получить в ответ сообщение $\langle connect, L \rangle$, и при этом ребро e должно иметь статус *branch* (см. действие (2)). Ребро pq становится стержнем нового фрагмента, и процессы p и q обмениваются сообщениями $\langle initiate, L + 1, N, S \rangle$, придавая вновь образованному фрагменту новый ранг и новое имя. Новым именем становится вес $\omega(pq)$, а статус *find* заставляет каждый процесс приступить к поиску исходящего ребра наименьшего веса (см. действие (3)). Сообщение $\langle initiate, L + 1, N, S \rangle$ поступает в каждый узел нового фрагмента. Если же ранг процесса p был ниже, чем ранг процесса

```

(4) procedure test:
    begin if  $\exists q \in Neigh_p$  :  $stach_p[q] = basic$  then
        begin  $testch_p := q$  with  $stach_p[q] = basic$  and  $\omega(pq)$  minimal ;
            send  $\langle test, level_p, name_p \rangle$  to  $testch_p$ 
        end
    else begin  $testch_p := undef$  ;  $report$  end
    end

(5) После получения сообщения  $\langle test, L, F \rangle$  от  $q$ :
    begin if  $L > level_p$  then (* С ответом придется подождать! *)
        обработать это сообщение позднее
    else if  $F = name_p$  then (* внутреннее ребро *)
        begin if  $stach_p[q] = basic$  then  $stach_p[q] := reject$  ;
            if  $q \neq testch_p$ 
                then send  $\langle reject \rangle$  to  $q$ 
            else test
        end
    else send  $\langle accept \rangle$  to  $q$ 
    end

(6) После получения сообщения  $\langle accept \rangle$  от  $q$ :
    begin  $testch_p := undef$  ;
        if  $\omega(pq) < bestwt_p$ 
            then begin  $bestwt_p := \omega(pq)$  ;  $bestch_p := q$  end;
         $report$ 
    end

(7) После получения сообщения  $\langle reject \rangle$  от  $q$ :
    begin if  $stach_p[q] = basic$  then  $stach_p[q] := reject$ ;
         $test$ 
    end

```

Алгоритм 7.11. Алгоритм Галладжера—Хамблета—Спиры (Часть 2).

q , то p должен был отправить сообщение $\langle connect, L \rangle$ по ребру e и должен был получить в ответ сообщение $\langle initiate, L', N, S \rangle$ от процесса q (см. действие (2)). Это означает, что L' и N — текущий ранг и текущее имя процесса q . Это имя и этот ранг остаются неизменными у всех процессов, расположенных по ту же сторону от нашего ребра, что и узел q . Полученное от процесса q сообщение распространяется среди всех процессов, расположенных по ту же сторону от стержня, что и узел p , и вынуждает каждый такой процесс переменить имя и ранг фрагмента (см. действие (3)). Если q пребывает в поиске исходящего ребра наименьшего веса ($S = find$), то процессы в той части фрагмента, где расположен процесс p , присоединяются к поиску, вызывая процедуру *test*.

Каждый процесс в рассматриваемом фрагменте проводит поиск по всем исходящим из него ребрам (если таковые имеются, см. действия (4), (5), (6), и (7)), стремясь отыскать такое ребро, которое выводит за пределы данного фрагмента, и если такие ребра будут обнаружены, то процесс выбирает среди них ребро наименьшего веса. Информация о ребре наименьшего веса передается каждо-

```

(8) procedure report:
  begin if  $rec_p = \#\{q : stach_p[q] = branch \wedge q \neq father_p\}$ 
    and  $testch_p = undef$  then
      begin  $state_p := found$  ; send  $\langle report, bestwt_p \rangle$  to  $father_p$  end
    end
(9) После получения сообщения  $\langle report, \omega \rangle$  от  $q$ :
  begin if  $q \neq father_p$ 
    then (* ответить на сообщение initiate *)
      begin if  $\omega < bestwt_p$  then
        begin  $bestwt_p := \omega$  ;  $bestch_p := q$  end;
         $rec_p := rec_p + 1$  ; report
      end
    else (* ребро  $pq$  — стержень *)
      if  $state_p = find$ 
        then process this message later
      else if  $\omega > bestwt_p$ 
        then changeroot
      else if  $\omega = bestwt_p = \infty$  then stop
    end(10) procedure changeroot:
      begin if  $stach_p[bestch_p] = branch$ 
        then send  $\langle changeroot \rangle$  to  $bestch_p$ 
        else begin send  $\langle connect, level_p \rangle$  to  $bestch_p$  ;
           $stach_p[bestch_p] := branch$ 
        end
      end
    end
(11) После получения сообщения  $\langle changeroot \rangle$ :
  begin changeroot end

```

Алгоритм 7.12. Алгоритм Галладжера—Хамблета—Спиры (Часть 3)

му поддереву при помощи сообщения $\langle report, \omega \rangle$ (см. действие (8)). В узле p подсчитывается число полученных сообщений $\langle report, \omega \rangle$, с использованием переменной rec_p . Этой переменной присваивается значение 0, когда начинается поиск (см. действие (3)), и ее значение увеличивается на единицу при поступлении каждого очередного сообщения $\langle report, \omega \rangle$ (см. действие (9)). Каждый процесс отправляет сообщение $\langle report, \omega \rangle$ в родительский узел, как только он получит такого рода сообщения от всех своих преемников и завершит свой собственный локальный поиск исходящего ребра.

Сообщения $\langle report, \omega \rangle$ отправляются каждым процессом в направлении ребра-стержня, и сообщения от обоих стержневых узлов пойдут навстречу друг другу по этому ребру; оба узла получают сообщения от своих родительских узлов $father$ (см. действие (9)). Каждый стержневой процесс не обрабатывает сообщение, полученное от его напарника, до тех пор пока он сам не отправит ему сообщение $\langle report, \omega \rangle$. Как только оба сообщения $\langle report, \omega \rangle$ от стержневых узлов пройдут навстречу друг другу, стержневые узлы будут осведомлены о том, каков наименьший вес исходящего из фрагмента ребра. Здесь алгоритм и завершает работу,

если не будет обнаружено ни одного исходящего ребра (в обоих сообщениях будет фигурировать значение ∞).

Если же исходящее ребро наименьшего веса имеется, то отыскать это ребро можно, следуя в каждом узле по указателю $bestch$ и начиная движение из стержневого узла, расположенного на той же стороне данного фрагмента, на которой располагается искомое ребро. По этому ребру должно быть отправлено сообщение $\langle connect, L \rangle$, и указатели $father$ во всех узлах данного фрагмента должны указывать в этом направлении. Чтобы осуществить это, используется сообщение $\langle changeroot \rangle$. Стержневой узел, расположенный в той части фрагмента, где находится исходящее ребро наименьшего веса, отправляет сообщение $\langle changeroot \rangle$, которое следует по дереву к ребру наименьшего веса (см. действия (10) и (11)). Когда сообщение $\langle changeroot \rangle$ достигает узла, инцидентного исходящему ребру наименьшего веса, этот узел отправляет сообщение $\langle connect, L \rangle$ по указанному ребру.

Проверка ребер. Узел p , стремясь отыскать исходящее из него ребро наименьшего веса, просматривает один за другим в порядке возрастания веса всех примыкающих к нему ребер, имеющих статус *basic* (см. действие (4)). Локальный поиск ребра завершается в том случае, когда либо не найдется ни одного такого ребра (все ребра имеют статус *reject* или *branch*, см. действие (4)), либо одно из ребер идентифицируется как исходящее (см. действие (6)). Порядок, в котором процесс p просматривает ребра, гарантирует, что первое обнаруженное им исходящее из фрагмента ребро будет иметь наименьший вес среди всех исходящих из p ребер, выходящих за пределы данного фрагмента.

Для анализа ребра pq процесс p отправляет сообщение $\langle test, level_p, name_p \rangle$ процессу q и ожидает ответа, каковым может быть одно из трех сообщений $\langle reject \rangle$, $\langle accept \rangle$ или $\langle test, L, F \rangle$. Процесс q отправляет сообщение $\langle reject \rangle$ (см. действие (5)), если он обнаружит, что имя того фрагмента, в котором расположен узел q , совпадает с именем того фрагмента, в котором расположен узел p ; в этом случае процесс q также придаст этому ребру статус $\langle reject \rangle$. После получения сообщения $\langle reject \rangle$ процесс p также будет считать ребро pq отвергнутым и продолжит локальный поиск (см. действие (7)). Сообщение $\langle reject \rangle$ может быть опущено, если ребро pq было только что так же задействовано процессом q для отправления сообщения $\langle test, L, F \rangle$. В этом случае сообщение $\langle test, L, F \rangle$, полученное от q , истолковывается как отклик на сообщение, отправленное процессом p (см. действие (5)). Если имя фрагмента, к которому относится узел q , отличается от имени фрагмента, в котором расположен узел p , то в ответ отправляется сообщение $\langle accept \rangle$. При получении сообщения $\langle accept \rangle$, процесс p завершает свой локальный поиск исходящего из фрагмента ребра, имея в качестве результата ребро pq (см. (6)).

Обработка сообщения $\langle test, L, F \rangle$ процессом p откладывается, если $L > level_p$. Причина этого заключается в том, что узлы p и q могут на самом деле принадлежать одному и тому же фрагменту, но сообщение $\langle initiate, L + 1, N, S \rangle$ еще не достигло узла p . Тогда узел p мог бы ошибочно отправить узлу q сообщение $\langle accept \rangle$.

Соединение фрагментов. Как только будет определено исходящее из фрагмента $F = (name, level)$ ребро наименьшего веса, по нему отправляется сообщение $\langle \text{connect}, level \rangle$, которое будет получено узлом, принадлежащим фрагменту $F' = (name', level')$. Обозначим символом p процесс, отправивший указанное сообщение, а символом q процесс, который принял это сообщение. Узел q ранее уже отправлял сообщение $\langle \text{accept} \rangle$ процессу p в ответ на сообщение $\langle \text{test}, level, name \rangle$, поскольку поиск наилучшего исходящего ребра во фрагменте, к которому относится узел p , был завершен. Ожидание, проходящее перед ответом на тестовое сообщение (см. действие (5)), позволяет заключить, что $level' \geq level$.

Согласно правилам соединения, которые обсуждались ранее, на запрос $\langle \text{connect}, level \rangle$ будет отправлено ответное сообщение $\langle \text{initiate}, L + 1, N, S \rangle$ в двух случаях.

Вариант А. Если $level' > level$, то фрагмент, содержащий узел p , поглощается; все узлы этого фрагмента оповещаются о новом имени и новом ранге фрагмента посредством сообщения $\langle \text{initiate}, level', name', S \rangle$, которое распространяется по всем узлам фрагмента F . Весь поглощаемый фрагмент F становится поддеревом, присоединенным к узлу q , в остовном дереве фрагмента F' , и если узел q привлечен к поиску наилучшего ребра, исходящего из фрагмента F' , то все процессы, относящиеся к фрагменту F , должны принять участие в этом поиске. Поэтому процесс q включает свое состояние (*find* или *found*) в сообщение $\langle \text{initiate}, level', name', S \rangle$.

Вариант В. Если два фрагмента имеют одинаковый ранг и наилучшим ребром, исходящим из фрагмента F' , также является pq , то образуется новый фрагмент, ранг которого повышается на единицу, а его именем становится вес ребра pq (см. действие (2)). Такой случай возможен, если два ранга равны и сообщение о соединении получено по ребру, имеющему статус *branch*. Следует обратить внимание на то, что ребро приобретает статус *branch*, если по нему отправляется сообщение о соединении.

Во всех остальных случаях фрагмент F должен пребывать в ожидании того, что либо q отправит сообщение $\langle \text{connect}, L \rangle$, либо ранг фрагмента, содержащего q , повысится настолько, чтобы стал осуществим вариант А.

Корректность и сложность. Из подробного описания алгоритма должно быть ясно, что ребро, по которому из фрагмента отправляется сообщение $\langle \text{connect}, L \rangle$, действительно является исходящим из фрагмента ребром наименьшего веса. Отсюда, а также из утверждения 7.19 следует, что MST будет построено правильно, если каждый фрагмент действительно отправляет такое сообщение и соединяется с другим фрагментом, несмотря на ожидание, предусмотренное данным алгоритмом. Наиболее сложное сообщение включает в себя информацию о весе одного ребра, информацию о значении одного ранга (не более $\log N$ битов) и некоторое фиксированное число битов для оповещения о типе сообщения и состоянии узла.

Теорема 7.21. Алгоритм Галладжера—Хамблета—Спиры (алгоритм 7.10/7.11/7.12) вычисляет минимальное остовное дерево, используя при этом не более $5N \log N + 2|E|$ обменов сообщениями.

Доказательство. Потенциальная возможность для взаимной блокировки возникает в тех ситуациях, когда узлы или фрагменты должны пребывать в ожидании того, что определенные условия будут выполнены в другом узле или в другом фрагменте. Ожидание в стержневых узлах, связанное с сообщениями $\langle \text{report}, \omega \rangle$, не приводит к блокировке, потому что каждый стержневой узел рано или поздно получит сводки от всех своих преемников (если только весь фрагмент целиком не пребывает в ожидании другого фрагмента), после чего указанное сообщение будет обработано.

Рассмотрим случай, когда сообщение, отправленное из фрагмента $F_1 = (level_1, name_1)$ достигает какого-то узла фрагмента $F_2 = (level_2, name_2)$. Сообщение $\langle \text{connect}, level_1 \rangle$ должно ожидать обработки, если $level_1 \geq level_2$ и никакое сообщение $\langle \text{connect}, level_2 \rangle$ не было отправлено по тому же самому ребру из фрагмента F_2 (см. действие (2)). Сообщение $\langle \text{test}, level_1, name_1 \rangle$ должно ожидать обработки, если $level_1 > level_2$ (см. действие (5)). Во всех тех случаях, когда фрагмент F_1 ожидает фрагмент F_2 , выполняется одно из следующих условий:

- 1) $level_1 > level_2$;
- 2) $level_1 = level_2 \wedge \omega(e_{F_1}) > \omega(e_{F_2})$;
- 3) $level_1 = level_2 \wedge \omega(e_{F_1}) = \omega(e_{F_2})$ и F_2 все еще пребывает в поиске исходящего из этого фрагмента ребра наименьшего веса. (Коль скоро e_{F_1} является ребром, исходящим из F_2 , неравенство $\omega(e_{F_2}) > \omega(e_{F_1})$ выполняться не будет.)

Таким образом, взаимная блокировка по замкнутому циклу невозможна.

Каждое ребро может быть отвергнуто не более одного раза, и для этого требуется два сообщения; поэтому суммарное число отвергающих сообщений и тестовых сообщений, приводящих к отвержению, ограничено величиной $2|E|$. Каков бы ни был ранг фрагмента, узел получает не более одного иницирующего и одного допускающего сообщения и отправляет не более одной сводки (*report*), не более одного сообщения о перемене корня или о соединении и не более одного тестового сообщения, не приводящего к отвержению ребра. Если ранг равен нулю, то не происходит отправок допускающих сообщений и получения сводок или тестов. Каждый узел, обладающий самым высоким рангом, отправляет только сводку и получает только одно иницирующее сообщение. Таким образом, общее число обменов сообщениями ограничено величиной $2|E| + 5N \log N$. $\square$

7.3.5. Обсуждение GHS-алгоритма и его модификаций

Алгоритм Галладжера—Хамблета—Спиры, использующий только локальные сведения и обладающий оптимальной сложностью по числу обменов сообщениями, является одним из самых изощренных волновых алгоритмов. Этот алгоритм можно без труда обобщить так, чтобы он проводил избрание лидера, используя всего лишь два дополнительных сообщения. Данный алгоритм завершается в двух точках, каковыми являются стержневые точки последнего фрагмента (т. е. остовного дерева всей сети). Вместо выполнения команды останова алгоритма стержневые узлы сообщают друг другу свои отличительные признаки, и узел с наименьшим признаком становится лидером.

Был опубликован целый ряд модификаций рассмотренного алгоритма. Алгоритму GHS требуется время $\Omega(N^2)$, если некоторые узлы запустили алгоритм слишком поздно. Если использовать дополнительную процедуру побудки (для которой требуется не более $2|E|$ дополнительных сообщений) временная сложность данного алгоритма станет равна $5N \log N$ (см. упражнение 7.11). Авербах в работе [16] показал, что временная сложность алгоритма может быть понижена до $O(N)$, и при этом сложность по числу обменов сообщениями остается оптимальной, т. е. равной $O(|E| + N \log N)$.

Афек и др. в работе [1] приспособили данный алгоритм для вычисления остовного леса, обладающего тем свойством, что диаметр каждого дерева и общее число деревьев оцениваются величиной $O(\sqrt{N})$. Их алгоритм разбивает сеть на кластеры, как это описано в лемме 4.47, и вычисляет остовное дерево и центр для каждого кластера.

Можно задаться следующим вопросом: а не удастся ли построить произвольное остовное дерево более эффективно, чем это делается в случае минимальных остовных деревьев? Но из теоремы 7.15 следует, что нижняя оценка $\Omega(N \log N + |E|)$ сохраняется также и для произвольных деревьев. Йохансен и др. в работе [113] предложили алгоритм построения произвольного вычислительного дерева для разреженных сетей с использованием $3N \log N + 2|E| + O(N)$ обменов сообщениями, уменьшив, таким образом, постоянный множитель в алгоритме GHS. Бар-Илан и Церник [22] разработали алгоритм построения *случайного* остовного дерева, в котором всевозможные остовные деревья выбираются с равной вероятностью. Это рандомизованный алгоритм, для которого математическое ожидание числа используемых обменов сообщениями располагается в интервале от $O(N \log N + |E|)$ до $O(N^3)$ в зависимости от топологии сети.

Хотя для произвольных сетей сложность построения произвольного и минимального дерева одинакова, для клика это уже не так. Корач, Моран и Закс [120] показали, что для построения минимального остовного дерева во взвешенной клике требуется совершить $\Omega(N^2)$ обменов сообщениями. Этот результат свидетельствует о том, что учет топологии не позволяет уменьшить сложность поиска минимального остовного дерева ниже того уровня, который установлен в теореме 7.15. В следующем параграфе мы покажем, что произвольное остовное дерево в клике можно построить с использованием $O(N \log N)$ обменов сообщениями (см. также работу [119]).

7.4. Алгоритм Корача—Каттена—Морана

Было получено много результатов, касающихся задачи о выборах, и не только для случая кольцевых и произвольных сетей, но и для сетей других специальных топологий, таких как клики и т. п. В отдельных случаях наилучшие известные алгоритмы достигают сложности $O(N \log N)$ по числу обменов сообщениями, а иногда их сложность соответствует нижней оценке $\Omega(N \log N)$. Корач, Каттен и Моран в работе [118] показали, что между задачей о выборах и задачей обхода сети существует тесная взаимосвязь. Их основным результатом является общий

метод построения эффективного алгоритма избрания лидера для произвольного класса сетей на основе всякого алгоритма обхода сетей из того же класса.

Этот метод позволяет строить алгоритмы избрания лидера сложности $O(N \log N)$ для многих классов сетей. Поскольку наилучший возможный алгоритм обхода имеет сложность $O(x)$ (линейную сложность), более эффективных алгоритмов избрания при помощи этого метода построить нельзя. Напротив, как мы увидим в гл. 11, для сетей с заданной ориентацией существуют и более качественные алгоритмы избрания лидера. К тому же у алгоритма Корача—Каттена—Морана сложность по времени и сложность по числу обменов сообщениями одинаковы, тогда как известны примеры других алгоритмов избрания лидера, имеющих такую же сложность по числу обменов сообщениями, но меньшую сложность по времени. Интерес к данному методу обусловлен общностью этого метода и той взаимосвязью, благодаря которой алгоритм обхода используется в качестве отдельного «модуля» при решении задачи более высокого уровня — избрания лидера.

7.4.1. Модульная конструкция

В основу алгоритма Корача—Каттена—Морана положены идеи двух методов: метода угасания (см. § 7.3.1) и алгоритма Петерсона/Долева—Клейва—Роде (см. § 7.2.2). Сходство с методом угасания проявляется в том, что инициаторы выборов приступают к обходу сети, используя маркеры, помеченные их отличительными признаками. Если обход завершится (будет принято решение), то инициатор этого обхода будет считаться избранным; из самого описания алгоритма следует, что это случится в точности для одного обхода. В этом разделе мы опишем данный алгоритм для случая, когда в каналах поддерживается очередность, но если добавить совсем немного информации, которая будет переноситься маркерами и обрабатываться процессами, то указанный алгоритм можно будет использовать и в том случае, когда очередность сообщений в каналах не гарантируется (см. [118]).

Чтобы справиться с той ситуацией, когда есть несколько инициаторов, алгоритм оперирует на разных *уровнях*, подобно тому как алгоритм Петерсона/Долева—Клейва—Роде разбивает свое вычисление на туры. Если запущены по крайней мере два обхода, то один из маркеров рано или поздно достигнет того процесса, в котором уже успел побывать другой. Если сложится такая ситуация, то обход, который проводится при помощи вновь прибывшего маркера, будет прерван. Цель работы нашего алгоритма теперь состоит в том, чтобы привести две маркера вместе в один процесс, где они будут подавлены, и после этого начнется новый обход. Этот прием напоминает аналогичный прием в алгоритме Патерсона/Долева—Клейва—Роде, когда не более чем один из двух отличительных признаков имеет возможность уцелеть и перейти в другой тур. Вместо туров в алгоритме Корача—Каттена—Морана используются уровни; две маркера дают начало новому обходу только в том случае, если они находятся на одном и том же уровне, и при этом уровень вновь запущенного обхода будет на единицу больше. Если какой-либо маркер встречается с другим маркером более высокого уровня или достигает узла, который уже посетил маркер более высокого уровня, то этот

прибывший маркер просто подавляется без всякого влияния на маркер более высокого уровня.

```

var  $lev_p$       : integer    init  $-1$  ;
     $cat_p, wait_p$  :  $\mathcal{P}$       init  $undef$  ;
     $last_p$       :  $Neigh_p$    init  $undef$  ;
begin if  $p$  is initiator then
  begin  $lev_p := lev_p + 1$  ;  $last_p := trav(p, lev_p)$  ;
         $cat_p := p$  ; send  $\langle annex, p, lev_p \rangle$  to  $last_p$ 
  end;
  while ... (* Условие завершения, см. описание *) do
    begin receive token  $(q, l)$  ;
      if token is annexing then  $t := A$  else  $t := C$  ;
      if  $l > lev_p$  then (* Вариант I *)
        begin  $lev_p := l$  ;  $cat_p := q$  ;  $wait_p := undef$  ;  $last_p := trav(q, l)$  ;
              send  $\langle annex, q, l \rangle$  to  $last_p$ 
        end
      else if  $l = lev_p$  and  $wait_p \neq undef$  then (* Вариант II *)
        begin  $wait_p := undef$  ;  $lev_p := lev_p + 1$  ;  $last_p := trav(p, lev_p)$  ;  $cat_p := p$  ;
              send  $\langle annex, p, lev_p \rangle$  to  $last_p$ 
        end
      else if  $l = lev_p$  and  $last_p = undef$  then (* Вариант III *)
         $wait_p := q$ 
      else if  $l = lev_p$  and  $t = A$  and  $q = cat_p$  then (* Вариант IV *)
        begin  $last_p := trav(q, l)$  ;
              if  $last_p = decide$ 
                then  $p$  объявляет себя лидером else send  $\langle annex, q, l \rangle$  to  $last_p$ 
        end
      else if  $l = lev_p$  and  $((t = A$  and  $q > cat_p)$  or  $t = C)$  then (* Вариант V *)
        begin send  $\langle chase, q, l \rangle$  to  $last_p$  ;  $last_p := undef$  end
      else if  $l = lev_p$  then  $wait_p := q$  (* Вариант VI *)
    end
  end
end

```

Алгоритм 7.13. Алгоритм Корача—Каттена—Морана

Мы будем рассматривать алгоритм 7.13. Чтобы привести маркеры одного и того же уровня совместно в один процесс, каждому маркеру предписывается один из трех режимов: *захват*, *погоня* или *ожидание*. Всякий маркер будем представлять в виде пары (q, l) , где q — это инициатор данного маркера, а l — это его уровень. Переменная lev_p будет обозначать уровень, на котором пребывает процесс p , а переменная cat_p представляет инициатора того последнего маркера, который был захвачен процессом p , но уцелел и был переправлен этим процессом далее (это и есть *текущий активный обход* процесса p). Переменная $wait_p$ принимает значение $undef$, если ни один из маркеров не пребывает в режиме ожидания в процессе p , и значение этой переменной равно q , если некоторый маркер (q, lev_p) пребывает в режиме ожидания в процессе p . Переменная $last_p$

предназначена для маркеров, пребывающих в режиме погони: она указывает на того соседа, которому процесс p переправил захваченный им маркер уровня lev_p , если только догоняющий маркер уже не был отправлен вслед за ним в погоню; в таком случае $last_p = undef$. Наш алгоритм взаимодействует с алгоритмом обхода за счет обращения к функции $trav$: эта функция либо указывает того соседа, которому нужно переправить маркер, либо заявляет о принятии решения, если данный обход завершился.

Маркер (q, l) иницируется в режиме *захвата*, и в этом режиме он подчиняется алгоритму обхода (как и в случае IV для алгоритма 7.13), до тех пор пока не сложится одна из следующих ситуаций.

1. Алгоритм обхода завершен, и в таком случае процесс q становится лидером (см. вариант IV в описании алгоритма 7.13).
 2. Маркер достиг такого узла p , для которого выполняется неравенство $lev_p > l$; в этом случае наш маркер подавляется. (Этот вариант подразумевается в алгоритме 7.13; все условия в этом алгоритме требуют выполнения соотношения $l > lev_p$ или $l = lev_p$).
 3. Маркер достиг такого узла, в котором его уже ожидает другой маркер уровня l ; тогда оба маркера подавляются, и из этого узла начинается новый обход (см. вариант II в описании алгоритма 7.13).
 4. Маркер достиг узла уровня l , который посетил последним некоторый маркер с отличительным признаком $cat_p > q$ (см. вариант VI) или маркер в режиме *погони* (см. вариант III); тогда наш маркер переходит в режим *ожидания* и остается в этом узле.
 5. Маркер достиг узла уровня l , в котором в последний раз побывал некоторый маркер в режиме *захвата* с отличительным признаком $cat_p < q$; тогда наш маркер переходит в режим *погони* и переправляется по тому же каналу, по которому был отправлен предыдущий маркер (см. вариант V).
- Маркер (q, l) в режиме *погони* переправляется в каждом узле по тому каналу, по которому был отправлен самый последний из посетивших этот узел маркеров, если только не складывается одна из следующих ситуаций.

1. Маркер достиг процесса уровня $lev_p > l$; тогда наш маркер подавляется.
 2. Маркер достиг процесса, в котором хранится маркер уровня l , пребывающий в режиме *ожидания*; тогда оба маркера изымаются и в этом процессе начинается новый обход (см. вариант II).
 3. Маркер достиг процесса уровня l , и при этом последний из посетивших этот процесс маркеров пребывал в режиме *погони*; тогда наш маркер переходит в режим *ожидания* (см. вариант III).
- Маркер в режиме *ожидания* остается в том процессе, которого он достиг, до тех пор пока не сложится одна из следующих ситуаций.
1. Маркер более высокого уровня достигнет того же самого процесса; тогда ожидающий маркер будет подавлен (см. вариант I).
 2. Маркер того же самого уровня достиг того же самого процесса; тогда оба маркера изымаются, и в данном процессе начинается новый обход более высокого уровня (см. вариант II).

В алгоритме 7.13 все переменные и переносимые маркерами данные, касающиеся алгоритма обхода, не принимались во внимание. Отметим, что если процесс p получает маркер, уровень которого превосходит lev_p , то этот маркер пребывает в режиме захвата и его инициатором является процесс, отличный от p . Если же обход завершается в узле p , то p становится лидером и отправляет сообщения всем процессам, призывая их прекратить работу.

Корректность и сложность. Для доказательства корректности алгоритма Корака—Каттена—Морана будет показано, что число маркеров, порожденных на каждом уровне, сокращается, до тех пор пока на некотором уровне не будет выпущен только один маркер, инициатор которого и становится лидером.

Лемма 7.22. *Если на уровне l было порождено $k > 1$ маркеров, то на уровне $l + 1$ будет порождено не менее одного и не более $k/2$ маркеров.*

Доказательство. Поскольку при порождении одного маркера уровня $l + 1$ подавляются два маркера уровня l , всего на уровне $l + 1$ может быть выпущено не более $k/2$ маркеров.

Предположим, что найдется такой уровень l , что на уровне l было порождено $k > 1$ маркеров, а на уровне $l + 1$ ни одного. Обозначим символом q процесс с *наибольшим* отличительным признаком, который породил маркер на уровне l . Маркер (q, l) не завершает обход, так как он достигает некоторого процесса p , который уже отправил другой маркер уровня l . Когда это произойдет впервые, маркер (q, l) станет преследователем или, если процесс p уже отправил какой-то преследующий маркер, перейдет в режим ожидания. В любом случае это означает, что в сети есть преследующий маркер уровня l .

Рассмотрим маркер (r, l) с *наименьшим* отличительным признаком, за которым был отправлен в погоню один из преследующих маркеров. Сам маркер (r, l) не может быть преследователем, поскольку преследователи гонятся за маркерами с меньшим отличительным признаком. Поэтому мы можем предполагать, что маркер (r, l) переходит в режим ожидания, как только он достигнет процесса p' , который уже отправил какой-либо другой маркер уровня l . Пусть p'' — *последний* из процессов на пути, по которому следует маркер (r, l) , причем этот процесс получил после отправления (r, l) маркер, пребывавший в режиме захвата, который перешел затем в разряд преследователей маркера r . Этот преследующий маркер либо настигнет маркер (r, l) в узле p' , либо прервет погоню, если обнаружит какой-нибудь маркер, пребывающий в режиме ожидания, ранее чем будет достигнут узел p' . В обоих случаях, вопреки предположению, будет выработан маркер уровня $l + 1$. $\square$

Теорема 7.23. *Алгоритм Корака—Каттена—Морана (алгоритм 7.13) является алгоритмом избрания лидера.*

Доказательство. Допустим, что хотя бы один процесс запустил этот алгоритм. Согласно предыдущей лемме число маркеров, порожденных на каждом уровне, уменьшается, и поэтому найдется уровень, например уровень l , на котором будет порожден всего лишь один маркер, скажем (q, l) . Ни один из мар-

керов уровня $l' < l$ не завершил обход, и поэтому ни один из них не привел к тому, что некоторый процесс был избран лидером. Единственный маркер уровня l сталкивается только с такими процессами, уровень которых уступает l или для которых выполняется равенство $cat = q$ (если этот маркер достигает процесса, который он уже посетил ранее). В обоих случаях процессы переправляют его далее. Следовательно, обход завершится, и процесс q будет избран лидером. $\square$

Функцию f назовем *выпуклой*, если выполняется неравенство $f(a) + f(b) \leq f(a + b)$. Мы оценим здесь сложность нашего алгоритма исходя из предположения, что в его основу положен алгоритм $f(x)$ -обхода (см. § 6.6.3), где f — выпуклая функция.

Теорема 7.24. *Если использовать произвольный алгоритм $f(x)$ -обхода, где f — выпуклая функция, то алгоритм избрания лидера Корака—Каттена—Морана, будучи запущен k процессами-инициаторами, решит задачу о выборах, ограничившись $(1 + \log k)[f(N) + N]$ обменами сообщениями.*

Доказательство. Если алгоритм был запущен k процессами, то на уровне l может быть порождено не более $2^{-l}k$ маркеров, откуда следует, что наивысший уровень, который может быть достигнут, не превосходит величины $\lfloor \log k \rfloor$.

На любом уровне каждый процесс отправляет маркер в режиме захвата самое большее с одним отличительным признаком. Если на некотором уровне l есть маркеры с отличительными признаками $p_1, \dots, p_j$ и имеется N_i процессов, отправивших маркеры (p_i, l) , пребывающие в режиме захвата, то справедливо

неравенство $\sum_{i=1}^j N_i \leq N$. Так как используемый алгоритм обхода — это алгоритм $f(x)$ -обхода, маркеры (p_i, l) , остающиеся в режиме захвата, были переправлены не более $f(N_i)$ раз, и поэтому общее число обменов сообщениями, сопряженных с передачей маркеров уровня l , пребывающих в режиме захвата, не превосходит величины $\sum_{i=1}^j f(N_i)$, которая, в свою очередь, не превышает $f(N)$, ибо функция f выпуклая. На любом уровне каждый процесс отправляет не более одного догоняющего маркера, и, следовательно, общее количество догоняющих маркеров на каждом уровне не превосходит N .

Итак, мы имеем не более $(1 + \log k)$ различных уровней, на каждом из которых совершается самое большее $f(N) + N$ обменов сообщениями. Отсюда следует указанная в этой теореме оценка. $\square$

Метод Аттьи. Другой метод построения алгоритма избрания лидера на основе алгоритмов обхода был предложен Аттьей в работе [9]. В этом алгоритме обход, совершаемый некоторым маркером с отличительным признаком q , не прерывается в том случае, когда этот маркер достигает какого-либо процесса r , который уже посетил другой маркер, выпущенный, например, процессом p . Вместо этого маркер-захватчик переходит в режим ожидания в узле r , но зато посылает в погоню за маркером p «охотничий» маркер, который возвращается в узел r , если маркер p был успешно подавлен. Когда охотник возвращается,

нет нужды начинать новый обход; можно продолжить текущий обход, совершаемый маркером q , экономя тем самым используемые сообщения. Чтобы охотник мог вернуться в узел r , требуется двунаправленность сети. Если использовать произвольный алгоритм $f(x)$ -обхода, то в результате будет построен алгоритм избрания лидера, имеющий сложность по числу обменов сообщениями, приблизительно равную величине $3 \sum_{i=1}^N f(N/i)$.

7.4.2. Приложения алгоритма Корача—Каттена—Морана

Если для некоторого класса сетей существует алгоритм $f(x)$ -обхода, то этот класс будем называть $f(x)$ -обозримым. Так как многие классы сетей относятся к числу $O(x)$ -обозримых, предложенный метод позволяет строить алгоритмы избрания сложности $O(N \log N)$ по числу обменов сообщениями, и это самое лучшее, что может быть получено на основе указанного результата. Впрочем, в отдельных случаях при наличии ориентации в сети можно построить и более качественные алгоритмы (см. гл. 11).

Выборы в кольцах. Все кольца x -обозримы, и, следовательно, алгоритм Корача—Каттена—Морана проводит выборы лидера в кольце с использованием $2N \log N$ обменов сообщениями. Мы уже знаем из теоремы 7.13, что это наилучшая возможная оценка.

Выборы в кликах. Все клики $2x$ -обозримы, причем ориентация здесь не нужна, и, следовательно, алгоритм ККМ проводит выборы лидера в клике с использованием $3N \log N$ обменов сообщениями в соответствии с теоремой 7.24. Более тщательный анализ этого алгоритма показывает, что на самом деле в данном случае его сложность оценивается величиной $2N \log N + O(N)$. Чтобы догнать любой маркер достаточно использовать не более трех сообщений, поэтому общее число сообщений в режиме погони, отправленных по ходу вычисления, ограничено величиной $3 \sum_{i=0}^{\log N + 1} 2^{-i} N = O(N)$. Во время написания этой книги автору не было известно никаких алгоритмов избрания в кликах, имеющих лучшую оценку сложности, нежели $2N \log N + O(N)$. Нижняя оценка сложности $\Omega(N \log N)$ была установлена Корачем, Мораном и Заксом [119].

Указанные результаты остаются справедливыми только для клик, не наделенных ориентацией; при наличии ориентации можно построить алгоритмы линейной сложности.

Выборы в торах. Торовые сети с ориентацией являются x -обозримыми (см. алгоритм 6.10), и, следовательно, алгоритм ККМ проводит выборы лидера в торе с использованием $2N \log N$ обменов сообщениями. При отсутствии ориентации для обхода можно воспользоваться алгоритмом Тарри, который решает задачу обхода тора за линейное время. Петерсон в работе [160] предложил алгоритм избрания лидера на решетках и торах, который использует $O(N)$ обменов сообщениями и не требует разметки ребер.

7.5. Упражнения к главе 7

7.5.1

Упражнение 7.1. Докажите, что алгоритм избрания путем сравнения является волновым алгоритмом, если событие избрания процесса лидером рассматривать как событие решения.

Упражнение 7.2. Покажите, что сложность по времени алгоритма 7.1 составляет $2D$.

7.5.2

Упражнение 7.3. Докажите тождество $\sum_{i=1}^m \mathbf{H}_i = (m+1)\mathbf{H}_m - m$, используемое в § 7.2.1.

Упражнение 7.4. Покажите, что $\ln(N+1) < \mathbf{H}_N < \ln(N) + 1$. (Здесь $\ln$ обозначает натуральный логарифм.)

Упражнение 7.5. Рассмотрим алгоритм Ченя—Робертса, полагая, что каждый процесс является инициатором. При каком расположении отличительных признаков в кольце сложность по числу обменов сообщениями будет минимальной и сколько обменов сообщениями потребуется в этом случае?

Упражнение 7.6. Какова будет сложность в среднем алгоритма Ченя—Робертса, если ограничиться теми случаями, когда есть в точности S инициаторов, причем любой выбор одного из S процессов в качестве инициаторов одинаково возможен?

Упражнение 7.7. Приведите начальную конфигурацию для алгоритма 7.7, при которой алгоритму действительно потребуется провести $\lfloor \log N \rfloor + 1$ туров. Приведите также начальную конфигурацию, при которой этому алгоритму потребуется всего два тура, независимо от числа инициаторов. Может ли алгоритм завершить работу за один тур?

Упражнение 7.8. Определите множество E_{CR} (как это сделано непосредственно перед леммой 7.10) для алгоритма Ченя—Робертса.

7.5.3

Упражнение 7.9. Примените метод угасания к кольцевому алгоритму и сравните полученный алгоритм с алгоритмом Ченя—Робертса. В чем состоит различие и как оно проявляется?

Упражнение 7.10. Для каждого из семи типов сообщений, фигурирующих в алгоритме Галладжера—Хамблета—Спиры определите, может ли сообщение этого типа поступить в узел, пребывающий в состоянии *sleep*.

Упражнение 7.11. Предположим, что в алгоритме GHS применяется дополнительная процедура побудки, которая гарантирует, что этот алгоритм начнет свою работу в каждом узле не позднее чем через N единиц времени.

Докажите методом индукции, что спустя самое большее $5Nl - 3N$ единиц времени каждый узел будет иметь ранг l .

Докажите, что этот алгоритм завершит работу спустя не более $5N \log N$ единиц времени.

7.5.4

Упражнение 7.12. Покажите, что для планарных сетей существует алгоритм избрания сложности $O(N \log N)$.

Упражнение 7.13. Покажите, что для торов, *не наделенных* ориентацией, существует алгоритм избрания сложности $O(N \log N)$. (**Указание:** проведите анализ производительности алгоритма Тарри на торах.)

Упражнение 7.14. Покажите, что для гиперкубов, *не наделенных* ориентацией, существует алгоритм избрания сложности $O(N \log N)$.

Упражнение 7.15. Покажите, что для сетей, степень узлов которых ограничена величиной k (это означает, что каждый узел имеет не более k соседей), существует алгоритм избрания сложности $O(N(\log N + k))$.

ГЛАВА 8

ОБНАРУЖЕНИЕ ЗАВЕРШЕНИЯ

Вычисление распределенного алгоритма завершается, когда этот алгоритм достигает некоторой *заключительной конфигурации*, т. е. такой конфигурации, из которой данный алгоритм не может сделать ни одного шага. Это не всегда означает, что в заключительной конфигурации каждый процесс пребывает в одном из *заключительных состояний*, т. е. в таком состоянии, в котором не может произойти никаких событий. Рассмотрим, например, такую ситуацию, когда каждый процесс пребывает в состоянии готовности к приему сообщений, а все каналы пусты. Эта конфигурация является заключительной, хотя указанные состояния могли бы с равным успехом оказаться и промежуточными состояниями вычисления. В данном случае процессы не осведомлены о том, что вычисление уже завершилось; мы будем говорить, что произошло *неявное завершение* вычисления. Мы будем говорить также, что в процессе произошло *явное завершение* вычисления, если то состояние, в котором пребывает этот процесс в заключительной конфигурации, является заключительным состоянием данного процесса. Неявное завершение вычисления называют также *завершением обмена сообщениями*, потому что никакой обмен сообщениями невозможен, если достигнута заключительная конфигурация. Явное завершение вычисления называют также *завершением работы процессов*, потому что работа процессов завершается, если алгоритм явно завершает работу.

Обычно проще спроектировать алгоритм, завершающий вычисления неявно, чем алгоритм, работа которого завершается явно. Действительно, при проектировании алгоритма можно не обращать внимания на те детали, которые касаются завершения работы процессов; наше внимание сосредоточено на том, чтобы ограничить общее число событий, которые могут произойти по ходу вычисления. С другой стороны, в приложениях алгоритма может потребоваться, чтобы процессы завершали работу явно. Только после явного завершения работы можно будет сбросить значения используемых переменных и рассматривать результаты вычисления как окончательные. Кроме того, к заключительной конфигурации может привести и *блокировка* распределенного алгоритма; в этом случае при достижении заключительной конфигурации вычисление должно быть перезапущено.

В настоящей главе будут исследованы общие методы преобразования алгоритмов, в которых вычисления оканчиваются по завершении обмена сообщениями, в алгоритмы, в которых вычисления оканчиваются завершением работы процессов. Располагая такими методами, можно проектировать алгоритм, принимая в расчет лишь неявное завершение его вычислений (т. е. следя лишь за тем, чтобы

все вычисления нашего алгоритма были конечны), после чего, используя один из этих методов, привести алгоритм к такому виду, в котором вычисления оканчиваются по завершении работы процессов. Указанные методы предусматривают введение двух дополнительных алгоритмов, которые взаимодействуют друг с другом, а также с заданным алгоритмом, имеющим неявное завершение вычислений. Один из этих алгоритмов анализирует текущее вычисление исходного алгоритма и каким-либо образом обнаруживает, что это вычисление достигло заключительной конфигурации. После этого вызывается второй алгоритм, который рассылает сообщение о завершении вычисления всем процессам, призывая их перейти в заключительное состояние.

Самой сложной частью такого преобразования оказывается алгоритм, обнаруживающий завершение вычисления. Процедура рассылки весьма проста, и мы обсудим ее в § 8.1.3. В этой главе будет показано, что обнаружение завершения возможно для всех классов сетей, для которых можно разработать волновой алгоритм. Эти классы включают сети, допускающие избрание лидера, сети, процессы которых обладают отличительными признаками, древесные сети, а также сети, в которых процессам доступны сведения о таких топологических параметрах, как диаметр сети или число процессов. С другой стороны, в гл. 9 будет показано, что для безымянных сетей, размер которых заранее неизвестен процессам, существует неявно завершающийся алгоритм вычисления максимума входных данных всех процессов, но нет явно завершающегося алгоритма решения указанной задачи. И это означает, что обнаружение завершения невозможно для безымянных сетей, размер которых неизвестен.

В тех случаях, когда обнаружение завершения возможно, мы установим нижние оценки для числа сообщений, которыми обмениваются процессы при работе алгоритма обнаружения завершения. Будет показано также, что существуют алгоритмы, сложность которых соответствует указанным оценкам. В разд. 8.5.1 дается формальная постановка задачи на основе модели поведения распределенных вычислений, устанавливаются нижние оценки и описывается процедура рассылки оповещающих сообщений. В § 8.5.2 приводятся несколько решений поставленной задачи, предполагающих использование дерева (или леса) процессов, включающего все процессы, в которых вычисление еще продолжается. Решения, приведенные в этом параграфе, не очень сложны и соответствуют нижним оценкам из § 8.5.1. В этих двух параграфах представлены все основополагающие результаты, касающиеся существования и сложности алгоритмов обнаружения завершения. По ряду причин один алгоритм обнаружения завершения может оказаться для тех или иных приложений предпочтительнее другого. Поэтому в §§ 8.5.3 и 8.5.4 представлен ряд других решений рассматриваемой задачи.

8.1. Основные понятия

8.1.1. Определения

В этом параграфе мы введем в рассмотрение модель распределенных вычислений, опираясь на которую будем исследовать задачу завершения распределен-

ных вычислений. В основу этой модели положена модель вычислений, введенная в гл. 2, однако, мы будем опускать все детали, не имеющие отношения к проблеме обнаружения завершения вычислений.

Множество всех состояний Z_p процесса p разбивается на два подмножества *активных* и *пассивных* состояний. Состояние c_p процесса p считается активным, если процесс p в состоянии c_p может выполнить внутреннее действие или отправить сообщение; в противном случае это состояние считается пассивным. В пассивном состоянии c_p возможен только прием сообщений, или в нем вообще не будет происходить никаких событий, и в этом случае c_p будет относиться к заключительным состояниям процесса p . Говорят, что процесс p активен, если он пребывает в одном из активных состояний; в противном случае процесс p называют пассивным. Ясно, что отправлять сообщения может только активный процесс, а пассивный процесс может стать активным только после получения сообщения. Активный процесс может стать пассивным, если он перейдет в пассивное состояние. Чтобы облегчить описание алгоритмов в этой главе, сделаем ряд упрощений.

1. *Активный процесс становится пассивным только после осуществления внутреннего события.* Всякий процесс легко видоизменить так, чтобы он удовлетворял указанному требованию. Рассмотрим такое событие (c, m, d) отправления (или получения) сообщения процессом p , в котором состояние d является пассивным. Заменим это событие в процессе p на (c, m, d') , где d' — новое состояние, в котором возможно только одно событие (d', d) , и это событие является внутренним. Так как состояние d' активное, процесс p становится пассивным в результате осуществления внутреннего события (d', d) .

2. *Процесс всегда становится активным после получения сообщения.* Всякий процесс легко видоизменить так, чтобы он удовлетворял указанному требованию. Рассмотрим такое событие (c, m, d) получения сообщения процессом p , в котором состояние d является пассивным. Заменим в процессе p это событие на (c, m, d') , где d' — новое состояние, в котором может произойти только одно событие (d', d) , и это событие является внутренним. Так как состояние d' активное, процесс p становится активным после получения сообщения, а пассивным в результате осуществления внутреннего события (d', d) .

3. *Внутренние события, приводящие к тому, что процесс p становится пассивным, — это единственно возможные внутренние события в процессе p .* Внутренние события, переводящие процесс p из одного активного состояния в другое активное состояние, игнорируются, поскольку алгоритм обнаружения завершения не должен заниматься *выведыванием*; ему не разрешается использовать информацию о том, как организованы локальные вычисления процессов. Таким образом, для алгоритма обнаружения завершения все внутренние состояния процессов неразличимы.

Сведения о каком-либо состоянии p ограничиваются лишь знанием того, является это состояние активным или пассивным; эта информация представлена значением переменной $state_p$. Все переходы вычисления представлены на примере алгоритма 8.1. Как обычно, предполагается, что в начальной конфигурации

никакое сообщение не находится на этапе пересылки. Первоначально процессы могут быть как активными, так и пассивными.

```
var statep : (active, passive) ;
```

```
Sp: { statep = active }
  begin send ⟨mes⟩ end
```

```
Rp: { Сообщение ⟨mes⟩ доставлено процессу p }
  begin receive ⟨mes⟩ ; statep := active end
```

```
Ip: { statep = active }
  begin statep := passive end
```

Алгоритм 8.1. Базовый распределенный алгоритм

Чтобы отличать анализируемый алгоритм от алгоритма обнаружения завершения, мы будем называть его *базовым алгоритмом*, и вычисление базового алгоритма будем называть *базовым вычислением* или *основным вычислением*. Алгоритм обнаружения завершения будет носить название *контрольного* или *надзирающего алгоритма*, а его вычисление будем называть *контрольными* или *надзирающим* вычислением. Подобным же образом сообщения будут подразделяться на *базовые сообщения* и *контрольные сообщения*.

Будем полагать, что предикат **term** обращается в истину на всякой конфигурации, на которой не может произойти ни одного события в базовом вычислении; согласно приведенной ниже теореме это возможно только в том случае, когда все процессы пассивны и ни одно базовое сообщение не находится на этапе пересылки.

Теорема 8.1. *Справедливо следующее соотношение:*

$$\text{term} \iff (\forall p \in \mathbb{P} : \text{state}_p = \text{passive}) \\ \wedge (\forall pq \in E : M_{pq} \text{ не содержит сообщения } \langle \text{mes} \rangle).$$

Доказательство. Если все процессы пассивны, никакое внутреннее событие или событие отправления сообщения невозможно. Если, кроме того, ни по одному из каналов не передается какое-либо сообщение **mes**, то ни одно событие приема сообщения также не может произойти. Значит, невозможно вообще никакое базовое событие.

Если какой-нибудь процесс активен, то в этом процессе может произойти внутреннее событие или событие отправления сообщения, а если в одном из каналов содержится какое-либо сообщение **mes**, то может осуществиться прием этого сообщения. □

В заключительной конфигурации базового алгоритма каждый процесс ожидает поступления сообщения и будет всегда пребывать в этом ожидании. Задача, которую мы обсуждаем в этой главе, состоит в том, чтобы присоединить к нашей

системе контрольный алгоритм, который приведет все процессы в заключительные состояния, после того как базовое вычисление достигнет заключительной конфигурации. Для составного алгоритма (образованного за счет присоединения контрольного алгоритма к базовому) конфигурация, удовлетворяющая предикату **term**, не обязательно является заключительной; в общем случае в ней могут произойти события контрольного алгоритма. В контрольном алгоритме происходит обмен (контрольными) сообщениями. Эти сообщения могут быть отправлены пассивными процессами, и получение контрольного сообщения не будет переводить пассивный процесс в разряд активных.

Контрольный алгоритм состоит из алгоритма *обнаружения завершения вычисления* и алгоритма *оповещения о завершении вычисления*. Алгоритм обнаружения завершения вычисления вызывает алгоритм *Announce*, и этот алгоритм оповещения переводит все процессы в заключительные состояния. Всякий рассматриваемый алгоритм обнаружения завершения вычисления должен удовлетворять следующим трем требованиям.

1. **Невмешательство.** Алгоритм обнаружения завершения вычисления не должен оказывать влияния на вычисления базового алгоритма.

2. **Живость.** Если выполняется условие **term**, то алгоритм *Announce* должен быть вызван спустя конечное число шагов.

3. **Безопасность.** Если вызван алгоритм *Announce*, то конфигурация должна удовлетворять условию **term**.

Задача обнаружения завершения вычисления была впервые поставлена Франчесом в работе [87]. Франчес предложил решение, которое, однако, не удовлетворяло требованию невмешательства; его решение предусматривало первоначальное «замораживание» базового вычисления путем отмены всех событий, а затем проведение анализа сложившейся конфигурации с целью определения того, является ли она заключительной. Если конфигурация оказывалась заключительной, то вызывался алгоритм *Announce*; в противном случае базовое вычисление «размораживалось» и описанная процедура повторялась некоторое время спустя. Перечисленные выше требования не допускают такого решения и заставляют выявлять завершение вычисления «на лету», т. е. по ходу работы базового алгоритма. В этой главе при проведении доказательства корректности алгоритмов обнаружения завершения мы не касаемся вопроса о невмешательстве, поскольку из описания рассматриваемого алгоритма ясно видно, что это требование соблюдается.

Базовое вычисление называется централизованным, если в каждой начальной конфигурации в точности один процесс является активным, и децентрализованным, если число активных процессов в начальной конфигурации может быть произвольным. Централизованные базовые вычисления в работах, посвященных задаче обнаружения завершения, часто называют *диффузными вычислениями*. Контрольное вычисление называется централизованным, если для его выполнения выделен один специальный процесс, и децентрализованным, если все процессы выполняют один и тот же контрольный алгоритм.

8.1.2. Две нижние оценки

Сложность алгоритмов обнаружения завершения вычисления выражается, как и прежде, посредством параметров N и $|E|$, а также числа M сообщений, которыми обмениваются процессы при выполнении базового вычисления. Сложность обнаружения завершения вычисления связана также с затратами на выполнение волнового алгоритма; мы обозначим символом W сложность по числу обменов сообщениями наилучшего волнового алгоритма. Безусловно, величина W зависит от таких характеристик рассматриваемого класса сетей, как топология сети, наличие процесса-лидера, а также от сведений, которыми изначально располагают процессы.

Мы покажем, что в наихудшем случае сложность задачи обнаружения завершения вычисления, как для централизованных так и для децентрализованных алгоритмов ограничена снизу величиной M . Мы покажем также, что сложность задачи обнаружения завершения децентрализованных базовых вычислений ограничена снизу величиной W . Мы завершим этот параграф обсуждением нижней оценки величины $|E|$ по числу обменов сообщениями, установленную Чандрасекараном и Венкатесаном в работе [43].

Теорема 8.2. *Для всякого алгоритма обнаружения завершения вычисления существует такое базовое вычисление, осуществляющее M обменов базовыми сообщениями, для обнаружения завершения которого рассматриваемый алгоритм совершает не менее M обменов контрольными сообщениями.*

Доказательство. Если наша система способна достичь такой конфигурации, начиная с которой в контрольном алгоритме может происходить обмен бесконечно большим числом сообщений, и при этом не будет осуществляться ни одного базового события, то утверждение теоремы, очевидно, справедливо. Поэтому далее в доказательстве этой теоремы мы будем полагать, что контрольный алгоритм откликается только конечным числом сообщений на каждое базовое событие.

Рассмотрим конфигурацию γ , в которой активны два процесса p и q и нет никаких сообщений, находящихся на этапе пересылки. Если базовый алгоритм является централизованным, то такая конфигурация может быть достигнута из начальной конфигурации посредством обмена одним базовым сообщением; в противном случае в число таких конфигураций входит и начальная конфигурация.

Вначале рассмотрим такое вычисление, в котором, начиная с конфигурации γ , оба процесса становятся пассивными после параллельного выполнения своих действий, т. е. система достигает конфигурации $\delta = I_p(I_q(\gamma))$. Завершение вычисления должно быть обнаружено спустя конечное число шагов, но при этом ни p , ни q не могут вызвать процедуру *Announce*, не получив предварительно сообщения от своего напарника, ибо в противном случае произойдет ошибочное обнаружение завершения вычисления в конфигурации, в которой один из процессов все еще остается активным. А если завершение вычисления обнаруживает третий процесс, то для этого необходимы по меньшей мере два сообщения. Значит,

прежде чем завершение базового вычисления будет обнаружено, должен быть совершен обмен по крайней мере одним контрольным сообщением в конфигурации δ .

Не ограничивая общности, предположим, что в конфигурации δ процесс p отправляет контрольное сообщение. Рассмотрим вычисление, в котором, начиная с конфигурации γ , пассивным становится только процесс p , т. е. система достигает конфигурации $\gamma_p = I_p(\gamma)$. Состояние процесса p будет одним и тем же в конфигурациях γ_p и δ , и, следовательно, p также отправит контрольное сообщение в конфигурации γ_p . Контрольный алгоритм может проработать еще некоторое время, однако завершения базового вычисления он не обнаружит, поскольку процесс q все еще остается активным. После того как контрольный алгоритм прекратит обмен сообщениями, процесс q отправит базовое сообщение процессу p , и, таким образом, мы вернемся к конфигурации, в которой оба процесса p и q будут активны. Контрольный алгоритм снова вступит в действие, но после конечного числа шагов вновь будет достигнута конфигурация, в которой оба процесса p и q будут активны и никакие сообщения не будут находиться на этапе пересылки. Подводя итог сказанному, отметим следующее:

- 1) конфигурация, в которой оба процесса p и q активны и нет сообщений на этапе пересылки, может быть достигнута из начальной конфигурации после обмена не более чем одним базовым сообщением;
- 2) базовый алгоритм может перейти из одной такой конфигурации в другую после обмена всего лишь одним сообщением, вынуждая контрольный алгоритм совершить обмен по меньшей мере одним контрольным сообщением;
- 3) если базовое вычисление после пребывания в такой конфигурации завершится, то для обнаружения этого понадобится совершить обмен по крайней мере одним контрольным сообщением.

Из этих замечаний и следует утверждение теоремы. $\square$

Теорема 8.3. *Для обнаружения завершения децентрализованного базового вычисления в худшем случае требуется совершить обмен не менее чем W контрольными сообщениями.*

Доказательство. Рассмотрим базовое вычисление, в котором вообще не происходит обмена сообщениями и каждый активный процесс становится пассивным после осуществления первого же события. Для обнаружения того, что такое базовое вычисление завершилось, требуется применить волновой алгоритм, если истолковывать обнаружение завершения вычисления (вызов процедуры *Announce*) как событие решения. И в самом деле, вызов процедуры *Announce* должен произойти спустя конечное число шагов, откуда следует, что сам алгоритм обнаружения завершится и примет решение. Если решению не будет предшествовать хоть одно событие в каком-либо процессе, то окажется возможным другое базовое вычисление, в котором процесс q не станет пассивным. Событие решения не будет связано причинно-следственной зависимостью ни с одним событием в процессе q , а это означает, что алгоритм обнаружения завершения вычисления может ошибочно вызвать процедуру *Announce* в то время, когда процесс q

еще остается активным. И поскольку алгоритм обнаружения завершения вычисления оказывается волновым, ему потребуется совершить не менее W обменов сообщениями. $\square$

Запуск алгоритма обнаружения завершения вычисления. Чандрасекаран и Венкатесан в работе [43] сумели установить нижнюю оценку числа обменов сообщениями, равную $|E|$, налагая следующие два дополнительных ограничения.

C1. В каналах поддерживается очередность сообщений.

C2. Алгоритм обнаружения завершения вычисления может начать работу в любой момент, после того как было запущено базовое вычисление, т.е. начиная с произвольной конфигурации базового вычисления.

Если соблюдаются эти допущения, то алгоритм обнаружения завершения вычисления, не отправивший контрольного сообщения по некоторому особо выделенному ребру, например pq , будет вынужден принять неверное решение. Непосредственно перед запуском контрольного алгоритма базовое вычисление отправляет одно дополнительное сообщение по каналу pq . Это сообщение не будет зарегистрировано контрольным алгоритмом, в результате чего он придет к ошибочному выводу о завершении вычисления. Алгоритм Чандрасекарана и Венкатесана отправляет контрольные сообщения по каждому каналу, вследствие чего все сообщения, отправленные перед запуском контрольного алгоритма, будут доставлены по назначению еще до того, как будет принято решение о завершении вычисления.

Проводя рассуждения, аналогичные тем, которые применялись в работе [43], можно показать, что задача вообще не имеет решения, если соблюдается допущение C2, а допущение C1 не имеет места. В этой главе мы будем предполагать, что контрольный алгоритм начинает работу в начальной конфигурации базового вычисления, т.е. в базовом вычислении до начала работы контрольного алгоритма не происходит ни одного незарегистрированного события. Если принять указанное предположение вместо допущения C2, то наша задача будет иметь решение тогда и только тогда, когда во всех каналах соблюдается очередность сообщений, и такое решение найдено в работе [43].

8.1.3. Как остановить процессы

Чтобы оповестить о завершении вычисления все процессы, по сети широко-ковещательно распространяется сообщение $\langle \text{stop} \rangle$. Каждый процесс отправляет такое сообщение всем своим соседям, но делает это не более одного раза, либо совершая локальный вызов процедуры *Announce*, либо после получения первого сообщения $\langle \text{stop} \rangle$. Когда сообщение $\langle \text{stop} \rangle$ будет получено от всех соседей, процесс выполняет оператор **stop** и в результате этого переходит в заключительное состояние. Указанный метод оповещения представлен в алгоритме 8.2.

Алгоритм 8.2 можно применять для связанных сетей произвольной топологии, включая ориентированные сети; в нем не предусматривается наличия лидера, отличительных признаков и какой-либо осведомленности об особенностях топологии.

```
var  $SentStop_p$  : bool   init false ;
     $RecStop_p$    : integer init 0 ;
```

Procedure *Announce*:

```
begin if not  $SentStop_p$  then
    begin  $SentStop_p := true$  ;
        forall  $q \in Out_p$  do send  $\langle \text{stop} \rangle$  to  $q$ 
    end
end
```

{Сообщение $\langle \text{stop} \rangle$ поступило процессу p }

```
begin receive  $\langle \text{stop} \rangle$  ;  $RecStop_p := RecStop_p + 1$  ;
    Announce ;
    if  $RecStop_p = \#In_p$  then halt
end
```

Алгоритм 8.2. Алгоритм оповещения

8.2. Деревья и леса вычислений

Решения задачи обнаружения завершения вычисления, описанные в этом параграфе, основаны на построении и динамическом обновлении ориентированного графа, который называется *графом вычисления*. В число вершин этого графа входят все активные процессы и все базовые сообщения, находящиеся на этапе пересылки. Завершение вычисления будет обнаружено, как только граф вычисления становится пустым. Все те решения задачи, которые приведены в настоящем параграфе, требуют, чтобы сеть была неориентированной, т.е. чтобы сообщения по каждому каналу можно было отправлять в обе стороны. В §8.2.1 описано решение рассматриваемой задачи для централизованных базовых вычислений, в которых граф вычислений представляет собой дерево, корнем которого является инициатор. В §8.2.2 приведено обобщение этого решения для децентрализованных базовых вычислений с использованием леса, в котором каждый инициатор базового вычисления является корнем одного из деревьев.

8.2.1. Алгоритм Дейкстры—Шолтена

Алгоритм Дейкстры—Шолтена [65] обнаруживает завершение *централизованного* базового вычисления (такое вычисление в работе [65] называют *диффузным вычислением*). Инициатор базового вычисления (который в работе [65] называется *окружением*) играет также особую роль в алгоритме обнаружения завершения вычисления; мы будем обозначать этот процесс символом p_0 .

Алгоритм обнаружения завершения строит и постоянно обновляет дерево вычисления $T = (V_T, E_T)$, которое обладает следующими двумя свойствами.

1. Граф T либо является пустым, либо представляет собой ориентированное дерево, корнем которого является вершина p_0 .

2. Множество V_T включает все активные процессы и все базовые сообщения, находящиеся на этапе пересылки.

Инициатор p_0 вызывает процедуру *Announce*, если $p_0 \notin V_T$; согласно первому свойству граф T в таком случае является пустым, а согласно второму свойству это означает, что вычисление завершилось.

Чтобы указанные свойства дерева вычислений сохранялись по мере продвижения базового вычисления, граф T должен быть расширен в случае отправления базового сообщения или в том случае, когда процесс, не входящий в состав дерева, становится активным. Когда процесс p отправляет базовое сообщение $\langle \text{mes} \rangle$, это сообщение добавляется к дереву в качестве вершины $\langle \text{mes} \rangle$, причем родительской вершиной для нее становится процесс p . Когда процесс p , не входящий в состав дерева, становится активным после получения сообщения от процесса q , вершина q становится родительской вершиной для процесса p . Чтобы обозначить явно отправителя сообщения, всякое базовое сообщение $\langle \text{mes} \rangle$, отправленное процессом q , будет обозначаться записью $\langle \text{mes}, q \rangle$.

```

var statep : (active, passive)  init if p = p0 then active else passive ;
   scp      : integer           init 0 ;
   fatherp : ℙ                 init if p = p0 then p else undef ;

```

S_p : { state_p = active }

begin send $\langle \text{mes}, p \rangle$; sc_p := sc_p + 1 end

R_p : {Сообщение $\langle \text{mes}, q \rangle$ достигло процесса p }

begin receive $\langle \text{mes}, q \rangle$; state_p := active ;

if father_p = undef then father_p := q else send $\langle \text{sig}, q \rangle$ to q

end

I_p : { state_p = active }

begin state_p := passive ;

if sc_p = 0 then(* Удалить p из T *)

begin if father_p = p

then Announce

else send $\langle \text{sig}, \text{father}_p \rangle$ to father_p ;

father_p := undef

end

end

A_p : { Сигнал $\langle \text{sig}, p \rangle$ достигает p }

begin receive $\langle \text{sig}, p \rangle$; sc_p := sc_p - 1 ;

if sc_p = 0 and state_p = passive then

begin if father_p = p

then Announce

else send $\langle \text{sig}, \text{father}_p \rangle$ to father_p ;

father_p := undef

end

end

Удалять вершины из дерева T также необходимо, и на то есть две причины. Во-первых, базовое сообщение удаляется, как только оно будет получено. Во-вторых, чтобы контрольный алгоритм сработал, дерево должно исчезнуть спустя конечное число шагов после завершения базового алгоритма. Сообщения являются листьями дерева T ; в каждом процессе есть переменная, используемая для подсчета числа вершин-последователей этого процесса в дереве T . Удаление из дерева вершины, являющейся последователем процесса p , осуществляется совсем в другом процессе q ; это происходит в результате того, что вершина-последователь либо соответствует сообщению, которое достигло процесса q , либо соответствует самому процессу q . Чтобы в процессе p не нарушался учет вершин-последователей, этому процессу может быть отправлено сигнальное сообщение $\langle \text{sig}, p \rangle$, когда удаляется вершина-последователь процесса p . Это сообщение замещает удаленную вершину-последователя процесса p , а удаление этой вершины из дерева осуществляется самим процессом p после получения указанного сигнала; при этом p уменьшает значение счетчика последователей на единицу.

Все эти действия описаны в алгоритме 8.3. Каждый процесс p наделен переменной father_p . Значение этой переменной полагается равным *undef*, если $p \notin V_T$. Если же p является корнем дерева, то значением переменной father_p является имя процесса p . И наконец, если p является некорневой вершиной дерева T , то значением переменной father_p служит имя родительской вершины данного процесса. Переменная sc_p служит для подсчета числа последователей вершины p в дереве T .

При обосновании корректности алгоритма мы покажем строго формально, что определенный таким образом граф T является деревом и при этом становится пустым только в том случае, когда базовое вычисление завершилось. Для всякой конфигурации γ алгоритма 8.3 определим два множества

$$V_T = \{p: \text{father}_p \neq \text{undef}\} \cup \{\langle \text{mes}, p \rangle \text{ на этапе пересылки}\} \\ \cup \{\langle \text{sig}, p \rangle \text{ на этапе пересылки}\}$$

и

$$E_T = \{(p, \text{father}_p): \text{father}_p \neq \text{undef} \wedge \text{father}_p \neq p\} \\ \cup \{(\langle \text{mes}, p \rangle, p): \langle \text{mes}, p \rangle \text{ на этапе пересылки}\} \\ \cup \{(\langle \text{sig}, p \rangle, p): \langle \text{sig}, p \rangle \text{ на этапе пересылки}\}.$$

Свойство безопасности нашего алгоритма обеспечивается предпосылкой P , которая определяется следующим образом:

$$P \equiv \text{state}_p = \text{active} \implies p \in V_T \quad (1)$$

$$\wedge (u, v) \in E_T \implies u \in V_T \wedge v \in V_T \cap \mathbb{P} \quad (2)$$

$$\wedge sc_p = \#\{v: (v, p) \in E_T\} \quad (3)$$

$$\wedge V_T \neq \emptyset \implies T \text{ — дерево с корнем } p_0 \quad (4)$$

$$\wedge (\text{state}_p = \text{passive} \wedge sc_p = 0) \implies p \notin V_T. \quad (5)$$

Содержательный смысл этого инварианта таков. Согласно определению множество вершин графа T включает все сообщения (как базовые, так и контрольные), а также, согласно соотношению (1), все активные процессы. Соотношение (2)

играет вспомогательную роль; в нем утверждается, что T действительно является графом и все дуги направлены к процессам. Соотношение (3) гласит о том, что подсчет вершин-последователей проводится каждым процессом корректно, а в соотношении (4) утверждается, что T является деревом и процесс p_0 — это его корень. Соотношение (5) служит для того, чтобы показать, что это дерево действительно исчезает, если базовое вычисление завершится. Для доказательства корректности нужно иметь в виду, что из предпосылки P следует, что равенство $father_p = p$ соблюдается только тогда, когда $p = p_0$.

Лемма 8.4. *Предпосылка P является инвариантом алгоритма Дейкстры—Шолтена.*

Доказательство. Первоначально для всех процессов p , отличных от p_0 , выполняется равенство $state_p = passive$, и при этом верно, что $father_{p_0} \neq undef$; отсюда следует соотношение (1). Кроме того, $E_T = \emptyset$, и отсюда следует соотношение (2). Соотношение (3) выполняется, поскольку $sc_p = 0$ для каждого процесса p . Так как $V_T = \{p_0\}$ и $E_T = \emptyset$, граф T является деревом с корнем p_0 , и это подтверждает соотношение (4). В множестве V_T есть только один процесс p_0 , и он является активным.

S_p . При выполнении блока S_p никакой процесс не становится активным, ни один процесс не удаляется из множества V_T , и поэтому соотношение (1) сохраняет свою справедливость.

Применимость действий этого блока означает, что родительская вершина p вновь появившейся вершины $\langle mes, p \rangle$ принадлежит множеству V_T , и это доказывает, что соотношение (2) соблюдается.

В результате выполнения указанного действия к множеству V_T добавляется новая вершина $\langle mes, p \rangle$, а к множеству E_T — новая дуга $(\langle mes, p \rangle, p)$. Это означает, что граф T будет по-прежнему оставаться деревом, а значение переменной sc_p правильным образом увеличилось на единицу в соответствии с появлением у вершины p новой вершины-последователя. Таким образом, соотношения (3) и (4) сохраняются.

Ни один из процессов не становится пассивной листовой вершиной в дереве T , и ни один из процессов не заносится в множество V_T ; поэтому соотношение (5) сохраняет свою справедливость.

R_p . Процесс p либо уже содержался в множестве V_T (в случае $father_p \neq undef$), либо был занесен в V_T в результате выполнения рассматриваемого действия. Поэтому соотношение (1) сохраняется.

Если переменная $father_p$ изменяет значение, то ее новым значением будет q , а если сигнал послан процессом p , то его родительской вершиной также является процесс q , и поскольку q содержится в V_T , соотношение (2) сохраняется.

У процесса q число вершин-последователей не изменяется, ввиду того что последователь $\langle mes, q \rangle$ вершины q замещается либо вершиной-последователем p , либо вершиной-последователем $\langle sig, q \rangle$. А это означает, что значение sc_q остается корректным и соотношение (3) по-прежнему выполняется.

Структура графа не изменяется, и поэтому соотношение (4) сохраняется.

В любом случае после выполнения рассматриваемого действия процесс p оказывается принадлежащим множеству V_T , и, следовательно, соотношение (5) сохраняется.

I_p . Если процесс p становится пассивным, то соотношения (1), (2), (3) и (4) сохраняются. То, что процесс p ранее был активным, означает, что он входил в состав множества V_T . Если $sc_p = 0$, то p удаляется из V_T и соотношение (5) будет также выполнено.

Далее мы рассмотрим, что произойдет, когда процесс p будет удален из графа T , т. е. когда процесс p станет пассивной листовой вершиной графа T .

Если сигнал был отправлен процессом p , то родительской вершиной для вершины сигнала в нашем графе является последняя родительская вершина для p . Эта вершина содержится в множестве V_T , и, следовательно, соотношение (2) сохраняется. В этом случае указанный сигнал занимает место процесса p в качестве последователя процесса $father_p$. Поэтому значение sc_{father_p} остается неизменным, и соотношение (3) сохраняет справедливость. Структура графа при этом не изменяется, и, значит, соотношение (4) также остается верным.

В противном случае, т. е. когда имеет место равенство $father_p = p$, выполняется условие $p = p_0$, и, коль скоро p является листовой вершиной, процесс p оставался единственной вершиной графа T . Поэтому после удаления этой вершины граф T становится пустым, и соотношение (4) сохраняет справедливость.

A_p . Получение сигнала приводит к уменьшению на единицу числа вершин-последователей процесса p . Поскольку при этом выполняется соответствующий оператор присваивания, соотношение (3) сохраняет справедливость. То, что процесс p был родительской вершиной для вершины, соответствующей указанному сигналу, означает, что p принадлежит множеству V_T . Если выполнено условие $state_p = passive$ и после получения сигнала будет выполняться равенство $sc_p = 0$, то вершина p будет удалена. Поэтому соотношение (5) остается верным.

Если процесс p будет удален из множества V_T , то инвариант сохранится в силу тех же самых соображений, которые были приведены, когда речь шла о действии I_p . $\square$

Теорема 8.5. *Алгоритм Дейкстры—Шолтена (алгоритм 8.3) является корректным алгоритмом обнаружения завершения вычисления; в нем используется M обменов контрольными сообщениями.*

Доказательство. Обозначим символом S суммарное значение всех счетчиков числа вершин-последователей, т. е. $S = \sum_{p \in P} sc_p$. Первоначально S равно нулю. Значение S увеличивается при отправлении базового сообщения, которое совершает действие S_p , и уменьшается, когда происходит прием контрольного сообщения (см. действие A_p). При этом S всегда остается неотрицательным числом (это следует из соотношения (3)). Значит, в каждом вычислении число контрольных сообщений не превосходит числа базовых сообщений.

Чтобы обосновать необходимое условие корректности алгоритма, предположим, что базовое вычисление уже завершилось. После такого завершения вычисления могут выполняться только действия A_p . А так как S уменьшается на единицу при совершении каждого такого перехода, алгоритм достигнет заклю-

чительной конфигурации. Отметим, что в этой конфигурации в множестве V_T отсутствуют сообщения. Кроме того, согласно соотношению (5) в множестве V_T отсутствуют пассивные листовые вершины. Следовательно, дерево T не имеет листовых вершин, а это означает, что граф T является пустым. Указанное дерево становится пустым, как только процесс p_0 удаляет из этого дерева самого себя, а программа устроена так, что на этом шаге процесс p_0 вызывает процедуру *Announce*.

Чтобы обосновать достаточное условие корректности алгоритма, заметим, что только процесс p_0 может вызвать процедуру *Announce* и это может быть сделано только тогда, когда p_0 удаляет самого себя из множества V_T . Когда это происходит, граф T согласно соотношению (4) становится пустым. Отсюда следует, что выполняется предикат **term**. $\square$

В алгоритме Дейкстры—Шолтена достигнут замечательный баланс между контрольными и базовыми сообщениями; для каждого базового сообщения, отправленного процессом p процессу q , рассмотренный алгоритм отправляет в точности одно сообщение от процесса q к процессу p . Сложность по числу контрольных сообщений достигает нижней оценки, указанной в теореме 8.2, и поэтому данный алгоритм является оптимальным по сложности в наихудшем случае алгоритмом обнаружения завершения централизованных вычислений.

При описании алгоритма, приведенном в этом параграфе, во всех сообщениях явно указывались их родительские вершины. Однако обычно в этом нет необходимости, так как родительской вершиной базового (или контрольного) сообщения всегда является процесс-отправитель (соответственно процесс-адресат) этого сообщения.

8.2.2. Алгоритм Шави—Франчеза

Для случая децентрализованных базовых вычислений алгоритм Дейкстры—Шолтена был обобщен Шави и Франчезом в работе [175]. В предложенном ими алгоритме граф вычислений является *лесом*, каждое дерево которого имеет в качестве корня один из инициаторов базового вычисления. Для обозначения дерева, корнем которого служит процесс p , будем использовать запись T_p .

В алгоритме Шави—Франчеза строится такой граф $F = (V_F, E_F)$, что (1) либо F пуст, либо F представляет собой лес, каждое дерево которого имеет один из инициаторов в качестве корня, и (2) множество V_F содержит все активные процессы и все базовые сообщения¹⁾. Как и в алгоритме Дейкстры—Шолтена, завершение базового вычисления будет обнаружено, когда указанный граф становится пустым. К сожалению, если граф представляет собой лес, то совсем не просто убедиться в том, что этот граф стал пустым. И в самом деле, поскольку корнем дерева вычислений служит процесс p_0 , именно процесс p_0 может оценить, является ли дерево пустым, и вызвать в этом случае процедуру *Announce*. Когда мы имеем дело с лесом, каждый инициатор может установить пустоту своего

¹⁾Пребывающие на этапе пересылки. — Прим. перев.

собственного дерева, но это еще не будет означать, что весь лес стал пустым графом.

Проверка того, что все деревья исчезли, проводится при помощи одной-единственной волны. Упомянутый лес строится так, чтобы всякое дерево T_p , ставшее пустым, оставалось пустым и в дальнейшем. Отметим, что это не препятствует процессу p вновь становиться активным; однако если процесс p становится активным после исчезновения его собственного дерева, то он заносится в дерево другого инициатора. Каждый процесс принимает участие в распространении волны только тогда, когда его дерево исчезает; и если волна приводит к тому, что принимается решение, то вызывается процедура оповещения *Announce*. (Вызов процедуры *Announce* будет излишним, если волновой алгоритм устроен так, что решение принимает каждый процесс; в таком случае процесс просто останавливается после принятия решения, и этим завершается работа волнового алгоритма.)

Все эти действия описаны в алгоритме 8.4; в приведенном описании волновой алгоритм в явном виде не выделен. Каждый процесс p располагает переменной $father_p$. Значение этой переменной полагается равным *undef*, если $p \notin V_T$. Если же p является корнем дерева, то значением переменной $father_p$ является имя процесса p . И наконец, если p является некорневой вершиной дерева T , то значением переменной $father_p$ служит имя родительской вершины данного процесса. Переменная sc_p используется для подсчета числа последователей вершины p в дереве T . Булева переменная $empty_p$ принимает значение **true** тогда и только тогда, когда дерево, в котором содержится вершина p , становится пустым.

Доказательство корректности рассматриваемого алгоритма весьма сходно с доказательством корректности алгоритма Дейкстры—Шолтена. Для всякой конфигурации γ алгоритма 8.4 определим следующее множество вершин:

$$V_T = \{p: father_p \neq undef\} \cup \{\langle \mathbf{mes}, p \rangle \text{ на этапе пересылки}\} \\ \cup \{\langle \mathbf{sig}, p \rangle \text{ на этапе пересылки}\}$$

и дуг

$$E_F = \{(p, father_p): father_p \neq undef \wedge father_p \neq p\} \\ \cup \{\langle \langle \mathbf{mes}, p \rangle, p \rangle: \langle \mathbf{mes}, p \rangle \text{ на этапе пересылки}\} \\ \cup \{\langle \langle \mathbf{sig}, p \rangle, p \rangle: \langle \mathbf{sig}, p \rangle \text{ на этапе пересылки}\}.$$

Достаточные условия корректности (так называемое свойство безопасности алгоритма) обеспечиваются предпосылкой Q , которая будет определена ниже. Эта предпосылка является инвариантом алгоритма, и обоснование этого факта, конечно, в точности следует схеме доказательства леммы 8.4. Смысл соотношений (1)–(5) предпосылки Q остается тем же самым, что и у инварианта алгоритма Дейкстры—Шолтена, а соотношение (6) выражает свойство корректной оценки процессом того, является ли он корнем одного из деревьев в рассматриваемом

лесе. Лес, несомненно, пуст, если ни один из процессов не является корнем:

$$Q \iff state_p = active \implies p \in V_F \quad (1)$$

$$\wedge (u, v) \in E_F \implies u \in V_F \wedge v \in V_F \cap \mathbb{P} \quad (2)$$

$$\wedge sc_p = \#\{v: (v, p) \in E_F\} \quad (3)$$

$$\wedge V_F \neq \emptyset \implies F \text{ является лесом} \quad (4)$$

$$\wedge (state_p = passive \wedge sc_p = 0) \implies p \notin V_F \quad (5)$$

$$\wedge empty_p \iff T_p \text{ пусто} \quad (6)$$

Лемма 8.6. Предпосылка Q является инвариантом алгоритма Шави—Франчеца.

Доказательство. Первоначально для каждого процесса p , не являющегося инициатором, справедливо равенство $state_p = passive$, а для каждого инициатора p выполняется условие $father_p = p$; отсюда следует соотношение (1). Кроме того, $E_F = \emptyset$, и отсюда следует соотношение (2). Поскольку $sc_p = 0$ для каждого процесса p , верно соотношение (3). Так как $V_F = \{p: p \text{ является инициатором}\}$ и $E_F = \emptyset$, граф F является лесом, образованным из деревьев, состоящих из одной-единственной вершины-инициатора; отсюда следует соотношение (4). Процессы из множества V_F являются инициаторами, и эти процессы активны; значит, верно соотношение (5). Первоначально условие $empty_p$ равносильно утверждению « p не является инициатором», и дерево T_p действительно пусто в том и только том случае, когда процесс p не является инициатором. Это служит обоснованием соотношения (6).

S_p . Ни один процесс не становится активным при выполнении блока S_p , и ни один процесс не удаляется из множества V_F ; значит, соотношение (1) сохраняет справедливость.

Тот факт, что этот блок выполняется, означает, что процесс p , порождающий новую вершину, которая становится его последователем, принадлежит множеству V_F , и поэтому соотношение (2) сохраняется.

В результате выполнения рассматриваемого блока к множеству V_F присоединяется вершина $\langle mes, p \rangle$, а к множеству E_F — дуга $(\langle mes, p \rangle, p)$. Следовательно, граф F по-прежнему остается лесом, а переменная sc_p , увеличивая свое значение, отмечает тем самым появление у вершины p нового последователя. Значит, соотношения (3) и (4) также соблюдаются.

Ни один процесс не становится пассивной листовой вершиной, и ни один новый процесс не включается в V_F ; поэтому соотношение (5) остается верным.

Коль скоро к непустому дереву добавляется новая листовая вершина, никакое дерево не становится пустым, а так как никакая переменная $empty$ не изменяет своего значения, справедливость соотношения (6) сохраняется.

R_p . Процесс p либо уже содержался в множестве V_F (в случае $father_p \neq \text{undef}$), либо присоединяется к этому множеству в результате выполнения рассматриваемого блока. Значит, соотношение (1) остается справедливым.

Если переменной $father_p$ присваивается новое значение, то им является имя процесса q , а если отправляется сигнал, то родительской вершиной для него ста-

новится все тот же процесс q , который содержится в V_T . Поэтому соотношение (2) сохраняется.

Число последователей процесса q не изменяется, поскольку последователь $\langle mes, q \rangle$ вершины q замещается либо вершиной p , либо вершиной $\langle sig, q \rangle$. А это значит, что равенство (3) остается верным.

Структура нашего графа не изменяется, и поэтому соотношение (4) также соблюдается.

Ни один процесс не становится пассивной листовой вершиной, и ни один новый процесс не включается в V_F . Поэтому соотношение (5) остается верным.

Ни одно дерево не исчезает, и никакое новое дерево не появляется. Значит, соотношение (6) сохраняется.

```

var statep : (active, passive)  init if p is initiator then active else passive ;
    scp : integer               init 0 ;
    fatherp :  $\mathbb{P}$                init if p is initiator then p else undef ;
    emptyp : bool              init if p is initiator then false else true ;
Sp: { statep = active }
  begin send  $\langle mes, p \rangle$  ; scp := scp + 1 end
Rp: { Сообщение  $\langle mes, q \rangle$  поступило процессу p }
  begin receive  $\langle mes, q \rangle$  ; statep := active ;
    if fatherp = undef then fatherp := q else send  $\langle sig, q \rangle$  to q
  end
Ip: { statep = active }
  begin statep := passive ;
    if scp = 0 then (* Удалить p из F *)
      begin
        if fatherp = p then emptyp := true else send  $\langle sig, father_p \rangle$  to fatherp ;
        fatherp := undef
      end
    end
  end
Ap: { Сигнал  $\langle sig, p \rangle$  достигает p }
  begin receive  $\langle sig, p \rangle$  ; scp := scp - 1 ;
    if scp = 0 and statep = passive then
      begin
        if fatherp = p then emptyp := true else send  $\langle sig, father_p \rangle$  to fatherp ;
        fatherp := undef
      end
    end
  end

```

Все процессы проводят параллельное выполнение волнового алгоритма, причем отправление сообщений или принятие решения разрешается только тем процессам p , у которых переменная $empty_p$ имеет значение true.

При осуществлении события *decide* вызывается процедура *Announce*.

Алгоритм 8.4. Алгоритм Шави—Франчеца

I_p . Если процесс p становится пассивным, то соотношения (1), (2), (3) и (4) остаются справедливыми. Тот факт, что процесс p ранее был активным, означает,

что он принадлежал множеству V_F . Если $sc_p = 0$, то процесс p будет удален из V_F , и поэтому соотношение (5) остается верным.

Посмотрим теперь, что произойдет, когда процесс p будет исключен из F , т. е. когда он станет пассивной листовой вершиной графа F . Если был отправлен сигнал, то родительской вершиной этого сигнала становится последняя родительская вершина процесса p , которая принадлежит V_F . Следовательно, соотношение (2) остается верным. В этом случае указанный сигнал замещает p в качестве последователя вершины $father_p$. Поэтому значение sc_{father_p} остается правильным, и равенство (3) будет по-прежнему верным. Структура графа не изменяется, и, значит, соотношение (4) также соблюдается. Никакие деревья не становятся пустыми, и поэтому соотношение (6) сохраняется. В противном случае, т. е. в том случае, когда имеет место равенство $father_p = p$, процесс p играл роль корневой вершины, и тот факт, что p оказался листовой вершиной, означает, что p был единственной вершиной дерева T_p . Поэтому при удалении этой вершины дерево T_p становится пустым, и, благодаря тому что оператор присваивания придает переменной $empty_p$ значение true, соотношение (6) сохраняется.

А_р. При получении сигнала число последователей вершины p уменьшается на единицу, и соответствующий оператор присваивания обеспечивает выполнимость равенства (3). Тот факт, что процесс p был родительской вершиной этого сигнала, означает, что он принадлежал множеству V_F . Если выполнялось равенство $state_p = passive$ и после получения сигнала обнаружилось, что $sc_p = 0$, то вершина p удаляется из графа. Значит, соотношение (5) по-прежнему соблюдается.

Если вершина p была удалена из множества V_F , то, повторив те же рассуждения, которые применялись при анализе блока **I_р**, можно убедиться в том, что инвариант по-прежнему сохраняется. $\square$

Теорема 8.7. *Алгоритм Шави—Франчеца (алгоритм 8.4) является корректным алгоритмом обнаружения завершения вычисления; в нем используется $M + W$ обменов контрольными сообщениями.*

Доказательство. Как при доказательстве теоремы 8.5, можно показать, что число сигналов не превосходит числа базовых сообщений. Кроме сигналов контрольный алгоритм отправляет только сообщения для одной волны. Значит, всего будет отправлено не более $M + W$ контрольных сообщений.

Чтобы обосновать необходимое условие корректности алгоритма, предположим, что базовое вычисление завершилось. Спустя конечное число шагов рассматриваемый алгоритм обнаружения завершения вычисления достигает заключительной конфигурации, и точно так же, как в теореме 8.5, можно показать, что в этой конфигурации граф F пуст. Следовательно, в распространении волны может участвовать каждый процесс. И коль скоро была достигнута заключительная конфигурация, все события, присущие этой волне, осуществились, включая хотя бы одно событие решения, которое было обязано вызвать процедуру *Announce*.

Чтобы обосновать достаточное условие корректности алгоритма, напомним, что процедура *Announce* вызывается только тогда, когда в волновом алгоритме принимается решение. Это означает, что каждый процесс p уже отправил сооб-

щение по волне или принял решение. А рассматриваемый нами алгоритм устроен так, что процесс p совершает это только тогда, когда $empty_p$ имеет значение true. Никакое действие не может вновь придать переменной $empty_p$ значение false, и поэтому, когда вызывается процедура *Announce*, для каждого процесса p переменная $empty_p$ имеет значение true. Тогда согласно соотношению (6) множество V_F пусто, и отсюда следует, что предикат **term** выполнен. $\square$

Число обменов контрольными сообщениями в алгоритме Шави—Франчеца — это величина такого же порядка, что и нижняя оценка, установленная в теоремах 8.2 и 8.3. Этот алгоритм является оптимальным по сложности в наихудшем случае алгоритмом обнаружения завершения децентрализованных вычислений (при условии, что он снабжен оптимальным волновым алгоритмом).

Для применения всех алгоритмов, рассмотренных в этом параграфе, требуется, чтобы каналы связи были двунаправленными; для каждого сообщения, отправленного от p к q , должен быть отправлен сигнал от q к p . Сложность в среднем будет такой же, как и сложность в наихудшем случае: для каждого выполнения алгоритма на одно базовое сообщение приходится одно сигнальное сообщение, а в случае алгоритма Шави—Франчеца, еще и проход ровно одной волны.

8.3. Решения на основе волновых алгоритмов

Есть две причины, по которым полезно рассмотреть некоторые другие алгоритмы обнаружения завершения вычисления, помимо тех двух алгоритмов, которые были приведены в §8.5.2. Во-первых, рассмотренные ранее алгоритмы можно применять только в том случае, когда каналы являются двунаправленными. А во-вторых, хотя эти алгоритмы имеют оптимальную сложность по числу сообщений *в наихудшем случае*, существуют алгоритмы, имеющие лучшую сложность в среднем. Все алгоритмы из предыдущего параграфа используют при каждом выполнении такое же число сообщений, что и в наихудшем случае.

В этом параграфе мы изучим некоторые алгоритмы, в основу которых положен принцип многократного выполнения волнового алгоритма; в конце каждой волны либо обнаруживается завершение вычисления, либо запускается новая волна. Завершение вычисления обнаруживается, если в каждом процессе будут выполнены некоторое локальное условие.

Начнем с того, что рассмотрим конкретные примеры таких алгоритмов, в которых в качестве волнового алгоритма выступает кольцевой алгоритм. Предполагается, что топология нашей сети допускает выделение некоторого кольца; однако обмен базовыми сообщениями не ограничивается каналами, входящими в состав кольца. Процессы занумерованы, нумерация ведется от p_0 до p_{N-1} , и кольцевой алгоритм иницируется процессом p_0 путем отправления маркера процессу p_{N-1} . Процесс p_{i+1} (для $i < N-1$) переправляет этот маркер процессу p_i . Обход маркером кольца завершается, как только он будет возвращен обратно процессу p_0 .

Первое решение такого рода, которое мы обсудим, — это алгоритм Дейкстры—Фейджена—ван Гастерена (§8.3.1). Этот алгоритм обнаруживает заверше-

ние вычислений с синхронным обменом сообщениями. Сразу нескольким авторам удалось обобщить этот алгоритм для случая вычислений с асинхронным обменом сообщениями; главная проблема здесь состоит в проверке того, что каналы связи пусты. Мы обсудим решение, предложенное Сафрой (см. §8.3.2), в котором каждый процесс проводит подсчет числа отправленных и полученных сообщений. Сравнивая показания счетчиков, можно на самом деле убедиться в том, что каналы пусты. Для этой цели можно также воспользоваться подтверждениями (§8.3.3); но это решение вновь требует, чтобы каналы были двунаправленными, и число контрольных сообщений будет по меньшей мере равно тому количеству сообщений, которое используется алгоритмом Шави—Франчеца.

В §8.3.4 указанный принцип обнаружения завершения вычислений будет обобщен для случая произвольного волнового алгоритма.

8.3.1. Алгоритм Дейкстры—Фейджена—ван Гастерена

```

var statep : (active, passive);
Cpq: { statep = active }
  begin (* процесс p отправляет базовое сообщение,
        и оно сразу же принимается процессом q *)
    stateq := active
  end
Ip: { statep = active }
  begin statep := passive end

```

Алгоритм 8.5. Базовый алгоритм с синхронным обменом сообщениями

Алгоритм Дейкстры, Фейджена и ван Гастерена [66] обнаруживает завершение базового вычисления, в котором используется *синхронный* обмен сообщениями; действия подобного вычисления приведены на примере алгоритма 8.5. Для таких вычислений условие их завершения имеет следующий вид:

$$\mathbf{term} \iff \forall p : \text{state}_p = \text{passive}.$$

Можно сравнить приведенный алгоритм и условие **term** с алгоритмом 8.1 и теоремой 8.1.

Построение алгоритма состоит из ряда небольших шагов. Каждый шаг можно легко понять, а корректность алгоритма является следствием одного инварианта, который мы построим вместе с самим алгоритмом. Такой подход заимствован из статьи [66]. В дальнейшем символом t условимся обозначать номер того процесса, который удерживает маркер или должен получить маркер, если этот маркер пребывает на этапе пересылки. Только процесс p_t может осуществить передачу маркера, и при этом t уменьшится на 1. Волна угасает, когда $t = 0$, и, следовательно, инвариант P должен быть выбран так, чтобы о завершении вычисления можно было судить на основе самого P , того факта, что $t = 0$, и дополнительной

информации, представленной в процессе p_0 . Инвариантное соотношение должно соблюдаться, когда p_0 запускает волну, т. е. когда $t = N - 1$.

В качестве первого приближения будем полагать, что $P = P_0$, где P_0 задается следующим соотношением:

$$P_0 \equiv \forall i (N > i > t) : \text{state}_{p_i} = \text{passive}.$$

И в самом деле, P_0 обращается в истину, когда $t = N - 1$, и если выполняются равенства $t = 0$ и $\text{state}_{p_0} = \text{passive}$, то отсюда мы заключаем, что вычисление завершилось. При передаче маркера P_0 сохраняет справедливость только тогда, когда в передаче участвуют одни лишь пассивные процессы; значит, нам приходится ввести следующее правило.

Правило 1. Только пассивный процесс обладает маркером.

При таком режиме работы условие P сохраняется при передаче маркера, а также при выполнении внутренних действий; к сожалению, P не сохраняется при выполнении коммуникационных действий. Предикат P_0 может быть опровергнут, если процесс p_j активизируется таким процессом p_i , что $j > t$ и $i \leq t$ (см. упражнение 8.4). Так как P_0 можно опровергнуть, в качестве P выбирается менее жесткое утверждение $(P_0 \vee P_1)$, причем P_1 выбирается так, чтобы в случае каждого опровержения предиката P_0 предикат P_1 обращался в истину. Мы будем полагать, что каждый процесс может иметь некоторый цвет, который задается переменной *color*; эта переменная может иметь одно из двух значений *white* или *black*. Положим $P = (P_0 \vee P_1)$, где предикат P_1 определяется соотношением

$$P_1 \equiv \exists j (t \geq j \geq 0) : \text{color}_{p_j} = \text{black}.$$

Всякий раз, когда предикат P_0 опровергается, предикат P_1 либо уже выполняется, либо должен обратиться в истину, если процесс-отправитель окрашен в цвет *black*.

Правило 2. Всякий процесс-отправитель имеет цвет *black*.

Коль скоро имеет место соотношение $(P \wedge \text{color}_{p_0} = \text{white} \wedge t = 0) \implies \neg P_1$, мы можем обнаружить завершение вычисления и при помощи нового инварианта, а именно оценив окрашен ли процесс p_0 в цвет *white* (будучи при этом пассивным), когда он владеет маркером.

Ослабление условия P успешно предотвращает его опровержение при выполнении коммуникационных действий; но и ослабленное утверждение можно опровергнуть при передаче маркера, а именно в том случае, когда процесс p_t является единственным процессом, окрашенным в цвет *black*, и при этом осуществляет передачу маркера. Чтобы спасти ситуацию, требуется дальнейшее ослабление условия P . Будем предполагать, что маркер тоже окрашен (в один из двух цветов *white* или *black*), а P имеет вид $(P_0 \vee P_1 \vee P_2)$, где предикат P_1 определяется соотношением

$$P_2 \equiv \text{маркер окрашен в цвет } \textit{black}.$$

Предикат P_2 сохраняется при передаче маркера, если процессы, окрашенные в цвет *black*, передают маркер того же цвета.

Правило 3. Если процесс, окрашенный в цвет *black* и отличный от p_0 , передает маркер, то маркер также приобретает окраску *black*.

Так как верно соотношение (маркер окрашен в цвет *white*) $\implies \neg P_2$, завершение вычисления может быть обнаружено процессом p_0 , а именно путем проверки того, получил ли этот процесс маркер цвета *white* (будучи при этом пассивным и окрашенным в цвет *white*).

И в самом деле, теперь можно убедиться в том, что внутренние действия, обмен базовыми сообщениями и передача маркера не нарушают истинности предиката P . Но за счет окрашивания маркера в цвет *black* возникает эффект *неудачных волн*; процесс p_0 не способен выявить завершение вычисления, если полученный им маркер имел цвет *black*. Если одна волна оканчивается неудачно, то должна быть запущена новая волна.

Правило 4. Когда одна волна оканчивается неудачно, процесс p_0 запускает новую волну.

Конечно, и следующая волна так же завершится неудачей, как и ее предшественница, если процесс, окрашенный в цвет *black*, не будет иметь возможности вновь приобрести окраску *white*. Действительно, процессы, окрашенные в цвет *black*, придадут такой же цвет маркеру, когда будут передавать его, и поэтому очередная волна будет неудачной.

```

var statep : (active, passive) ;
    colorp : (white, black) ;
Cpq: { statep = active }
    begin (* процесс p отправляет базовое сообщение,
           которое получает процесс q *)
        colorp := black ;    (* Правило 2 *)
        stateq := active
    end

Ip: { statep = active }
    begin statep := passive end
Приступить к обнаружению завершения вычисления,
выполняется один раз процессом p0:
    begin send ⟨tok, white⟩ to pN-1 end

Tp: (* Процесс p обладает маркером ⟨tok, c⟩ *)
    { statep = passive }    (* Правило 1 *)
    begin if p = p0
        then if (c = white ∧ colorp = white)
            then Announce
            else send ⟨tok, white⟩ to pN-1    (* Правило 4 *)
        else if (colorp = white)    (* Правило 3 *)
            then send ⟨tok, c⟩ to Nextp
            else send ⟨tok, black⟩ to Nextp ;
            colorp := white    (* Правило 5 *)
        end
    end

```

Алгоритм 8.6. Алгоритм Дейкстры—Фейджена—ван Гастерена

Заметим, что процесс p_i , обретший окраску *white*, не опровергает P , если $i > t$, а также что условие P всегда будет выполнено, когда p_0 запускает волну, передавая маркер процессу p_{N-1} . Отсюда следует, что процесс можно всегда безопасно перекрасить в цвет *white* после передачи маркера.

Правило 5. Каждый процесс приобретает цвет *white* сразу после передачи маркера.

Этого режима установления цвета *white* вполне достаточно, чтобы обеспечить успех одной из волн после завершения базового вычисления. Приведенные рассуждения представлены в виде алгоритма 8.6.

Теорема 8.8. Алгоритм Дейкстры—Фейджена—ван Гастерена (алгоритм 8.6) является корректным алгоритмом обнаружения завершения базовых вычислений с синхронным обменом сообщениями.

Доказательство. Предикат P определяется формулой $P_0 \vee P_1 \vee P_2$, а сам алгоритм устроен так, что P является инвариантом этого алгоритма. Завершение базового вычисления обнаруживается, когда пассивный процесс p_0 , окрашенный в цвет *white*, владеет маркером, который также окрашен в цвет *white*. И действительно, когда это происходит, окраска маркера означает, что выполняется условие $\neg P_2$, окраска процесса p_0 и тот факт, что $t = 0$, означают, что выполняется условие $\neg P_1$, а выполнимость условия P_0 и то состояние, в котором пребывает процесс p_0 , приводят к заключению о выполнимости предиката **term**. Таким образом, достаточное условие корректности обеспечено.

Чтобы обосновать необходимое условие корректности, предположим, что базовое вычисление завершилось. Начиная с этого момента, все процессы по получении маркера передают его без задержки. Когда маркер завершает полный обход кольца в первый раз после завершения базового вычисления, все процессы будут иметь окраску *white*, а когда маркер обойдет кольцо вторично, будет обнаружено, что базовое вычисление завершилось. $\square$

Теперь мы попытаемся оценить число контрольных сообщений, используемых рассматриваемым алгоритмом. Базовое вычисление, фигурирующее в доказательстве теоремы 8.2, вынуждает маркер совершать по меньшей мере один обход кольца для каждой пары базовых сообщений; значит, в худшем случае алгоритму потребуется $\frac{1}{2}NM$ контрольных сообщений (см. упражнение 8.5).

Но данный алгоритм может использовать гораздо меньше сообщений в среднем, когда анализируется «произвольное» вычисление. Предположим, что сложность базового вычисления по времени равна T . Поскольку маркер всегда передается последовательно, вполне разумно полагать, что он будет передан приблизительно T раз, прежде чем базовое вычисление завершится. (И даже эта оценка может оказаться чрезмерно завышенной, так как только активные процессы откладывают передачу маркера.) Поскольку маркер передается менее $3N$ раз после завершения базового вычисления, нашему алгоритму требуется в таком случае совершить менее $T + 3N$ обменов контрольными сообщениями. И хотя сложность базового вычисления равна T (речь идет о сложности по времени), но если вычисление обладает достаточным параллелизмом, то его сложность по числу

сообщений может оценена величиной $\Omega(NT)$. Если параллелизм позволяет каждому процессу отправлять некоторое постоянное число α сообщений за единицу времени, сложность базового вычисления по числу сообщений может достигать величины $NT\alpha$, т. е. $\Omega(NT)$. Таким образом, количество контрольных сообщений, оцениваемое величиной $O(N+T)$, оказывается тогда гораздо меньшим, чем можно было бы ожидать исходя из сложности задачи обнаружения завершения вычисления в наихудшем случае.

8.3.2. Подсчет базовых сообщений: алгоритм Сафры

Синхронный обмен сообщениями, присущий базовому вычислению алгоритма Дейкстры—Фейджена—ван Гастерена, существенно ограничивает общее применение этого алгоритма. Ряд авторов обобщили указанный алгоритм на случай вычислений с асинхронным обменом сообщениями (см., например, алгоритм 8.1). В данном параграфе мы обсудим решение Сафры (см. [63]); оно обладает тем свойством, что его сложность в среднем сравнима со сложностью алгоритма Дейкстры—Фейджена—ван Гастерена.

Для каждой конфигурации обозначим символом B число сообщений, находящихся на этапе пересылки. Тогда предикат **term** эквивалентен следующей формуле:

$$(\forall p: state_p = passive) \wedge B = 0.$$

Разработаем вновь инвариант P таким образом, чтобы заключение о завершении вычисления можно было вынести на основе P , равенства $t = 0$ и некоторой информации, предоставленной процессом p_0 . Инвариант должен соблюдаться, когда p_0 запускает волну, т. е. при $t = N - 1$.

Чтобы сделать информацию о значении B доступной процессам (хотя бы в распределенном виде), снабдим процесс p счетчиком сообщений mc_p так, чтобы процессы придерживались как инварианта условия P_m следующего вида:

$$P_m \equiv B = \sum_{p \in \mathbb{P}} mc_p.$$

Инвариантность условия P_m будет соблюдена, если первоначально полагать $mc_p = 0$ для каждого процесса p и потребовать, чтобы все процессы подчинялись следующему правилу.

Правило М. Когда процесс p отправляет сообщение, он увеличивает значение своего счетчика; когда процесс p получает сообщение, он уменьшает значение своего счетчика.

Такой инвариант должен предоставлять процессу p_0 возможность принять решение о выполнимости условия **term**, после того как он получит маркер ($t = 0$). Так как **term** теперь включает ограничение, налагаемое на значение B , мы будем использовать маркер для передачи целого числа q , чтобы суммировать показания счетчиков сообщений всех тех процессов, в которых побывал этот маркер. В качестве первого приближения положим $P = P_m \wedge P_0$, где предикат P_0

задается следующим образом:

$$P_0 \equiv (\forall i (N > i > t): state_{p_i} = passive) \wedge \left(q = \sum_{N > i > t} mc_{p_i} \right).$$

И действительно, P_0 обращается в истину, когда $t = N - 1$ и $q = 0$, причем в случае $t = 0$ из условия P следует утверждение

$$(\forall i > 0: state_{p_i} = passive) \wedge (mc_{p_0} + q = B).$$

Поэтому p_0 может обнаружить завершение базового вычисления, если будут выполнены равенства $state_{p_0} = passive$ и $mc_{p_0} + q = 0$.

Утверждение P_0 выполняется, когда процесс p_0 запускает волну, передавая маркер процессу p_{N-1} с пометкой $q = 0$. При передаче маркера утверждение P_0 будет оставаться верным в том случае, когда в передаче маркера участвуют только пассивные процессы, и при этом к пометке маркера добавляется показание счетчика сообщений. Таким образом, мы вводим следующее правило.

Правило 1. Только пассивный процесс может работать с маркером, и если процесс передает маркер, то он добавляет к пометке маркера q значение своего счетчика сообщений.

При таком режиме работы условие P сохраняется как при передаче маркера, так и при выполнении внутренних действий. К сожалению, P не сохраняется при получении сообщения таким процессом p_i , для которого $i > t$. Опровержение предиката P_0 происходит, как только будет получено некоторое сообщение, т. е. это возможно только при условии $B > 0$. Поскольку до получения этого сообщения утверждение P_0 было верно, это означает, что в таком случае должно выполняться условие P_1 , которое задается следующим соотношением:

$$P_1 \equiv \left(\sum_{i \leq t} mc_{p_i} + q \right) > 0.$$

Утверждение P_1 остается справедливым и после получения сообщения процессом p_i , номер i которого удовлетворяет неравенству $i > t$; следовательно, более слабое утверждение P , которое определяется формулой $P_m \wedge (P_0 \vee P_1)$, сохраняет свою справедливость и в случае получения сообщения процессом p_i , $i > t$.

Уточненное таким образом условие P тем не менее позволяет процессу p_0 обнаружить завершение базового вычисления по тем же самым признакам: при $t = 0$ утверждение P_1 верно лишь при $mc_{p_0} + q > 0$, а это означает, с учетом равенства $mc_{p_0} + q = 0$ (проверку которого требуется провести для обнаружения завершения вычисления), что имеет место $\neg P_1$. Передача маркера, равно как и отправление базового сообщения, сохраняет предикат P_1 . К сожалению, условие P_1 может быть опровергнуто при получении некоторого сообщения процессом p_i , номер i которого удовлетворяет неравенству $i \leq t$. Как и в §8.3.1, такую ситуацию можно исправить, снабдив каждый процесс окраской и введя следующее правило.

Правило 2. Процесс-получатель окрашивается в цвет *black*.

```

var  $state_p$  : (active, passive) ;
       $color_p$  : (white, black) ;
       $mc_p$  : integer init 0 ;
 $S_p$ : {  $state_p = active$  }
  begin send  $\langle mes \rangle$  ;
       $mc_p := mc_p + 1$  (* Rule M *)
  end
 $R_p$ : { Сообщение  $\langle mes \rangle$  доставлено процессу  $p$  }
  begin receive  $\langle mes \rangle$  ;  $state_p := active$  ;
       $mc_p := mc_p - 1$  ; (* Правило M *)
       $color_p := black$  (* Правило 2 *)
  end

 $I_p$ : {  $state_p = active$  }
  begin  $state_p := passive$  end
Приступить к обнаружению завершения вычисления,
выполняется один раз процессом  $p_0$ :
  begin send  $\langle tok, white, 0 \rangle$  to  $p_{N-1}$  end
 $T_p$ : (* Процесс  $p$  обрабатывает маркер  $\langle tok, c, q \rangle$  *)
  {  $state_p = passive$  } (* Правило 1 *)
  begin if  $p = p_0$ 
    then if  $(c = white) \wedge (color_p = white) \wedge (mc_p + q = 0)$ 
      then Announce
      else send  $\langle tok, white, 0 \rangle$  to  $p_{N-1}$  (* Правило 4 *)
    else if  $(color_p = white)$  (* Правила 1 и 3 *)
      then send  $\langle tok, c, q + mc_p \rangle$  to  $Next_p$ 
      else send  $\langle tok, black, q + mc_p \rangle$  to  $Next_p$ ;
       $color_p := white$  (* Правило 5 *)
    end
  end

```

Алгоритм 8.7. Алгоритм Сафры

После этого заменим прежнее условие P на новое, которое будет представлено формулой $P_m \wedge (P_0 \vee P_1 \vee P_2)$, где

$$P_2 \equiv \exists j \ (t \geq j \geq 0) : color_{p_j} = black.$$

Каждое получение сообщения, влекущее за собой опровержение утверждения P_1 , делает истинным P_2 , и поэтому ни одно базовое действие не приводит к опровержению утверждения P . Поскольку верна импликация $(P \wedge color_{p_0} = white \wedge t = 0) \implies \neg P_2$, новое утверждение оставляет возможность обнаружить завершение вычисления. Это можно сделать, проверив, окрашен ли процесс p_0 в цвет *white* (будучи при этом пассивным), когда он обладает маркером.

Ослабленный вариант утверждения P нельзя опровергнуть в результате выполнения базовых действий; но и его можно опровергнуть, если принять во внимание передачу маркера, например, когда процесс p_t , передающий маркер, — это единственный процесс цвета *black*. Спасти ситуацию может дальнейшее ослабление утверждения P . Мы вновь будем полагать, что маркер также имеет окраску

(*white* или *black*), а само утверждение P ослабим, приведя его к виду $P_m \wedge (P_0 \vee P_1 \vee P_2 \vee P_3)$ где

$$P_3 \equiv \text{the token is black.}$$

При передаче маркера предикат P_3 остается истинным, если процессы цвета *black* передают маркер, также имеющий окраску *black*.

Правило 3. Когда процесс цвета *black* передает маркер, маркер приобретает окраску *black*.

Поскольку верна импликация «маркер окрашен в цвет *white*» $\implies \neg P_3$, процесс p_0 тем не менее способен обнаружить завершение вычисления. Для этого нужно проверить, верно ли, что этот процесс получил маркер цвета *white* (будучи при этом пассивным и окрашенным в цвет *white*).

Действительно, теперь можно убедиться, что утверждение P сохраняет справедливость при выполнении внутренних действий, обмене базовыми сообщениями и при передаче маркера. Прохождение волны не приводит к успеху, если при возвращении маркера процессу p_0 оказывается, что этот маркер окрашен в цвет *black*, p_0 окрашен в цвет *black* или $mc_{p_0} + q \neq 0$. Если одна волна была неудачной, то нужно запустить новую.

Правило 4. Когда прохождение волны заканчивается неудачно, процесс p_0 запускает новую волну.

Очередная волна, впрочем, может оказаться столь же неудачной, как и ее предшественница, если процесс цвета *black* не будет иметь возможности сменить свою окраску на *white*. И в самом деле, процессы, окрашенные в цвет *black*, при передаче маркера придают ему ту же окраску, вследствие чего новая волна также потерпит неудачу.

Обратим внимание на то, что, переименовав окраску на *white*, процесс p_i не опровергнет тем самым утверждение P , если $i > t$, а само утверждение P всегда остается истинным, когда p_0 запускает волну, передавая маркер процессу p_{N-1} . Отсюда следует, что процессы при передаче маркера могут менять окраску с *black* на *white*, не нарушая истинности утверждения P .

Правило 5. Сразу после передачи маркера всякий процесс окрашивается в цвет *white*.

При таком режиме перемены окраски можно гарантировать, что после завершения базового вычисления какая-нибудь волна рано или поздно окажется успешной. Приведенные здесь рассуждения воплощены в алгоритме 8.7.

Теорема 8.9. Алгоритм Сафры (алгоритм 8.7) является корректным алгоритмом обнаружения завершения вычислений с асинхронным обменом сообщениями.

Доказательство. Рассматриваемый нами алгоритм устроен так, что его инвариантом является предикат P , который определяется формулой $P_m \wedge (P_0 \vee P_1 \vee P_2 \vee P_3)$.

Чтобы обосновать достаточное условие корректности, следует заметить, что алгоритм обнаруживает завершение вычисления только тогда, когда выполняются равенства $t = 0$, $state_{p_0} = passive$, $color_{p_0} = white$ и $mc_{p_0} + q = 0$. Из этих равенств следует выполнимость условий $\neg P_3$, $\neg P_2$ и $\neg P_1$, а это, в свою очередь,

означает, что выполняется условие $P_m \wedge P_0$. С учетом того, что $state_{p_0} = passive$ и $mc_{p_0} + q = 0$, последнее условие влечет за собой выполнимость предиката **term**.

Чтобы обосновать необходимое условие корректности, заметим, что после завершения базового вычисления счетчики сообщений будут иметь постоянные значения и сумма их будет равна 0. Волна, запущенная из такой конфигурации, завершится так, что будет соблюдено равенство $mc_{p_0} + q = 0$ и при этом все процессы будут окрашены в цвет *white*. Следующая за ней волна непременно будет успешной. $\square$

В отличие от алгоритма Дейкстры—Фейджена—ван Гастерена, сложность в наихудшем случае алгоритма Сафры не ограничена сверху; маркер может передаваться неограниченное число раз, пока все процессы будут пассивны, но какое-то сообщение будет долгое время пребывать на этапе пересылки. Так же как и в случае алгоритма Дейкстры—Фейджена—ван Гастерена, математическое ожидание сложности по числу сообщений составляет величину $\Theta(T + N)$ для базового вычисления, имеющего сложность по времени, равную T .

Алгоритм векторного подсчета. Маттерн в работе [141] предложил алгоритм, сравнимый по эффективности с алгоритмом Сафры, но требующий при этом отдельного счетчика сообщений для каждого процесса-адресата. В процессе p в качестве счетчика сообщений используется массив $mc_p[\mathbb{P}]$. Когда p отправляет сообщение процессу q , содержимое счетчика процесса p изменяется при помощи оператора $mc_p[q] := mc_p[q] + 1$, а когда p получает сообщение, содержимое счетчика процесса p изменяется при помощи оператора $mc_p[p] := mc_p[p] - 1$. Вместе с маркером передается также и массив счетчиков, и, когда процесс p владеет маркером, содержимое массива mc_p передается с маркером, а сам счетчик сбрасывается в $\vec{0}$ (массив, все компоненты которого равны 0). Завершение вычисления регистрируется, когда массив, переносимый маркером, становится равным $\vec{0}$.

Алгоритм векторного подсчета обладает некоторыми преимуществами по сравнению с алгоритмом Сафры, но у него есть и серьезные изъяны. Одно из достоинств этого алгоритма состоит в том, что завершение вычисления обнаруживается быстрее, а именно спустя один тур после того, как завершится базовое вычисление. Другое положительное качество данного алгоритма состоит в том, что, пока вычисление не завершилось, продвижение маркера задерживается более часто, и это позволяет сократить число обменов контрольными сообщениями в ходе работы алгоритма. В алгоритме Сафры каждый пассивный процесс передает маркер; а вот в алгоритме векторного подсчета такой процесс не будет проводить передачу маркера, если содержащаяся в маркере информация позволяет заключить, что какое-то сообщение находится на этапе пересылки и будет доставлено процессу p .

Главный недостаток алгоритма векторного подсчета состоит в том, что вместе с маркером передается большой объем информации (если быть точным, то по одному целому числу для каждого из процессов), и все это передается в каждом контрольном сообщении. Этот недостаток может быть не столь серьезным, если число процессов невелико. Другой изъян состоит в том, что для этого ал-

горитма требуется знание отличительных признаков всех остальных процессов. Если вектор представляется массивом, то каждый процесс должен быть заранее осведомлен о всем многообразии отличительных признаков процессов системы; но это требование может быть ослаблено, если вектор представляется в виде множества пар, каждая из которых состоит из отличительного признака и целого числа. Первоначально каждый процесс должен знать отличительные признаки соседних процессов (чтобы правильно увеличивать показания счетчика), а все прочие отличительные признаки будут изучены по ходу вычисления.

```

var  $state_p$  : (active, passive) ;
       $color_p$  : (white, black) ;
       $unack_p$  : integer init 0 ;
 $S_p$ : {  $state_p = active$  }
      begin send  $\langle mes \rangle$  ;  $unack_p := unack_p + 1$  ; (* Правило А *)
               $color_p := black$  (* Правило В *)
      end  $R_p$ ; Сообщение  $\langle mes \rangle$  от  $q$  доставлено процессу  $p$ 
      begin receive  $\langle mes \rangle$  ;  $state_p := active$  ;
              send  $\langle ack \rangle$  to  $q$  (* Правило А *)
      end
 $I_p$ : {  $state_p = active$  }
      begin  $state_p := passive$  end
 $A_p$ : {Подтверждение  $\langle ack \rangle$  доставлено процессу  $p$  }
      begin receive  $\langle ack \rangle$  ;  $unack_p := unack_p - 1$  end (* Правило А *)
Приступить к обнаружению завершения вычисления;
исполняется один раз процессом  $p_0$ :
      begin send  $\langle tok, white \rangle$  to  $p_{N-1}$  end
 $T_p$ : (* Процесс  $p$  владеет маркером  $\langle tok, c \rangle$  *)
      {  $state_p = passive \wedge unack_p = 0$  }
      begin if  $p = p_0$ 
              then if  $c = white \wedge color_p = white$ 
                      then Announce
                      else send  $\langle tok, white \rangle$  to  $p_{N-1}$ 
              else if ( $color_p = white$ )
                      then send  $\langle tok, c \rangle$  to  $Next_p$ 
                      else send  $\langle tok, black \rangle$  to  $Next_p$  ;
                       $color_p := white$  (* Правило В *)
              end
      end

```

Алгоритм 8.8. Обнаружение завершения вычисления с использованием подтверждений

8.3.3. Использование подтверждений

В алгоритме Сафры проводится подсчет отправленных и полученных базовых сообщений, чтобы установить, что никакие базовые сообщения не пребывают на этапе пересылки. Это требование можно обеспечить, используя подтверждения; сразу несколько авторов предложили снабдить этим механизмом алгоритм

Дейкстры—Фейджена—ван Гастерена (см., например, работу [152]). Подобную модификацию основного метода мы обсудим лишь вкратце, поскольку полученный в результате алгоритм ни в коей мере не улучшает алгоритм Шави—Франчеца и поэтому уже вышел из употребления.

Прежде всего заметим, что утверждение «ни одно сообщение не пребывает на этапе пересылки» равносильно утверждению «для каждого процесса p ни одно сообщение, отправленное p , не пребывает на этапе пересылки». На каждый процесс возлагается ответственность за отправленные им сообщения, т.е. процесс должен предотвратить вызов процедуры *Announce*, до тех пор пока он не удостоверится, что все отправленные им базовые сообщения достигли адресата. В рассматриваемом методе обнаружения завершения вычисления для каждого процесса p локальное условие $quiet(p)$ определяется так, чтобы выполнялась импликация

$$quiet(p) \implies (state_p = passive \wedge \text{ни одно сообщение, отправленное процессом } p, \text{ не пребывает на этапе пересылки}).$$

Тогда будет верно, что $(\forall p : quiet(p)) \implies \text{term}$.

Чтобы установить, что ни одно сообщение, отправленное процессом p , не пребывает на этапе пересылки, каждое сообщение сопровождается подтверждением и каждый процесс ведет учет числа подтверждений, которые должны быть им получены. Точнее говоря, предикат P_a , который определяется соотношением

$$\forall p : (unpack_p = \#(\text{сообщения на этапе пересылки, отправленные процессом } p) + \#(\text{подтверждения на этапе пересылки, адресованные } p)).$$

должен быть инвариантом, что обеспечивается следующим правилом.

Правило А. При отправлении сообщения процесс p увеличивает значение $unpack_p$; при получении сообщения от процесса q , процесс p отправляет процессу q подтверждение; при получении подтверждения процесс p уменьшает значение $unpack_p$.

Указанное выше требование для условия $quiet$ (при выполнении условия $quiet(p)$ процесс p является пассивным, и ни одно сообщение, отправленное процессом p , не пребывает на этапе пересылки) будет теперь соблюдено, если задать условие $quiet$ следующим соотношением

$$quiet(p) \equiv (state_p = passive \wedge unpack_p = 0).$$

Построение нашего алгоритма обнаружения завершения вычисления проводится поэтапно, подобно тому как был построен алгоритм Дейкстры—Фейджена—ван Гастерена. Вначале рассматривается предикат P_0 , который задается соотношением

$$P_0 \equiv \forall i \ (N > i > t) : quiet(p).$$

Предикат P_1 нужно вводить осмотнительно, поскольку активизация процесса p_j процессом p_i в случае, когда $j > t$ и $i \leq t$, не происходит совместно с событием отправления сообщения процессом p_i . Однако если процесс p_j активизирован

(вследствие чего предикат P_0 будет ложным), то соотношением

$$P_b \equiv \forall p : (unpack_p > 0 \implies color_p = black),$$

при помощи правила В, мы приходим

$$P_b \equiv \forall p : (unpack_p > 0 \implies color_p = black),$$

при помощи правила В, мы приходим к заключению о том, что, как только предикат P_0 становится ложным, будет выполнено условие P_1 , и поэтому утверждение $(P_0 \vee P_1)$ не будет опровергнуто.

Правило В. После отправления сообщения всякий процесс окрашивается в цвет *black*; всякий процесс имеет окраску *white* только когда для него выполняется условие $quiet$.

В результате мы получим алгоритм 8.8, инвариант которого задается формулой $P_a \wedge P_b \wedge (P_0 \vee P_1 \vee P_2)$, где

$$P_a \equiv \forall p : (unpack_p = \#(\text{сообщения, отправленные процессом } p) + \#(\text{подтверждения, адресованные } p))$$

$$P_b \equiv \forall p : (unpack_p > 0 \implies color_p = black)$$

$$P_0 \equiv \forall i \ (N > i > t) : quiet(p)$$

$$P_1 \equiv \exists i \ (t \geq i \geq 0) : color_{p_i} = black$$

$$P_2 \equiv \text{маркер окрашен в цвет } black.$$

Теорема 8.10. Алгоритм 8.8 является корректным алгоритмом обнаружения завершения вычислений с асинхронным обменом сообщениями.

Доказательство. Завершение вычисления объявляется в том случае, когда процесс p_0 удовлетворяет условию $quiet$ и владеет маркером, окрашенным в цвет *white*. Отсюда следует, что условия P_2 и P_1 не выполняются, и, значит, верна формула $P_a \wedge P_b \wedge P_0$. Из этой формулы можно заключить, принимая во внимание условие $quiet(p_0)$, что все процессы также удовлетворяют условию $quiet$, и, следовательно, предикат **term** истинен.

Когда завершается базовое вычисление, спустя некоторое время будут получены все подтверждения и все процессы будут удовлетворять условию $quiet$. Первая же волна, запущенная после того, как условие $quiet$ будет выполнено для всех процессов, завершится, окрасив все процессы в цвет *white*, и завершение базового вычисления будет объявлено по окончании следующей волны. $\square$

Решение, основанное на ограниченной задержке передачи сообщений. В [187, § 4.1.3] описан один подход к решению задачи обнаружения завершения вычисления (а также целого ряда других задач), в основу которого положено допущение о том, что задержка передачи сообщений ограничена некоторой константой μ (см. также § 3.3.2). В таком случае всякий процесс остается возбужденным на протяжении интервала времени, длительность которого равна μ , после отправления самого последнего сообщения, и при этом процесс остается окрашенным в цвет *black*, до тех пор пока он возбужден, подобно тому как это было описано выше для решения задачи с использованием подтверждений.

Процесс p перестает быть возбужденным, если (1) самое последнее отправленное сообщение этот процесс осуществил не менее μ единиц времени тому назад, и (2) этот процесс является пассивным. Мы предоставляем читателю самостоятельно провести полное формальное построение алгоритма.

8.3.4. Обнаружение завершения вычислений при помощи волн

До сих пор в этом параграфе мы обсуждали алгоритмы обнаружения завершения вычислений, в которых для передачи контрольных сообщений использовались кольцевые структуры; в основу этих алгоритмов были положены волновые алгоритмы для колец. Аналогичные решения были предложены и для других топологий. Так, например, Франчез и Роде в работе [88], а также Топор в работе [193] разработали алгоритм, в котором для передачи контрольных сообщений используется остовное дерево. Тянь и Ван Ливен (см [181]) построили децентрализованное решение для кольцевых сетей, древесных сетей, а также для сетей произвольной топологии. Внимательное изучение этих решений показывает, что все они очень похожи друг на друга, и отличие проявляется лишь в тех волновых алгоритмах, на которых основываются эти решения.

В данном параграфе мы приведем набросок общего метода построения алгоритма обнаружения завершения вычисления (вместе с его инвариантом), в основу которого положен произвольный, а не специальный (например, кольцевой) волновой алгоритм. Для каждой волны первое событие отправления процессом сообщения по ходу этой волны или принятия им решения мы будем называть *посещением* волной указанного процесса. Предполагается, что в случае необходимости процесс может отложить посещение, пока не будут выполнены некоторые локальные условия. Последующие события, связанные с прохождением той же самой волны через указанный процесс уже никогда не откладываются.

В этом параграфе мы проведем построение алгоритма только для случая синхронного обмена базовыми сообщениями, как это было осуществлено в §8.3.1. Наше построение можно обобщить на случай асинхронного обмена сообщениями, подобно тому, как это было сделано в §§8.3.2 и 8.3.3.

Инвариант нашего алгоритма должен позволять обнаруживать завершение вычисления, когда волна принимает решение. Поэтому в качестве первого приближения мы полагаем $P = P_0$, где

$$P_0 \equiv \text{все посещенные процессы пассивны.}$$

Действительно, коль скоро к моменту принятия решения были посещены все процессы, это требование позволяет обнаружить завершение вычисления в том случае, когда волна принимает решение. Но требование P_0 , кроме того, соблюдается и в том случае, когда волна запускается (и ни один процесс еще не посещен). Условие P_0 сохраняется в ходе работы волнового алгоритма, если принять во внимание следующее правило.

Правило 1. Волна посещает только пассивные процессы.

К сожалению, P_0 будет опровергнуто, когда процесс, посещенный волной, активизируется другим процессом, который волна еще не посетила. Поэтому каждый процесс снабжается окраской и требование P замещается ослабленным условием $(P_0 \vee P_1)$, где

$$P_1 \equiv \text{имеется процесс цвета } black, \text{ который еще не был посещен.}$$

Ослабленный инвариант сохраняется при выполнении следующего правила.

Правило 2. Процесс, отправивший базовое сообщение, приобретает окраску *black*.

Но даже и это условие может быть опровергнуто по ходу движения волны, если она посетила единственный ранее не посещенный процесс цвета *black*. Чтобы найти выход из положения, нужно еще более ослабить требование P . Каждый процесс будет придавать волне окраску *white* или *black*. Волна преобразуется таким образом, чтобы по ходу ее движения определялась *самая темная* из приданных ей окрасок; здесь следует напомнить, что волновые алгоритмы способны вычислять точные нижние грани, и в качестве таковой рассматривается «самая темная окраска». Когда волна принимает решение, определяется самая темная из всех приданных ей окрасок. Это будет окраска *white*, если все процессы придали волне окраску *white*, и окраска *black*, если хотя бы один процесс привнес в волну окраску *black*. Мы будем полагать, что волна по ходу своего движения имеет окраску *white*, если ни один процесс еще не привнес в нее окраску *black*; если же хотя бы один процесс привнес в волну окраску *black*, то и вся волна будет иметь цвет *black*. Таким образом, всякий процесс после посещения его волной либо привносит в волну окраску *white*, сохраняя ее цвет неизменным, либо привносит окраску *black* и окрашивает тем самым всю волну в цвет *black*.

Требование P замещается еще более ослабленным условием $(P_0 \vee P_1 \vee P_2)$, где

$$P_2 \equiv \text{волна имеет окраску } black.$$

Это условие сохраняется при выполнении. Это условие сохраняется при выполнении следующего правила.

Правило 3. Всякий процесс, который посетила волна, придает волне свою текущую окраску.

И действительно, обмен базовыми сообщениями, равно как и действия, связанные с прохождением волны, сохраняют указанное условие, и оно, таким образом, становится инвариантом. Волна оканчивается безрезультатно, если процессы, принимающие решения, находят, что волна окрашена в цвет *black*. Но в таком случае просто запускается очередная волна. Эта новая волна может привести к успеху только тогда, когда процессы приобретут окраску *white*, а это случится сразу после того, как их посетит эта волна.

Правило 4. Процесс, принявший решение в том случае, когда волна имела окраску *black*, запускает новую волну.

Правило 5. Каждый процесс приобретает окраску *white* сразу после того, как его посетила очередная волна.

Эти правила позволяют гарантировать, что рано или поздно после завершения базового вычисления прохождение очередной волны будет успешным. Действительно, если базовое вычисление завершилось, то первая волна, запущенная после этого завершения вычисления, окрасит все процессы в цвет *white* и следующая за ней волна успешно завершится.

В этом алгоритме в каждый момент времени может распространяться только одна волна. Если две волны, назовем их А и В, распространяются параллельно, то «побелка» процесса волной В может нарушить инвариант для волны А. Поэтому если возникает необходимость в децентрализованном алгоритме обнаружения завершения вычисления, то децентрализованный волновой алгоритм должен быть применен так, чтобы все инициаторы алгоритма обнаружения работали совместно над одной и той же волной. Можно воспользоваться также и другим принципом обнаружения, согласно которому разные волны могут вычисляться параллельно, не вызывая нарушений в правильной работе алгоритма обнаружения завершения вычислений (см. § 8.4.2).

8.4. Прочие решения

В этом параграфе мы обсудим два других метода решения задачи обнаружения завершения вычислений: алгоритм возвращения кредита и алгоритм штемпеля времени.

8.4.1. Алгоритм возвращения кредита

В статье [142] Маттерн предложил алгоритм, позволяющий обнаруживать завершение вычисления очень быстро, точнее говоря, спустя не более одной единицы времени, после того как это случилось (принимая во внимание идеализированные допущения о времени, представленные в определении 6.31). Алгоритм обнаруживает завершение централизованного вычисления при условии, что каждый процесс способен отправить сообщение непосредственно самому инициатору вычисления (т. е., сеть содержит подграф, имеющий форму звезды, в центре которой расположен инициатор).

В этом алгоритме каждому сообщению и каждому процессу вручается некоторый *кредит*; его значением служит вещественное число, расположенное между 0 и 1 (включительно). Алгоритм стремится сохранить инвариантность следующих утверждений.

S1. Суммарный кредит, выданный сообщениям и процессам, равен 1.

S2. Всякому базовому сообщению вручается положительный кредит.

S3. Всякому активному процессу вручается положительный кредит.

Процессы, имеющие положительный кредит и не подпадающие под перечисленные законы (т. е. пассивные процессы), возвращают кредит инициатору. Инициатор поступает подобно банку и собирает все кредиты, которые ему отправляют, в переменную *ret* (*возвращенные кредиты*).

Когда инициатор владеет всеми кредитами, из требований, предписывающих активным процессам и базовым сообщениям обладать положительным креди-

том, следует, что таковые процессы и сообщения отсутствуют. Следовательно, предикат **term** будет истинен.

Правило 1. Если $ret = 1$, то инициатор вызывает процедуру *Announce*.

Чтобы выполнялось необходимое условие корректности, нужно обеспечить возвращение всех кредитов инициатору по завершении базового вычисления. Если базовое вычисление завершилось, то никаких базовых сообщений уже нет и нам нужно разобраться с теми кредитами, которыми обладают процессы.

Правило 2. Как только процесс становится пассивным, он отправляет свой кредит инициатору.

```

var statep : (active, passive) init if  $p = p_0$  then active else passive ;
credp : fraction init if  $p = p_0$  then 1 else 0 ;
ret : fraction init 0 ; for  $p_0$  only

```

```

Sp: { statep = active } (* Правило 3 *)
begin send (mes, credp/2) ; credp := credp/2 end

```

```

Rp: { сообщение (mes, c) поступило процессу p }
begin receive (mes, c) ; statep := active ;
      credp := credp + c (* Правила 4 и 5b *)
end

```

```

Ip: { statep = active }
begin statep := passive ;
      send (ret, credp) to p0 ; credp := 0 (* Правило 2 *)
end

```

```

Ap0: { Сообщение (ret, c) поступило процессу p0 }
begin receive (ret, c) ; ret := ret + c ;
      if ret = 1 then Announce (* Правило 1 *)
end

```

Алгоритм 8.9. Алгоритм возвращения кредита

В начальной конфигурации активен один лишь инициатор, и он имеет положительный кредит, равный 1, причем $ret = 0$, а это означает, что требования S1–S3 соблюдены. По ходу вычисления нужно поддерживать инвариантность этих соотношений. Для этого служат следующие правила. Во-первых, каждому базовому сообщению должен быть выделен положительный кредит, когда оно отправляется; к счастью, отправитель сообщения активен и поэтому имеет положительный кредит.

Правило 3. Когда активный процесс p отправляет сообщение, имеющийся в его распоряжении кредит разделяется между самим процессом p и этим сообщением.

Всякому процессу должен быть выделен положительный кредит, когда он активизируется; к счастью, то сообщение, получение которого способствовало активизации, обладает положительным кредитом.

Правило 4. Когда процесс активизируется, ему вручается кредит, которым обладало активизировавшее его сообщение.

И лишь одну возможность не охватывают перечисленные правила — получение активным процессом базового сообщения. Такой процесс уже располагает положительным кредитом и поэтому для выполнения требования S_3 он не нуждается в том кредите, которым обладает указанное сообщение; однако этим кредитом нельзя пренебречь, ибо в таком случае было бы нарушено требование S_1 . Активный процесс, получивший сообщение, может распорядиться кредитом двумя способами, и оба способа приводят к корректному алгоритму.

Правило 5а. Как только активный процесс получает базовое сообщение, кредит, которым обладало это сообщение, возвращается инициатору.

Правило 5б. Как только активный процесс получает базовое сообщение, кредит, которым обладало это сообщение, добавляется к тому кредиту, которым располагал сам процесс.

Описанные правила приводят к алгоритму 8.9 (здесь приведен вариант, в котором реализовано правило 5б). В этом алгоритме предполагается, что каждый процесс осведомлен об имени инициатора (по крайней мере, в тот момент, когда процесс впервые становится пассивным). Когда инициатор становится пассивным, он отправляет сообщение самому себе.

Теорема 8.11. Алгоритм возвращения кредита (алгоритм 8.9) является корректным алгоритмом обнаружения завершения вычислений.

Доказательство. Алгоритм применяет правила 1–5, и поэтому формула $S_1 \wedge S_2 \wedge S_3$, где

$$\begin{aligned} S_1 &\equiv 1 = \left(\sum_{\langle \text{mes}, c \rangle} c \right) + \left(\sum_{p \in \mathbb{P}} \text{cred}_p \right) + \left(\sum_{\langle \text{ret}, c \rangle} c \right) + \text{ret} \\ S_2 &\equiv \forall \langle \text{mes}, c \rangle \text{ на этапе пересылки} : c > 0 \\ S_3 &\equiv \forall p \in \mathbb{P} : (\text{state}_p = \text{passive} \implies \text{cred}_p = 0) \\ &\quad \wedge (\text{state}_p = \text{active} \implies \text{cred}_p > 0), \end{aligned}$$

является инвариантом алгоритма. Завершение вычисления обнаруживается, когда $\text{ret} = 1$. С учетом инварианта это означает, что предикат **term** истинен.

Чтобы обосновать необходимое условие корректности, заметим, что после завершения вычисления не выполняется ни одно базовое действие. Следовательно, происходит только прием сообщений $\langle \text{ret}, c \rangle$, и с каждым приемом число сообщений на этапе пересылки сокращается на единицу. Поэтому алгоритм достигает заключительной конфигурации. В такой конфигурации базовые сообщения отсутствуют (по определению предиката **term**), для всех процессов p выполняется равенство $\text{cred}_p = 0$ (по определению предиката **term** и в силу требования S_3) и отсутствуют сообщения $\langle \text{ret}, c \rangle$ (ибо конфигурация является заключительной).

Это означает, что $\text{ret} = 1$ (в силу требования S_1) и завершение вычисления будет обнаружено. $\square$

Если применяется правило 5а, то число контрольных сообщений на единицу превосходит число базовых сообщений (мы принимаем в расчет также и сообщения, которые процесс p_0 отправляет самому себе, когда становится пассивным). Если применяется правило 5б, то число контрольных сообщений на единицу превосходит число внутренних событий в базовом вычислении, которое, в свою очередь, не более чем на единицу превосходит число базовых сообщений. Представляется, что правило 5б более предпочтительно с точки зрения сложности по числу сообщений контрольного алгоритма. Совсем иначе дело обстоит, когда речь заходит о битовой сложности. По правилу 5а каждая доля кредита в системе, за исключением ret , — это некоторая отрицательная степень числа 2 (т. е. она равна 2^{-i} для некоторого натурального i). Представляя кредит логарифмом по основанию $1/2$, мы сокращаем битовую сложность передаваемых сообщений.

Алгоритм возвращения кредита — единственный алгоритм в этой главе, который предусматривает включение дополнительной информации (значение кредита) в базовые сообщения. Этот прием, при котором к базовым сообщениям присоединяется дополнительная информация, называют «оседлать поросенка». Если такой эффект нежелателен, то кредит, выделенный базовому сообщению, может быть передан с контрольным сообщением, которое отправляется сразу же после этого базового сообщения. (В алгоритме, который представлен в следующем параграфе, также требуется «оседлать поросенка», если при реализации алгоритма используются логические часы Лампорта.)

Проблема может возникнуть, если кредит (выделяемый сообщениям и процессам) составляет фиксированное число битов. В таком случае существует наименьший положительный кредит и разделить эту величину кредита на два уже невозможно. Когда нужно разделить наименьший возможный кредит, базовое вычисление приостанавливается, до тех пор пока процесс не востребует дополнительного кредита от инициатора. Инициатор вычитает эту величину из ret (и в результате ret может принять отрицательное значение) и передает кредит запрашивающему процессу. После получения кредита базовое вычисление возобновляется. Это повышение кредита приводит к блокировке базового вычисления вопреки требованию, гласящему, что алгоритм обнаружения завершения вычисления не должен оказывать влияния на базовое вычисление. К счастью, такие операции случаются редко.

8.4.2. Решения, использующие штемпель времени

В этом параграфе рассматриваются решения задачи обнаружения завершения вычисления, использующие в своей основе штемпели времени. Предполагается, что все процессы снабжены часами (см. §2.3.3); для этой цели пригодны как аппаратные таймеры, так и логические часы Лампорта (см. раздел 2.3.3). Сам принцип обнаружения завершения вычислений был предложен в работе Раны [161].

```

var  $state_p$  : (active, passive) ;
 $\theta_p$  : integer init 0 ; (* Логические часы *)
 $unack_p$  : integer init 0 ; (* Число неподтвержденных сообщений *)
 $qt_p$  : integer init 0 ; (* Время недавно случившегося перехода в quiet *)

 $S_p$ : {  $state_p = active$  }
  begin  $\theta_p := \theta_p + 1$  ; send  $\langle mes, \theta_p \rangle$  ;  $unack_p := unack_p + 1$  end
 $R_p$ : { Сообщение  $\langle mes, \theta \rangle$  из  $q$  достигло  $p$  }
  begin receive  $\langle mes, \theta \rangle$  ;  $\theta_p := \max(\theta_p, \theta) + 1$  ;
    send  $\langle ack, \theta_p \rangle$  to  $q$  ;  $state_p := active$ 
  end
 $I_p$ : {  $state_p = active$  }
  begin  $\theta_p := \theta_p + 1$  ;  $state_p := passive$  ;
    if  $unack_p = 0$  then (*  $p$  переходит в состояние quite *)
      begin  $qt_p := \theta_p$  ; send  $\langle tok, \theta_p, qt_p, p \rangle$  to  $Next_p$  end
    end
 $A_p$ : { Подтверждение  $\langle ack, \theta \rangle$  достигло  $p$  }
  begin receive  $\langle ack, \theta \rangle$  ;  $\theta_p := \max(\theta_p, \theta) + 1$  ;
     $unack_p := unack_p - 1$  ;
    if  $unack_p = 0$  and  $state_p = passive$  then (*  $p$  переходит в quiet *)
      begin  $qt_p := \theta_p$  ; send  $\langle tok, \theta_p, qt_p, p \rangle$  to  $Next_p$  end
  end
 $T_p$ : { Маркер  $\langle tok, \theta, qt, q \rangle$  достиг  $p$  }
  begin receive  $\langle tok, \theta, qt, q \rangle$  ;  $\theta_p := \max(\theta_p, \theta) + 1$  ;
    if quiet( $p$ ) then
      if  $p = q$  then Announce
      else if  $qt \geq qt_p$  then send  $\langle tok, \theta_p, qt, q \rangle$  to  $Next_p$ 
  end

```

Алгоритм 8.10. Алгоритм Раны

Как и для решений из раздела 8.3.3, в основу решения Раны положен локальный предикат *quiet*(p), определенный для каждого процесса p так, что

$$quiet(p) \implies state_p = passive \wedge$$

∧ никакое базовое сообщение, отправленное p ,
не пребывает на этапе пересылки,

откуда следует импликация $(\forall p \text{ } quiet(p)) \implies \text{term}$. Как и ранее, условие *quiet* определяется следующим соотношением

$$quiet(p) \equiv (state_p = passive \wedge unack_p = 0).$$

Цель алгоритма — проверить, верно ли, что в определенный момент времени t все процессы удовлетворяют условию *quiet*. Отсюда следует, что в момент времени t вычисление завершилось. Это осуществляется при помощи волны, по ходу которой каждому процессу задается вопрос о том, достиг ли этот процесс в данный момент состояния *quiet* и будет ли он оставаться в этом состоянии в дальнейшем.

Всякий процесс, не пребывающий в состоянии *quiet*, не реагирует на сообщения волны, тем самым эффективно гася саму волну.

В отличие от решений, предложенных в разделе 8.5.3, взаимодействие волны с процессом p не оказывает влияния на переменные процесса p , используемые для обнаружения завершения вычисления. (Взаимодействие с волной может повлиять на переменные самого волнового алгоритма, а также, в случае использования логических часов Лампорта, на часы процесса.) Вследствие этого правильность работы нашего алгоритма не нарушается при параллельном распространении нескольких волн.

Алгоритм Раны — это децентрализованный алгоритм; каждый процесс выполняет один и тот же алгоритм обнаружения завершения вычисления. Децентрализованный алгоритм может быть также получен и в том случае, если снабдить процедуру из раздела 8.3.4 децентрализованным волновым алгоритмом. То решение, которое было предложено Раной, позволяет процессам инициировать их собственные волны, распространяемые параллельно.

Процесс p , перейдя в состояние *quiet*, сохраняет в памяти тот момент времени qt_p , когда произошло это событие, и запускает волну, чтобы проверить, все ли процессы уже перешли в состояние *quiet* к моменту времени qt_p . Если это так, то завершение вычисления считается обнаруженным. В противном случае существует процесс, который перейдет в состояние *quiet* позже, и он запустит новую волну. Именно по такому принципу устроен алгоритм 8.10, в котором используются логические часы Лампорта, а в качестве волнового алгоритма выбран кольцевой алгоритм.

Теорема 8.12. *Алгоритм Раны (алгоритм 8.10) является корректным алгоритмом обнаружения завершения вычислений.*

Доказательство. Чтобы обосновать необходимое условие корректности, предположим, что для конфигурации γ выполняется условие **term**, где a подтверждений все еще пребывают на этапе пересылки. Начиная с этого момента выполняются только действия A_p и T_p . Коль скоро выполнение каждого действия A_p приводит к сокращению на единицу числа сообщений $\langle ack, \theta \rangle$, пребывающих на этапе пересылки, может быть совершено лишь конечное число таких шагов. Каждый процесс переходит в состояние *quiet* самое большее один раз; следовательно, маркеры порождаются не более N раз, и каждый маркер передается не более N раз. Таким образом, спустя $a + N^2$ шагов алгоритм обнаружения завершения вычисления достигает заключительной конфигурации δ , в которой все еще выполняется условие **term**.

Обозначим символом p_0 процесс, у которого показатель времени qt в конфигурации δ является максимальным, т. е. в заключительной конфигурации для каждого процесса p выполняется неравенство $qt_{p_0} \geq qt_p$. Когда процесс p_0 пребывал в состоянии *quiet* в последний раз (т. е. в момент времени qt_{p_0}), он отправил маркер $\langle tok, qt_{p_0}, qt_{p_0}, p_0 \rangle$. Этот маркер совершил полный обход по кольцу и был возвращен процессу p_0 . Действительно, каждый процесс p в тот момент, когда к нему попал маркер, должен был пребывать в состоянии *quiet* и при этом должно было выполняться неравенство $qt_p \leq qt_{p_0}$. Если бы это было не так, то

процесс p по получении маркера пришлось бы установить на своих часах значение, превышающее qt_{p_0} , и перейти в состояние *quiet* позже, нежели это довелось процессу p_0 , вопреки нашему выбору процесса p_0 . Когда маркер был возвращен процессу p_0 , этот процесс все еще пребывал в состоянии *quiet* и поэтому вызвал процедуру *Announce*.

Чтобы обосновать достаточное условие корректности рассматриваемого алгоритма, предположим, что процесс p_0 вызывает процедуру *Announce*. Это происходит, когда p_0 пребывает в состоянии *quiet* и получает назад свой маркер $\langle \text{tok}, \theta, qt, p_0 \rangle$, который был переправлен ему поочередно всеми процессами. Проведем доказательство от противного. Допустим, что условие **term** не соблюдается в тот момент, когда процесс p_0 обнаруживает завершение вычисления; это означает, что имеется процесс p , который не удовлетворяет условию *quiet*. В таком случае процесс p перестал удовлетворять условию *quiet* уже *после того*, как расстался с маркером, выпущенным процессом p_0 ; ведь p должен был пребывать в состоянии *quiet*, когда передавал этот маркер. Пусть q — *первый* процесс, который перестал удовлетворять условию *quiet* после передачи маркера $\langle \text{tok}, \theta, qt, p_0 \rangle$. Это означает, что процесс q был активизирован в результате получения сообщения от некоторого процесса r , который еще не участвовал в передаче маркера, выпущенного процессом p_0 , ибо в противном случае процесс r перестал бы удовлетворять условию *quiet* после передачи этого маркера, но до того, как этому условию перестал удовлетворять процесс q , вопреки выбору q .

Теперь после передачи указанного маркера неравенство $\theta_q > qt_{p_0}$ будет по-прежнему соблюдаться. Отсюда следует, что в подтверждении, отправленном процессом q процессу r в ответ на сообщение, которое вывело q из состояния *quiet*, стоит метка времени $\theta_0 > qt_{p_0}$. Таким образом, когда r перейдет в состояние *quiet* после получения этого подтверждения, будет выполняться неравенство $\theta_r > qt_{p_0}$, и поэтому в тот момент, когда процессу r вручат маркер, будет выполняться соотношение $qt_r > p_0$. Согласно описанию алгоритма процесс r не станет передавать маркер, вопреки тому, что маркер совершил полный обход по кругу. $\square$

Доказательство корректности, опирающееся на инварианты и функцию нормирования, предложено в работе ван Везеля [200]. Модификация рассмотренного алгоритма, не зависящая от кольцевой топологии, предложена в работе Хуана [109].

8.5. Упражнения к главе 8

8.5.1

Упражнение 8.1. Опишите активные и пассивные состояния алгоритма A.2. Где можно найти эти состояния в алгоритме A.1?

8.5.2

Сложность по времени алгоритма обнаружения завершения вычислений определяется количеством единиц времени, которое разделяют в худшем слу-

чае (при идеализированных допущениях, указанных в определении 6.31) момент завершения базового вычисления и момент вызова процедуры *Announce*.

Упражнение 8.2. Какова сложность по времени алгоритма Дейкстры—Шолтена?

Упражнение 8.3. Алгоритм Шави—Франчета применяется к произвольной сети с уникальными отличительными признаками процессов, а для того, чтобы уменьшить издержки, связанные с передачей контрольных сообщений, в качестве волнового алгоритма используется алгоритм Галладжера—Хамблета—Спирсы. Сложность по времени обнаружения завершения вычисления составляет величину $\Omega(N \log N)$.

Можно ли уменьшить эту оценку сложности по времени до величины $O(N)$ за счет дополнительного обмена $O(N)$ контрольными сообщениями?

8.5.3

Упражнение 8.4. Почему предикат P_0 , используемый при описании алгоритма Дейкстры—Фейджена—ван Гастерена, *не* опровергается, когда процесс p_j активизируется процессом p_i так, что $j \leq t$ или $i > t$?

Упражнение 8.5. Покажите, что для всякого m существует такое базовое вычисление, в котором происходит обмен m базовыми сообщениями, и при этом алгоритм Дейкстры—Фейджена—ван Гастерена совершает обмен $m \cdot (N - 1)$ контрольными сообщениями.

8.5.4

Упражнение 8.6. Какие изменения следует внести в алгоритм 8.9, чтобы применить правило 5а из алгоритма возвращения кредита, вместо правила 5б?

Упражнение 8.7. В алгоритме Раны предполагается, что процессы наделены отличительными признаками. Предположим теперь, что все процессы анонимны, но обладают возможностью отправлять сообщения своим последователям по кольцу, и при этом число процессов заранее известно. Внесите в алгоритм 8.10 необходимые изменения, позволяющие ему работать в рамках таких допущений.

Упражнение 8.8. Обоснуйте корректность алгоритма Раны (алгоритм 8.10) на основе инвариантов этого алгоритма.

Упражнение 8.9. Внесите изменения в алгоритм Раны (алгоритм 8.10) так, чтобы для передачи сообщений можно было использовать произвольный волновой алгоритм, а не только кольцевой алгоритм.

ГЛАВА 9

АНОНИМНЫЕ СЕТИ

В предыдущих главах, для того чтобы различать процессы распределенной системы, обычно выдвигалось допущение об уникальности отличительных признаков. Это допущение справедливо применительно к большинству существующих распределенных систем. В качестве отличительного признака процесса часто выбирается адрес, по которому данному процессу отправляются сообщения. Однако для упомянутой цели процессы можно также идентифицировать посредством *косвенной адресации* (см. § 2.4.4), при которой каждый процесс выделяется среди своих соседей за счет локально известных имен каналов.

Совсем по-другому отличительные признаки использовались в гл. 7; в этой главе при помощи отличительных признаков удавалось разрушать симметрию на множестве процессов. Типичный пример подобного явления уже проявляется в простом алгоритме Ченя—Реверса (алгоритм 7.3), когда он иницируется двумя процессами p и q . В ходе его вычисления процесс p получает маркер, порожденный процессом q , а q получает маркер, порожденный процессом p . Ситуация была бы абсолютно симметричной, не будь у нас возможности упорядочить отличительные признаки процессов. Это упорядочение разрушает симметрию; наименьший из двух процессов выживает, а наибольший терпит неудачу. В гл. 7 можно отыскать и другие алгоритмы, которые дают более сложные примеры разрушения симметрии при помощи сопоставления отличительных признаков.

Отличительные признаки, обладающие глобальной уникальностью, применялись также и для обнаружения завершения вычислений, например в алгоритме Финна (алгоритм 6.8). Процесс обнаруживает, что ему удалось косвенным образом собрать информацию о всех процессах, если совпадают два семейства имен, где одно из этих семейств *Inc* включает имена всех соседей процессов из множества *NInc* (более подробно это описано в § 6.2.6). Подобное же использование отличительных признаков, но в гораздо более изощренной форме применяется и в алгоритме Галладжера—Хамблета—Спиры (см. § 7.3.2).

В этой главе сравниваются вычислительные возможности *анонимных* сетей и *именованных* сетей: какие из задач, разрешимые для именovaných сетей, можно также решить и в случае анонимных сетей? Этот вопрос совсем не праздный, хотя интересен он, главным образом, с чисто теоретической точки зрения, поскольку в большинстве распределенных систем все процессы снабжены уникальными именами. Но на практике мы можем столкнуться также и с анонимными сетями, когда дешевые (например встроенные) устройства объединяются в сеть. Примером здесь могут служить игрушки вида Lego MindStorm: многочисленные контроллеры можно собрать в модели и соединить друг с другом, но при этом кон-

троллеры являются серийными микросхемами, и все они совершенно идентичны. Прежде всего, результаты, представленные в настоящей главе, выделяют классы задач, которые нельзя решить, не прибегая к существенной помощи имен процессов. Возможно, неожиданным примером здесь служит реализация синхронного обмена сообщениями (см. § 9.1.4). Кроме того, представлены результаты, показывающие, что в случае конструирования сетей из идентичных процессоров, наподобие микросхем для транспьютеров или игрушек Lego MindStorm, должны быть соблюдены определенные условия. Компоненты должны быть снабжены генераторами случайных чисел, чтобы выполнять вероятностные алгоритмы, и размер сети должен быть известен процессорам, или же для сети необходимо применять централизованную инициализацию. И действительно, контроллеры MindStorm снабжены случайными числами.

В этой главе мы покажем, что в анонимных сетях можно разрушить симметрию, но невозможно обнаружить завершение вычисления, если размер сети заранее не известен. Разрушить симметрию можно при помощи *вероятностных алгоритмов*. В таких алгоритмах процессы многократно «подбрасывают монету», до тех пор пока не получат разные результаты; как только это случится, симметрия будет разрушена. Излишне говорить, что построить распределенный алгоритм, обеспечивающий сравнение и повторное подбрасывание, оказывается гораздо сложнее, нежели строго сформулировать его принцип действия (см. § 9.5.3). Обнаружение завершения возможно, если известен размер сети; подсчитывается число процессов, согласных с заявленным результатом, и завершение считается обнаруженным, если это число в точности равно размеру сети. Обнаружить завершение вычисления, используя подбрасывание монеты, нельзя, поскольку существует очень малая вероятность того, что при таком подбрасывании процессы будут получать одинаковые результаты, после чего может быть принято ошибочное заключение о завершении вычисления (см. § 9.5.4).

Как уже было отмечено, вероятностные алгоритмы могут быть плодотворно использованы в анонимных сетях для разрушения симметрии; их формальное определение приведено в § 9.5.1. В связи с этим возникает несколько различных вероятностных определений корректности, которые приведены в том же параграфе. Вероятностные алгоритмы очень важны для анонимных сетей, но помимо этого они представляют интерес и в других разделах теории распределенных вычислений. Так, в гл. 14 будет показано, что они играют важную роль для достижения отказоустойчивости распределенных систем. Кроме того, для некоторых задач, разрешимых при помощи детерминированных алгоритмов, существуют вероятностные решения, имеющие меньшую сложность; однако далее эту тему мы в нашей книге обсуждать не будем.

9.1. Основные понятия

Анонимной сетью называется сеть, в которой невозможно различать процессы по индивидуальным отличительным признакам. Предполагается также, что нет и заранее выделенного лидера. Следовательно, процессы, подключенные к одному и тому же числу каналов, совершенно идентичны. Энглуин в работе [8]

предложила использовать термин *процессный ряд* для бесконечной последовательности процессов $P_1, P_2, \dots$, где P_i — процесс степени i (т. е. подключен к i каналам). Для обозначения сетей, в которых все процессы одинаковы (симметричны), используется также термин *симметричные системы*.

9.1.1. Определения

В этом параграфе будет показано, как можно обобщить модель распределенных систем, введенную нами в гл. 2, чтобы охватить алгоритмы, опирающиеся на рандомизацию. Рандомизация состоит в том, чтобы при помощи «электронного подбрасывания монеты» и генератора случайных чисел добиться случайного поведения процесса, причем каждое из возможных случайных проявлений с известной вероятностью. Рандомизацию следует отличать от недетерминизма, обусловленного как непредсказуемостью относительного быстрогодействия процессов, так и тем обстоятельством, что всякий процесс наделен возможностью по-разному продолжить одно и то же вычисление за счет реализации различных событий. Мы применяем программное средство, чтобы определить распределение вероятностей на совокупности вычислений системы, и благодаря этому получаем возможность говорить о том, что свойства алгоритма соблюдаются с определенной вероятностью. Недетерминизм, присущий распределенным системам, нельзя контролировать, и поэтому проектировщик алгоритмов должен всегда уметь совладать с наихудшим из всех вариантов выбора, возникающим в связи с неустраняемым недетерминизмом.

Вероятностные алгоритмы. Модель *вероятностного процесса* образуется за счет того, что некоторый процесс подбрасывает монету на каждом шаге своего выполнения. Совокупность очередных возможных шагов зависит не только от состояния процесса, но также и от исходов подбрасывания монеты на предыдущих шагах процесса. Первый шаг процесса зависит от исхода подбрасывания, совершенного в начальном состоянии.

Невероятностные процессы (или алгоритмы) обычно называют *детерминированными процессами* (или алгоритмами); однако этот термин не исключает возможности того, что сеть невероятностных процессов функционирует недетерминированно. Такого рода сети ведут себя недетерминированно в соответствии с моделью из гл. 2, но при этом выбор обусловлен расписанием событий в вычислении, и он не поддается никакому вероятностному анализу. А вот выбор, возникающий в связи с рандомизацией, подчиняется вероятностному анализу, благодаря которому на множестве вычислений системы вводится распределение вероятностей. Чтобы сделать это распределение явным, в формальной модели предполагается, что вся последовательность подбрасываний монеты, которая осуществляется процессом по ходу его выполнения, задана заранее до начала выполнения.

Чтобы лучше понять формальное определение, мы вначале рассмотрим процессы, допускающие только внутренние события. *Вероятностный процесс* задается четверкой $p = (Z, I, \vdash^0, \vdash^1)$, где элементы множеств Z и I точно такие же,

как и в определении 2.4, а $\vdash^0$ и $\vdash^1$ — это бинарные отношения на Z . Процесс совершает i -й шаг согласно отношению $\vdash^0$, если исходом i -го подбрасывания монеты был ноль, и согласно отношению $\vdash^1$ в противном случае. Рассмотрим бесконечную последовательность $\rho = (\rho_0, \rho_1, \dots)$ исходов подбрасывания монеты, т. е. последовательность, состоящую из нулей и единиц. Тогда ρ -*выполнением* процесса p назовем такую максимальную последовательность состояний $c_0, c_1, \dots$, что $c_0 \in I$ и $(c_i, c_{i+1}) \in \vdash^{\rho_i}$. Отметим, что неустраняемый недетерминизм по-прежнему присутствует, поскольку заданному исходу подбрасывания монеты могут отвечать различные события. Однако никаких вероятностных допущений относительно их выбора сделать нельзя. А вот что касается вероятностного выбора, то здесь делается допущение о том, что для всякого $k \geq 0$ любая последовательность из k битов осуществляется в качестве префикса ρ с одной и той же вероятностью (а именно 2^{-k}). Если последовательность ρ задана, то вероятностный процесс ведет себя так же, как и детерминированный процесс, который можно рассматривать как частный случай вероятностного процесса.

Чтобы ввести обмен сообщениями между процессами, всякий процесс определяют восьмеркой, состоящей из множества состояний, множества начальных состояний и шести отношений событий, а именно событий отправления, событий приема и внутренних событий как для нулевого, так и для единичного исхода подбрасывания монеты. *Вероятностный распределенный алгоритм* представляется в виде совокупности $\mathbb{P}$ вероятностных процессов. Вместо того чтобы выписывать полное формальное определение, мы укажем здесь отличия данного определения от определения, приведенного в § 2.1.2. *Конфигурация* вероятностного распределенного алгоритма складывается из состояния и натурального числа для каждого из процессов и совокупности сообщений, если в алгоритме используется асинхронный обмен сообщениями. Для каждого процесса натуральное число обозначает количество событий, осуществленных данным процессом. Переходы системы определяются так же, как и в § 2.1.2, с той лишь разницей, что всякий переход, относящийся к процессу p , увеличивает на единицу число шагов, выполненных p .

Пусть ρ — это приписывание последовательностей ρ_p , состоящих из нулей и единиц, каждому процессу p . Тогда ρ -*вычислением* системы называется вычисление, в котором i -й переход, относящийся к процессу p , задается отношением событий, определяемым i -м элементом последовательности ρ_p . Если совокупность последовательностей ρ задана, то вероятностный алгоритм ведет себя так же, как и детерминированный распределенный алгоритм, который, в свою очередь, может рассматриваться как частный случай вероятностного распределенного алгоритма. Вероятностное допущение состоит в том, что для каждого $k > 0$ всякая совокупность из N последовательностей длины k имеет одну и ту же вероятность осуществления (а именно 2^{-kN}) в качестве префиксов последовательностей из ρ . Вследствие этого допущения мы получаем, что в каждом частном случае вероятностного алгоритма всякий процесс выполняет тот или иной локальный алгоритм с вероятностью, равной единице.

Вероятностная корректность и сложность. В этой главе мы будем рассматривать задачи, критерий правильного решения которых характеризуется *постусловием* ψ ; цель алгоритма состоит в том, чтобы достичь заключительной конфигурации, в которой выполняется ψ . Это постусловие зависит от конкретной задачи; для задачи об избрании его следует понимать как «один процесс пребывает в состоянии *leader* а все остальные процессы — в состоянии *lost*», а для задачи вычисления размера сети ψ определяется требованием «для каждого процесса p значение $size_p$ равно N ». Заключительная конфигурация может быть либо завершенной по признаку окончания приема сообщений (все процессы ожидают поступления какого-нибудь сообщения), либо завершенной по признаку окончания работы процессов (все процессы пребывают в заключительном состоянии).

Корректность детерминированных алгоритмов задается так, как это было определено в §2.5.2. Детерминированный алгоритм *завершается*, если каждое вычисление достигает заключительной конфигурации. Детерминированный алгоритм считается *частично корректным* (относительно предусловия ϕ), если каждая заключительная конфигурация данного алгоритма удовлетворяет ψ . Детерминированный алгоритм называется *корректным*, если он завершается и является частично корректным.

Для вероятностных алгоритмов корректность определяется на основании анализа частных случаев алгоритма, полученных при уточнении приписывания последовательностей ρ . Вероятностный алгоритм называется *ρ -завершающимся*, если каждое ρ -вычисление достигает заключительной конфигурации. Вероятностный алгоритм считается *ρ -частично корректным* (относительно предусловия ϕ), если каждая заключительная конфигурация всякого ρ -вычисления удовлетворяет предусловию ϕ . Вероятностный алгоритм называется *ρ -корректным*, если он является ρ -завершающимся и ρ -частично корректным.

Вероятностный алгоритм считается *завершающимся*, если он является ρ -завершающимся для каждого приписывания последовательностей ρ . Вероятностный алгоритм называется *частично корректным*, если он является ρ -частично корректным для каждого ρ . Вероятностный алгоритм называется *корректным*, если он является и ρ -корректным для каждого ρ .

Вероятностный алгоритм *завершается с вероятностью P* , если вероятность того, что алгоритм является ρ -завершающимся, не меньше P . Вероятностный алгоритм является *частично корректным с вероятностью P* , если вероятность того, что алгоритм является ρ -частично корректным, не меньше P . Вероятностный алгоритм является *корректным с вероятностью P* , если вероятность того, что алгоритм является ρ -корректным, не меньше P .

Завершаемость (равно как частичная корректность или корректность) не равносильна завершаемости (соответственно частичной корректности или корректности) с вероятностью единица. Вероятностный алгоритм может давать неверный результат для непустого множества приписываний ρ , но при этом вероятность того, что приписывание ρ принадлежит этому множеству, равна нулю.

Определим *ρ -сложность по числу обменов сообщениями* вероятностного алгоритма как наибольшее число обменов сообщениями, которое возможно

в ρ -вычислении. *Сложность в среднем по числу обменов сообщениями* вероятностного алгоритма — это математическое ожидание ρ -сложности по числу обменов сообщениями. Аналогично определяются сложность вероятностных алгоритмов по времени и по числу битов.

Обоснование отрицательных результатов. В этой главе будет доказан ряд отрицательных результатов, т. е. утверждений о том, что не существует алгоритмов, которые могли бы достичь тех или иных постусловий. Для детерминированных алгоритмов доказательство утверждений такого рода обычно построить проще; достаточно привести пример единственного вычисления, которое не завершается конфигурацией, удовлетворяющей постусловию ψ . Типичный пример доказательства такого рода представлен для теоремы 9.5.

Построение единственного бесконечного вычисления не опровергает существования нужного вероятностного алгоритма; такой алгоритм может иметь бесконечно много вычислений и при этом завершаться с вероятностью, равной единице. Чтобы доказать, что вероятностное решение невозможно, можно показать, как построить некорректное решение исходя из предполагаемого корректного вычисления, имеющего другую начальную конфигурацию. Примеры доказательства такого рода представлены в теоремах 9.12 и 9.13.

9.1.2. Классификация вероятностных алгоритмов

Вероятностные алгоритмы в зависимости от гарантируемой степени корректности подразделяются на следующие типы: алгоритмы типа *Лас-Вегас*, алгоритмы типа *Монте-Карло* и алгоритмы типа *Шервуд*. Вероятностные алгоритмы могут быть корректными (т. е., частично корректными и завершающимися), или только корректными с определенной вероятностью, как это было определено в §9.1.1.

Алгоритмы типа Шервуд. Корректные вероятностные алгоритмы называются *алгоритмами типа Шервуд* (см. [36]). Если нас интересует только разрешимость задачи (а не сложность ее решения), алгоритмы типа Шервуд представляют незначительный интерес, поскольку справедливо следующее утверждение.

Теорема 9.1. *Если существует корректный вероятностный алгоритм, в котором достигается постусловие ψ , то существует и корректный недетерминированный алгоритм, в котором достигается постусловие ψ .*

Доказательство. Предположим, что PA — корректный вероятностный алгоритм. Алгоритм PA является ρ -корректным для *каждого* ρ и, в частности, для такого ρ , в котором каждому процессу приписывается последовательность, состоящая из одних нулей; обозначим такое приписывание символом $\rho^{(0)}$.

Рассмотрим алгоритм DA , который действует так же, как действует PA в том случае, когда исход всякого подбрасывания монеты равен нулю (т. е. всякое обращение к случайному биту заменяется в программе числом 0). Такой алгоритм является детерминированным, и при этом он остается корректным, поскольку PA является $\rho^{(0)}$ -корректным. $\square$

Из этого результата не следует, что алгоритмы типа Шервуд бесполезны; их сложность в среднем может свидетельствовать об их преимуществе по сравнению со сложностью (в худшем случае) наилучших детерминированных алгоритмов. Рассмотрим в качестве примера алгоритм избрания Ченя—Робертса (алгоритм 7.3), который в худшем случае имеет сложность по числу сообщений, составляющую величину $\Theta(N^2)$, но при этом его сложность в среднем оценивается величиной $\Theta(N \log N)$. В рандомизованном варианте этого алгоритма каждый процесс дополняет свое имя случайно выбранной битовой строкой; эта случайно выбранная часть наиболее существенна для проведения сравнений. Независимо от распределения исходных отличительных признаков, математическое ожидание сложности модифицированного алгоритма равно сложности в среднем исходного алгоритма, т. е. составляет $\Theta(N \log N)$ сообщений.

В этой книге мы не будем касаться алгоритмов типа Шервуд.

Алгоритмы типа Лас-Вегас и Монте-Карло. Как следует из теоремы 9.1, самое большее, чего можно достичь при решении задачи, для которой не существует детерминированного решения, — это построить вероятностный алгоритм, который будет корректен с вероятностью P ($0 \leq P \leq 1$). Обычно каждое вычисление удовлетворяет либо требованию завершаемости, либо требованию частичной корректности.

Определение 9.2. Алгоритмом типа *Лас-Вегас* называется вероятностный алгоритм, который

- 1) завершается с положительной вероятностью и при этом
- 2) является частично корректным.

Алгоритмом типа *Монте-Карло* называется вероятностный алгоритм, который

- 1) завершается и при этом
- 2) является частично корректным с положительной вероятностью.

Вероятность завершения алгоритма типа Лас-Вегас чаще всего равна единице; бесконечные вычисления действительно возможны, но они случаются только тогда, когда неудачный набор случайных битов выбирается бесконечно часто. На практике это означает, что алгоритмы типа Лас-Вегас всегда завершаются, хотя при этом можно оценить лишь *среднее число* совершенных шагов, но нельзя указать верхней оценки числа шагов алгоритма. Примером алгоритма типа Лас-Вегас служит алгоритм избрания Айтаи и Роде (алгоритм 9.4).

Вероятность вычисления неверного результата алгоритмом Монте-Карло (если только это не алгоритм типа Шервуд) обычно меньше единицы. В конечном ошибочном вычислении используется лишь конечное число случайных битов, и поэтому вероятность такого вычисления положительна. Обычно удается установить верхнюю оценку числа шагов, совершаемых алгоритмом типа Монте-Карло. Примером алгоритма типа Монте-Карло служит алгоритм Айтаи и Роде вычисления размеров кольца (алгоритм 9.5).

При некоторых дополнительных ограничениях на основе алгоритма типа Лас-Вегас можно построить алгоритм типа Монте-Карло и наоборот; см. упражнения 9.1 и 9.2.

Свод характеристик и параметров. В этой главе все алгоритмы можно классифицировать согласно следующим четырем критериям; мы коротко отметим, алгоритмы какого типа представляются более привлекательными.

1. *Известный или неизвестный размер сети.* Алгоритмы, не опирающиеся на осведомленность о числе N , более предпочтительны, поскольку они являются более общими.

2. *Завершение по приему сообщений или по работе процессов.* Алгоритмы, завершающиеся по признаку окончания работы процессов, более предпочтительны, поскольку их завершение явно выражено и процессы могут использовать вычисленный результат.

3. *Детерминированный или вероятностный.* Детерминированные алгоритмы более предпочтительны, поскольку в них не используется генератор случайных чисел и, кроме того, свойства корректности, которым удовлетворяют такие алгоритмы, являются более сильными, нежели те, которым удовлетворяют алгоритмы типа Лас-Вегас и Монте-Карло.

4. *Тип Лас-Вегас или Монте-Карло.* Алгоритмы типа Лас-Вегас обычно представляются более привлекательными, поскольку завершение с некоторой вероятностью этих алгоритмов в практических ситуациях почти не отличимо от обычного завершения.

9.1.3. Задачи, рассматриваемые в этой главе

Наша цель состоит не в том, чтобы дать полное изложение теории вычислений на анонимных сетях, включая многочисленные результаты, которые были получены для этих сетей. Мы хотим лишь исследовать вычислительные возможности указанных сетей применительно к задачам избрания и вычисления функций, включая вычисление размера сети.

Эти задачи относятся к числу фундаментальных. Если можно избрать лидера, анонимная сеть будет обладать точно такими же вычислительными возможностями, как и сеть с именами, поскольку процессы можно именовать при помощи централизованного алгоритма, как это показано далее. В §9.5.3 будет показано, что лидер может быть избран при помощи вероятностного алгоритма, если известен размер сети, и это указывает на важность задачи о вычислении размера сети. Так как размер сети может быть легко вычислен централизованным алгоритмом (как это будет показано далее), задачи избрания и вычисления размера сети эквивалентны. Будет показано, что размер сети нельзя вычислить вероятностно, а задача о выборах неразрешима, если размер сети неизвестен.

Централизованные вычисления. Размер сети может быть легко вычислен при помощи одного-единственного вычисления алгоритма эха, если в распоряжении имеется единственный стартовый процесс (*leader*). Каждое сообщение,

отправляемое назад по остовному дереву, несет информацию о числе процессов в поддереве, корнем которого является процесс-отправитель; см. алгоритм 9.1.

```

var  $rec_p$       : integer init 0 ; (* Счетчик числа полученных сообщений *)
     $father_p$     :  $\mathbb{P}$       init undef ;
     $size_p$       : integer init 1 ; (* Размер поддерева с корнем  $p$  *)

```

Для инициатора:

```

begin forall  $q \in Neigh_p$  do send  $\langle tok, 0 \rangle$  to  $q$  ;
  while  $rec_p < \#Neigh_p$  do
    begin receive  $\langle tok, s \rangle$  ;  $rec_p := rec_p + 1$  ;
          $size_p := size_p + s$ 
    end (* Размер сети равен  $size_p$  *)
end

```

Для неинициаторов:

```

begin receive  $\langle tok, s \rangle$  from neighbor  $q$  ;  $father_p := q$  ;  $rec_p := rec_p + 1$  ;
  forall  $r \in Neigh_p, r \neq father_p$  do send  $\langle tok, 0 \rangle$  to  $r$  ;
  while  $rec_p < \#Neigh_p$  do
    begin receive  $\langle tok, s \rangle$  ;  $rec_p := rec_p + 1$  ;
          $size_p := size_p + s$ 
    end;
    send  $\langle tok, size_p \rangle$  to  $father_p$ 
  end
end

```

Алгоритм 9.1. Централизованное вычисление размера сети

Чтобы назначить процессам уникальные имена, сначала вычисляется размер сети при помощи алгоритма 9.1, после чего отрезок натурального ряда $\{0, \dots, N-1\}$ разделяется между процессами и каждое поддерево получает свой отрезок, длина которого соответствует его размеру.

Теорема 9.3. *Существует детерминированный централизованный алгоритм вычисления размера сети, который требует $2|E|$ обменов сообщениями и имеет сложность по времени, составляющую величину $O(N)$. Существует детерминированный централизованный алгоритм назначения уникальных имен процессам сети, который требует $2|E| + (N-1)$ обменов сообщениями и имеет сложность по времени, составляющую величину $O(N)$.*

Существует и более эффективный алгоритм назначения имен; см. упражнение 9.3. Как следует из теоремы 9.3, доступность лидера может со сравнительно небольшими издержками компенсировать отсутствие уникальных имен. Поэтому при изучении анонимных сетей предполагается, что наряду с отсутствием имен в сети нет и лидера.

9.1.4. Синхронный и асинхронный обмен сообщениями.

Результаты, связанные с выборами лидера на анонимных кликах, оказываются существенно зависящими от того, какого типа взаимосвязь проводится — с синхронным или асинхронным обменом сообщениями. При синхронном обмене сообщениями два процесса совместно участвуют в осуществлении единого перехода системы, и при этом симметрия между этими двумя процессами может быть разрушена.

Теорема 9.4. *Для сети, состоящей из двух процессов, взаимодействующих путем синхронного обмена сообщениями, существует детерминированный алгоритм избрания лидера, завершающийся по признаку окончания работы процессов.*

Доказательство. Каждый процесс имеет три состояния, а именно *sleep*, *leader*, и *lost*, причем начальным состоянием является *sleep*. Каждый процесс может либо отправить сообщение и перейти в заключительное состояние *leader*, либо принять сообщение и перейти в заключительное состояние *lost*.

В соответствии с этим система имеет девять конфигураций, из которых лишь три являются достижимыми; из начальной конфигурации (*sleep, sleep*) система может перейти в конфигурации (*leader, lost*) и (*lost, leader*). В каждой из этих двух конфигураций процессы завершили работу, и при этом в точности один из них пребывает в состоянии *leader*, в то время как другой пребывает в состоянии *lost*. $\square$

Этот алгоритм позаимствован из работы Энглуин [8]; она также обобщила данный алгоритм для случая клик произвольного размера. Если взаимосвязь осуществляется посредством асинхронного обмена сообщениями, то эта задача не имеет детерминированного алгоритма решения; это может быть установлено путем проведения рассуждений, аналогичных тем, которые будут использованы при доказательстве теоремы 9.5. Мы не хотим рассматривать решения, в которых симметрия разрушается за счет особенностей механизма взаимосвязи; поэтому мы ограничимся рассмотрением систем, в которых взаимосвязь осуществляется посредством асинхронного обмена сообщениями.

То решение, которое предложила Энглуин, очень существенно зависит от топологии; оно не пригодно, например, для кольцевых сетей. Результаты о невозможности получения некоторых решений, рассматриваемые в этой главе, будут обобщены для систем с синхронным обменом сообщениями в сетях с произвольной и кольцевой топологией.

Симметрия между двумя процессами может быть разрушена детерминированно, если сообщения передаются синхронно. Отсюда следует, что не существует детерминированного алгоритма, способного реализовать синхронный обмен сообщениями в анонимных сетях, в которых проводится асинхронный обмен сообщениями.

9.2. Детерминированные алгоритмы

В этом параграфе представлены некоторые результаты, касающиеся детерминированных алгоритмов. Наиболее важным является следующий отрицательный результат: избрать лидера детерминированно невозможно даже в том случае, когда речь идет о кольцевой сети, размер которой заранее известен. Будет также показано, что вычислять функции можно при условии, что размер кольца известен, и невозможно, если размер кольца неизвестен заранее.

9.2.1. Детерминированные выборы: отрицательные результаты

В этом параграфе будет показано, что провести детерминированные выборы лидера в анонимных сетях невозможно. Доказательство проводится с использованием понятия *симметрической конфигурации*. Показано, что существует вычисление, которое берет начало в симметрической конфигурации и достигает симметрической конфигурации через каждые N шагов. Невозможность получения детерминированного решения следует из того, что корректный алгоритм не завершает работу в симметрической конфигурации.

Теорема 9.5. *Не существует детерминированного алгоритма избрания лидера в анонимном кольце, размер которого заранее неизвестен.*

Доказательство. Мы приведем доказательство этого утверждения для случая однонаправленных колец, но если идея этого доказательства станет ясна, то распространить обоснование на случай двунаправленных колец не составит труда. Предположим, что ряд процессов, начинающийся процессом p_0 и оканчивающийся процессом p_{N-1} , образует кольцо так, что для каждого $i < N - 1$ процесс p_i может отправлять сообщения процессу p_{i+1} и процесс p_{N-1} может отправлять сообщения процессу p_0 . В конфигурации данного кольца обозначим символом c_i состояние процесса p_i , а символом M_i — совокупность сообщений, адресованных процессу p_i .

Конфигурацию будем называть *симметрической*, если состояния всех процессов одинаковы и также одинаковы множества сообщений, адресованных каждому процессу и пребывающих на этапе пересылки, т. е.

$$\forall i, j : (c_i = c_j \wedge M_i = M_j).$$

Для каждого алгоритма на данном кольце строится такое вычисление $C = (\gamma_0, \gamma_1, \dots)$, что каждая конфигурация γ_{kN} в последовательности C является симметрической. Обозначим символом γ_0 симметрическую начальную конфигурацию. Такая конфигурация существует, поскольку каждый процесс имеет одно и то же множество начальных состояний и первоначально все множества M_i — пустые.

Если конфигурация γ_{kN} является терминальной, то построение завершено; вычисление C завершается симметрической конфигурацией. В противном случае γ_{kN} является симметрической конфигурацией, в которой по меньшей мере одно событие осуществимо; предположим, что это событие e_i в процессе p_i . Симметричность конфигурации позволяет заключить, что такое же событие осуществимо

в каждом процессе. Из теоремы 2.19 следует, что это событие может быть осуществлено параллельно всеми процессами, и после выполнения этих N переходов образуется симметрическая конфигурация. Действительно, все процессы перешли из одинаковых состояний в одинаковые же состояния. Если событие предполагает прием сообщения, то это сообщение изымается из каждого из идентичных множеств M_i , а если происходит отправление сообщения, то оно добавляется к каждому из идентичных множеств M_i . Следовательно, продолжив наше вычисление и совершив N шагов, мы вновь придем к симметрической конфигурации $\gamma_{(k+1)N}$.

Таким образом, каждый алгоритм для данного анонимного кольца имеет вычисление, которое либо является бесконечным, либо завершается симметрической конфигурацией. В симметрической конфигурации либо все процессы пребывают в состоянии *leader*, либо ни один процесс не пребывает в этом состоянии; значит, в симметрической конфигурации один единственный процесс не может пребывать в состоянии *leader*. Для детерминированного алгоритма избрания лидера все конфигурации завершаются терминальной конфигурацией, в которой ровно один процесс пребывает в состоянии *leader*. Это означает, что таких алгоритмов для нашего кольца не существует. $\square$

Из теоремы 9.5 следует, что и более общая задача о выборах в произвольных сетях или кольцах неизвестного размера также неразрешима детерминированными алгоритмами.

9.2.2. Вычисление функций на кольце

В связи с невозможностью проведения выборов возникает вопрос о том, существуют ли более простые задачи, которые можно решить даже в том случае, если избрание лидера невозможно. В этом параграфе рассматривается задача вычисления значения функции на ориентированном анонимном кольце. Предположим, что на вход процесса p_i подаются данные $x_i \in X$, и пусть SX обозначает множество всех последовательностей элементов из X . Функцию f , определенную на SX , назовем *циклической*, если она инвариантна относительно циклических сдвигов входных данных, т. е. $f(y) = f(x)$ для всех циклических сдвигов y последовательности x . Примерами циклических функций могут служить функции суммирования, максимума и минимума на целых числах, булевы функции дизъюнкции, конъюнкции и сложения по модулю два, а также функция вычисления длины входа. Предположим, что процесс p наделен переменной $result_p$; тогда постусловие всякого алгоритма вычисления f имеет вид « $result_p = f(x_0, \dots, x_{N-1})$ ».

Результаты этого параграфа свидетельствуют о том, насколько важна информация о размере кольца, а также о том, насколько существенны различия в завершении вычисления по признаку окончания работы процессов и окончания приема сообщений.

Размер кольца известен. Если размер кольца известен, то всякая циклическая функция может быть вычислена некоторым детерминированным алгоритмом путем накопления всех входных данных каждым процессом. Такое решение носит

название *накопление входов*. Для накопления входов требуются $N(N - 1)$ сообщений, и эта величина служит нижней оценкой сложности вычисления некоторых функций детерминированными алгоритмами.

Теорема 9.6. *Если размер кольца известен, то всякая циклическая функция f может быть вычислена некоторым детерминированным алгоритмом, завершающим вычисление по признаку окончания работы процессов, с использованием $N(N - 1)$ сообщений.*

Доказательство. Чтобы сделать наше обоснование более наглядным, будем считать, что в каналах поддерживается очередность сообщений. По ходу работы алгоритма сообщения отправляются на протяжении $N - 1$ этапов. На первом этапе каждый процесс отправляет свои собственные входные данные соседу слева и принимает входные данные своего соседа справа. На каждом последующем этапе всякий процесс отправляет соседу слева данные, полученные им на предыдущем этапе, и принимает от соседа справа новые данные. Поскольку каждая порция входных данных на каждом этапе достигает всякий раз еще одного процесса, на i -м этапе всякий процесс принимает данные, принадлежащие его i -му по порядку соседу справа; таким образом, каждый процесс сумеет накопить все входные данные (в том порядке, в котором они располагаются относительно этого процесса на кольце) за $N - 1$ этапов. После окончания коммуникационной фазы каждый процесс вычисляет локально значение функции и завершает свою работу. $\square$

Теорема 9.7. *Всякий детерминированный алгоритм, завершающий вычисление функций AND, OR или SUM по признаку окончания работы процессов, использует $N(N - 1)$ сообщений в худшем случае.*

Доказательство. Рассмотрим детерминированный алгоритм A , завершающий вычисление по признаку окончания работы процессов и решающий одну из указанных задач. Определим *трассу* сообщений, подобно тому как это было сделано в определении и §7.2.3. Если процесс p отправляет сообщение перед тем, как впервые осуществить прием какого-либо сообщения, то для обозначения трассы этого сообщения будем использовать запись (p) . Если процесс p отправляет сообщение t в тот момент, когда самая длинная трасса полученного им ранее сообщения равна $(p_1, \dots, p_k)$, то трассой сообщения t будет запись $(p_1, \dots, p_k, p)$.

Все три задачи, упомянутые в формулировке теоремы, имеют одно общее качество: хотя бы для одной симметрической начальной конфигурации какой-либо из процессов, до того как он прекратит работу, должен получить сообщение, трасса которого имеет длину $N - 1$. Для задачи SUM это справедливо применительно к любой начальной конфигурации. Для задачи AND это будет иметь место, когда каждый процесс получает на входе значение *true*, а для задачи OR — когда каждый процесс получает на входе значение *false*. Чтобы убедиться, что сообщение, имеющее трассу длины $N - 1$, обязательно будет принято, рассмотрим вычисление AND в случае, когда каждый процесс получает на входе значение *true*; в таком случае будет вычислен ответ *true*. Допустим, что некоторый процесс завершает

работу, вычислив результат *true*, в то время, когда самая длинная трасса полученного этим процессом сообщения имеет длину $k < N - 1$. То событие, которое совершается в момент прекращения работы этого процесса, зависит только от событий, которые произошли в каких-либо $k + 1$ процессах, и, следовательно, оно произойдет и в ходе вычисления, в начальной конфигурации которого указанные $k + 1$ процессов получили на входе значение *true*, а остальные — *false*.

Как и прежде, назовем конфигурацию симметрической, если все процессы пребывают в одном и том же состоянии и в каждом канале содержится одно и то же множество сообщений. Точно так же, как и при доказательстве теоремы 9.5, строится вычисление, в котором симметрическая конфигурация образуется через каждые N шагов и в ходе которого в каждом процессе происходит одно и то же множество событий. Начальная конфигурация может быть выбрана таким образом, чтобы в ходе работы алгоритма было отправлено хотя бы одно сообщение, трасса которого имеет длину $N - 1$. В ходе такого вычисления для каждого натурального i , $i \leq N - 1$, будет отправлено сообщение, трасса которого имеет длину i . Но коль скоро в каждом процессе осуществляется одно и то же множество событий, общее число сообщений, трасса которых имеет длину i , кратно N , а значит, не может быть меньше N . Следовательно, суммарное число сообщений не может быть меньше $N(N - 1)$. $\square$

Размер кольца неизвестен. Если размер кольца неизвестен, то функции, отличные от констант, не могут быть вычислены никаким детерминированным алгоритмом, завершающим вычисление по признаку окончания работы процессов. Доказательство этого результата опирается на метод воспроизведения (правильного) вычисления алгоритма в различных частях кольца. Если же функция f равна константе (т. е., $f(x) = c$ для каждого x), то вычисление может быть выполнено тривиальным алгоритмом, который немедленно останавливается в каждом процессе p и выдает результат $result_p = c$; отрицательный результат относится только к тем функциям, которые не являются константами.

Теорема 9.8. *Если размер кольца заранее не известен, то не существует детерминированного алгоритма, завершающего вычисление по признаку окончания работы процессов и вычисляющего отличную от константы функцию.*

Доказательство. Так как функция f отлична от константы, существуют такие входные данные $x = (x_0, \dots, x_{l-1})$ и $y = (y_0, \dots, y_{l-1})$, что $f(x) \neq f(y)$; например, $f(x) = a$ и $f(y) = b$. Допустим, что A — это такой детерминированный алгоритм, завершающий вычисление по признаку окончания работы процессов, который допускает вычисление C_x на кольце с входными данными x , и при этом один из процессов p останавливается с результатом $result_p = a$, а также допускает вычисление C_y с входными данными y и один из процессов q останавливается с результатом $result_q = b$.

Допустим, что в ходе вычисления C_x самая длинная трасса сообщения, принятого процессом p , имеет длину K ; не исключено, что $K > k$, поскольку A может передавать сообщение по кругу несколько раз, прежде чем p прекратит работу.

сообщения, является заключительной, и, помимо того, в этом случае значением переменной $result_p$ является правильный ответ *false*.

Теперь рассмотрим случай, когда есть процесс, на вход которого подается *true*. Процесс p , на вход которого поступает *true*, присваивает переменной $result_p$ значение *true* и отправляет следующему процессу сообщение $\langle yes \rangle$. Так как каждый процесс отправляет не более одного сообщения (см. описание алгоритма), заключительная конфигурация будет достигнута и в этом случае. Если в заключительной конфигурации выполняется равенство $result_p = true$, то справедливо также и равенство $result_{Next_p} = true$, поскольку сообщение, отправленное процессом p будет получено процессом $Next_p$. Значит, если $result_p = true$ хотя бы для одного процесса p , то и для каждого процесса p будет выполняться равенство $result_p = true$, и поэтому в этом случае алгоритм завершит работу с правильным ответом для каждого процесса.

Вычисление функции AND проводится аналогично, но при этом все вхождения *true* и *false* следует поменять местами. $\square$

Частичная осведомленность о размере кольца. В качестве компромисса между полной осведомленностью и неосведомленностью о размере кольца можно рассмотреть такую ситуацию, при которой точное значение N неизвестно, но зато известны пределы, в которых оно заключено. Модифицировав доказательства, приведенные в настоящем параграфе, можно показать справедливость следующих утверждений.

1. Не существует детерминированного алгоритма вычисления функции SUM, который был бы корректным для двух колец разного размера.
2. Не существует детерминированного алгоритма вычисления функции XOR, который был бы корректным для колец, размер которых задается как четным, так и нечетным числом.
3. Если известна верхняя оценка S размера кольца, то функции AND и OR могут быть вычислены детерминированно с использованием $N(S-1)$ сообщений.

Другие топологии. Метод воспроизведения можно применять и для других классов сетевых топологий, но при этом требуется, чтобы небольшой граф можно было «обернуть» несколько раз вокруг графа большего размера. Такой тип отображения одного графа в другой называется *покрытием*.

Алгоритмы вычисления функций на анонимных сетях были построены также и для других топологий. Бем и Бодлаендер в работе [25] показали, что детерминированный сбор входных данных возможен на торе размера $n \times n$ с использованием $O(n^3)$ сообщений, а также, что величина $\Omega(n^3)$ является нижней оценкой сложности по числу сообщений для некоторых функций, включая AND и OR. Кранакис и Кризанц установили, что сбор входных данных возможен на n -мерном гиперкубе с использованием $O(2^n n^4)$ битов (см. [121]).

9.3. Вероятностный алгоритм избрания лидера

Если размер сети известен, то можно построить частично корректный алгоритм избрания, который завершает работу с вероятностью, равной единице, т. е.

относится к типу Лас-Вегас. Поскольку детерминированных алгоритмов не существует (см. теорему 9.5), алгоритм типа Лас-Вегас является наилучшим возможным. Мы приведем также алгоритм проведения выборов на анонимных кольцах неизвестного размера, предложенный Айтаи и Роде в работе [111]. В основу данного алгоритма положены те же принципы, которые использовались в алгоритме Ченя—Робертса (алгоритм 7.3), но для его применения потребуются некоторые изменения.

Поскольку в алгоритме Ченя—Робертса предполагается доступность отличительных признаков, всякий (анонимный) процесс p , который выступает в роли инициатора, прежде всего выбирает отличительный признак id_p случайным образом из множества $\{1, \dots, N\}$. Благодаря этому становится возможным проводить сравнение между маркерами различных процессов, но вместе с тем возникают и дополнительные трудности, связанные с тем, что несколько процессов могут выбрать один и тот же отличительный признак. В алгоритме Ченя—Робертса процесс узнает, что он получил свой собственный маркер, если отличительный признак маркера совпадает с отличительным признаком самого процесса, и это служит достаточным условием для того, чтобы процесс стал лидером.

Так как несколько процессов могут выбрать один и тот же отличительный признак, встроенный в маркер счетчик передач позволяет процессам распознавать получение их собственных маркеров; процесс p получает свой собственный маркер, если показатель счетчика переходов равен N . Получение такого маркера означает, что ни один процесс не выбрал меньший отличительный признак; но этого недостаточно, чтобы считаться избранным лидером, поскольку может оказаться так, что существует еще один процесс с тем же отличительным признаком. Чтобы распознать такую ситуацию, каждый маркер снабжен булевой переменной *un*, которая при генерации маркера имеет значение *true* и принимает значение *false*, если маркер передается процессом с тем же отличительным признаком. Таким образом, процесс становится лидером, когда он получает свой собственный маркер и при этом выполняется равенство $un = true$.

Последнее затруднение возникает в случае, когда процесс получает свой собственный маркер, но при этом выполняется равенство $un = false$; это означает, что процесс имеет минимальный отличительный признак, но при этом существует и другой процесс с тем же отличительным признаком. Все процессы с минимальным отличительным признаком могут получить свой собственный маркер и стать «победителями» алгоритма Ченя—Робертса. Если такое случится, то каждый «победитель» алгоритма Ченя—Робертса переходит в следующий тур и вновь запускает алгоритм Ченя—Робертса. Чтобы предотвратить очередную «ничью», каждый процесс начинает новый тур с того, что выбирает новый отличительный признак. Порядковый номер тура также отражен в маркере, и маркеры того или иного уровня пересекают всякую деятельность, относящуюся к турам с меньшим порядковым номером.

Подводя итог, заметим, что получение процессом p своего собственного маркера приводит к тому, что он становится победителем алгоритма Ченя—Робертса, если этот процесс пребывает в состоянии *cand*. Если p — единственный победитель, то он становится лидером; в противном случае p запускает алгоритм Че-

```

var  $state_p$  : (sleep, cand, leader, lost)   init sleep ;
       $level_p$  : integer                     init 0 ;
       $id_p$  : integer;
       $stop_p$  : bool                        init false ;
begin if  $p$  is initiator then
  begin  $level_p := 1$  ;  $state_p := cand$  ;  $id_p := \text{rand}(\{1, \dots, N\})$  ;
    send  $\langle \text{tok}, level_p, id_p, 1, true \rangle$  to  $Next_p$ 
  end;
while not  $stop_p$  do
  begin receive a message ; (* это либо маркер, либо сообщение о готовности *)
    if it is a token  $\langle \text{tok}, level, id, hops, un \rangle$  then
      if  $hops = N$  and  $state_p = cand$  then
        if  $un$  then (* считается избранным *)
          begin  $state_p := leader$  ; send  $\langle \text{ready} \rangle$  to  $Next_p$  ;
            receive  $\langle \text{ready} \rangle$  ;  $stop_p := true$ 
          end
        else (* на этом уровне есть и другие победители *)
          begin  $level_p := level_p + 1$  ;  $id_p := \text{rand}(\{1, \dots, N\})$  ;
            send  $\langle \text{tok}, level_p, id_p, 1, true \rangle$  to  $Next_p$ 
          end
        else if  $level > level_p$  or ( $level = level_p$  and  $id < id_p$ ) then
          begin  $level_p := level$  ;  $state_p := lost$  ;
            send  $\langle \text{tok}, level, id, hops + 1, un \rangle$  to  $Next_p$ 
          end
        else if  $level = level_p$  and  $id = id_p$  then
          send  $\langle \text{tok}, level, id, hops + 1, false \rangle$  to  $Next_p$ 
          else skip (* очистить маркер *)
        else begin send  $\langle \text{ready} \rangle$  to  $Next_p$  ;  $stop_p := true$  end
    end
  end

```

Алгоритм 9.4. Алгоритм Айтиа—Роде избрания лидера

ня—Робертса в следующем туре. Если процесс p получает маркер, порожденный другим процессом, то p переходит в состояние *lost* в том случае, когда этот маркер был порожден в одном из последующих туров или был порожден в том же туре, но при этом имеет меньший отличительный признак; маркер при этом передается далее по кругу. Маркер, порожденный в том же туре и несущий такой же отличительный признак, передается, но переменной un придается значение *false*. Маркер, порожденный в одном предыдущих туров, или порожденный в том же туре, но несущий меньший отличительный признак, изымается из обращения (т. е. его передача прекращается); см. алгоритм 9.4. Далее мы покажем, что алгоритм частично корректен и завершает работу с вероятностью, равной единице.

Лемма 9.10. *Если хотя бы один процесс порождает маркер в туре l , то либо один процесс будет избран лидером в этом туре и ни один процесс*

не порождает маркера в следующем туре $l+1$, либо хотя бы один процесс порождает маркер в туре $l+1$.

Доказательство. Процесс p порождает маркер в $(l+1)$ -м туре, если он получает свой собственный маркер (в l -м туре).

Допустим, что процесс p порождает маркер в l -м туре, но отличительный признак процесса не является минимальным среди отличительных признаков, которые были задействованы в l -м туре. В таком случае маркер не вернется к процессу p ; если этот маркер не будет устранен процессом, прошедшим в более высокий тур, он будет доставлен процессу, оперирующему в том же туре и имеющему меньший отличительный признак, где и будет устранен. (Обращаем внимание на то, что все процессы, которые порождают маркер в l -м туре, совершают это до того, как получают маркер процесса p в l -м туре.) Отсюда следует, что только тот процесс, которому достался минимальный отличительный признак в l -м туре, может быть избран в том же туре или получает право выработать маркер в $(l+1)$ -м туре.

Допустим, что процесс p порождает маркер в l -м туре и отличительный признак этого процесса является минимальным среди отличительных признаков, которые были задействованы в l -м туре. Если данный маркер не будет устранен каким-либо процессом, участвующим в более высоком туре (в таком случае хотя бы один процесс порождает маркер в $(l+1)$ -м туре), то этот маркер будет возвращен процессу p . Возможны два случая: либо p является единственным процессом, которому достался минимальный отличительный признак в l -м туре, либо имеются еще и другие процессы, обладающие таким же отличительным признаком. В первом случае маркер возвращается с пометкой $un = true$ и процесс p избирается лидером. Во втором случае, маркер возвращается с пометкой $un = false$ и в $(l+1)$ -м туре будет порожден какой-нибудь маркер. $\square$

Теорема 9.11. *Алгоритм Айтиа—Роде (алгоритм 9.4) является частично корректным алгоритмом избрания лидера для колец заранее неизвестного размера, завершающим работу с вероятностью, равной единице.*

Доказательство. Частичная корректность алгоритма, означающая, что в случае завершения работы остается в точности один лидер, доказывается с использованием леммы 9.10. Предположим, что l — это номер наивысшего тура, в котором вырабатывался маркер; величина l определена, поскольку по предположению рассматривается конечное вычисление. Тот факт, что ни один процесс не был избран в младших турах и ни один маркер не был порожден в $(l+1)$ -м туре, означает, что в l -м туре в точности один процесс был избран лидером.

Остается показать, что алгоритм завершает работу с вероятностью, равной единице. Если в некотором туре l имеются $k > 1$ процессов, которые выработали маркеры в туре l , то существует вероятность $P_k > 0$ того, что в этом туре в точности одному процессу достанется минимальный отличительный признак, и тогда алгоритм завершит работу. Обозначим символом P наименьшее значение P_k по всем k ; теперь если маркеры были выработаны в l -м туре, то вероятность того, что маркеры будут порождены также и в $(l+1)$ -м туре, не превосходит величину

ны $1 - P$. Следовательно, вероятность того, что алгоритм *не завершит* работу после l туров, ограничена сверху величиной $(1 - P)^l$, которая стремится к нулю, если устремить l к бесконечности. $\square$

Вероятностный анализ (проведенный в работе [111]) позволил выяснить, что ожидаемое число туров ограничено величиной $eN(N - 1)$. Ожидаемое число сообщений, отправленных в одном туре, ограничено величиной $O(N \log N)$ (это можно установить, рассуждая подобно тому, как это делалось в теореме 7.6), и отсюда следует, что ожидаемая сложность алгоритма по числу обменов сообщениями составляет величину $O(N \log N)$.

Произвольные сети. Воспользовавшись почти теми же самыми идеями, можно построить алгоритм проведения выборов в произвольных сетях. В основу этого алгоритма положен алгоритм 7.9, а не алгоритм Ченя—Робертса. Процесс «одерживает победу» в очередном туре этого алгоритма, если он инициировал алгоритм эха и все соседние процессы откликнулись эхом. Как и в алгоритме Айти—Роде, этого условия еще не достаточно, чтобы стать лидером, так как может оказаться, что выборы были затеяны несколькими процессами с одинаковыми отличительными признаками. Чтобы обнаружить это, размер дерева, построенного алгоритмом эха, вычисляется так, как это делается в алгоритме 9.1. Если есть несколько инициаторов с одинаковыми отличительными признаками, то каждый вызов эха приводит к построению дерева, в котором содержится только часть процессов. Когда процесс завершает выполнение алгоритма эха (получив отклик от каждого из соседей), ему становится известен размер дерева, которое при этом было построено. Если этот размер равен N , то процесс становится лидером, а иначе алгоритм эха запускается в следующем туре.

Полученный в результате алгоритм является частично корректным, и его работа завершается с вероятностью, равной единице. Полное описание этого алгоритма можно найти в статье [190]. Другие алгоритмы проведения выборов можно отыскать в работах [171] и [140].

9.4. Вычисление размера сети

Разнообразные результаты, представленные в этой главе, свидетельствуют о том, насколько важно знание размера сети. В этом параграфе мы рассмотрим задачу вычисления размера сети. Размер кольца может быть вычислен только при помощи алгоритма, оканчивающего работу по завершении обмена сообщениями, и при этом вероятность неправильного ответа будет отлична от нуля.

9.4.1. Отрицательные результаты

В этот параграф включены два результата, касающиеся невозможности вычисления размера кольца. Обоснование этих результатов опирается на ту же идею, которая использовалась для доказательства теоремы 9.8, а именно на идею о возможности повторения заданного (предположительно правильного) вычисления в различных частях большого кольца. Для вероятностных алгоритмов, как

будет показано, вероятность такого повторения отлична от нуля, и это следует из того, что при выполнении конечных вычислений всякий процесс использует лишь конечный префикс последовательности случайных битов.

Теорема 9.12. *Не существует алгоритма, оканчивающего вычисления по завершении работы процессов и правильно вычисляющего размер кольца с вероятностью $r > 0$.*

Доказательство. Предположим, что есть вероятностный алгоритм A , проводящий ρ -вычисление C_N на кольце размера N , и в вычислении C_N каждый процесс p завершает работу с результатом $result_p = N$. Выберем произвольный процесс p и предположим, что самая длинная трасса сообщения, полученного этим процессом p в вычислении C_N , имеет длину K . В вычислении C_N каждый процесс выполняет только конечное число шагов; обозначим символом L наибольшее число шагов, выполняемых всяким процессом в вычислении C_N .

Рассмотрим кольцо большего размера, в котором содержится сегмент из $K + 1$ процессов. Обозначим символом p_0 последний процесс этого сегмента, а символом p_i — i -й по порядку процесс сегмента, отстоящий от p_0 против хода часовой стрелки. Рассмотрим набор двоичных последовательностей ρ' , приписанных процессам этого кольца и обладающих следующим свойством. Первые L битов последовательности ρ'_{p_0} точно те же, что и первые L битов последовательности. Первые L битов последовательности ρ'_{p_i} , где $i \leq K$, точно те же, что и первые L битов последовательности ρ_q , где q — это i -й по порядку процесс, отстоящий от p против часовой стрелки в кольце размера N . Если длина сегмента $K + 1$ задана, то вероятность появления такого набора последовательностей очень мала, она равна $2^{-(K+1)L}$.

Существует ρ' -вычисление C алгоритма A на кольце большего размера, в ходе которого процесс p_i отправляет в точности те же сообщения, имеющие трассы длины не больше $\leq K + 1 - i$, что и i -й по порядку процесс, отстоящий против часовой стрелки от процесса p в малом кольце, в ходе вычисления C_N . Это доказывается точно так же, как делалось при обосновании теоремы 9.8. Как видно из построения, для всякого заданного сегмента длины $K + 1$ процессы этого сегмента с вероятностью, не меньшей $2^{-(K+1)L}$, повторяют вычисление C_N , в результате чего процесс p_0 завершит работу с результатом $result_{p_0} = N$, а этот результат неверен.

Вероятность того, что это произойдет на некотором заданном сегменте, чрезвычайно мала, но, выбрав кольцо достаточно большого размера, мы можем добиться того, что вероятность осуществления этого события в каком-либо сегменте кольца станет сколь угодно большой. Пусть задана вероятность $r > 0$. Возьмем такое натуральное число R , чтобы выполнялось неравенство $(1 - 2^{-(K+1)L})^R < r$, и рассмотрим кольцо размера $R(K + 1)$. В этом кольце есть R непересекающихся сегментов длины $K + 1$, и для каждого из этих сегментов вероятность того, что последний процесс не может завершить вычисление с неправильным ответом, не превосходит $1 - 2^{-(K+1)L}$. Тогда вероятность того, что ни один из процессов не завершит вычисление с неправильным ответом, будет меньше r . $\square$

На самом деле, применяя те же самые рассуждения, можно установить, что никакую функцию, отличную от константы, нельзя правильно вычислить с вероятностью $r > 0$ при помощи алгоритма, оканчивающего работу по завершении обмена сообщениями. А вот другой результат того же рода нельзя обобщить на случай произвольной функции, отличной от константы. Утверждается, что не существует алгоритмов типа Лас-Вегас, оканчивающих работу по завершении обмена сообщениями, и поэтому самое большее, на что можно рассчитывать, — это построение алгоритмов типа Монте-Карло, оканчивающих работу по завершении обмена сообщениями.

Теорема 9.13. *Не существует алгоритмов, оканчивающих работу по завершении обмена сообщениями и вычисляющих размер кольца с вероятностью, равной единице.*

Доказательство. Предположим, что есть вероятностный алгоритм A , имеющий ρ -вычисление C_N на кольце размера N , который оканчивает работу по завершении обмена сообщениями с результатом $result_p = N$ для некоторого процесса p . Каждый процесс совершает только конечное число шагов по ходу вычисления C_N . Обозначим символом L наибольшее число шагов, которое совершает всякий процесс в вычислении C_N . Занумеруем процессы в кольце размера N натуральными числами от 0 до $N - 1$.

Рассмотрим теперь кольцо размера $2N$, в котором все процессы занумерованы натуральными числами, начиная с 0 и оканчивая $2N - 1$, а также такое семейство ρ' двоичных последовательностей, что для каждого $i < N$ первые L битов последовательностей ρ'_{q_i} и $\rho'_{q_{i+N}}$, приписанных процессам q_i и q_{i+N} большого кольца, в точности повторяют первые L битов последовательности ρ_{p_i} , приписанной процессу p_i малого кольца. Устройство ρ' -вычисления алгоритма A на кольце размера $2N$ таково. Каждый шаг процесса p_i в вычислении C_N повторяется параллельно процессом q_i и его антиподом q_{i+N} . Поскольку последняя конфигурация вычисления C_N является заключительной, таковой же является и зпоследняя конфигурация вычисления C_{2N} , и в этой конфигурации есть по крайней мере два процесса q , вычисливших результат $result_q = N$, который в этом случае является неправильным.

Вероятность появления такого семейства последовательностей ρ' равна 2^{-2NL} . Это положительная величина, и она показывает, что алгоритм A не может правильно вычислить размеры колец с вероятностью, большей чем $1 - 2^{-2NL}$. $\square$

Различие вероятностей вычисления правильных результатов алгоритмами, оканчивающими работу по завершении работы процессов и по завершении обменов сообщениями, можно объяснить так. Алгоритм, оканчивающий вычисление по завершении работы процессов, выдает неверный результат, если где-то в кольце был совершен неудачный случайный выбор. При увеличении размеров кольца вероятность того, что это произойдет, возрастает. В алгоритме, оканчивающем работу по завершении обменов сообщениями, процессы, совершившие неудачный выбор, всегда имеют возможность оправиться от их неверного ответа за счет получения информации от других процессов. Следовательно, алгоритм, завершающий работу неявно, выдает неверный ответ, если неудачный случайный

выбор был совершен *везде* в заданном кольце. И хотя этого события нельзя совсем исключить, вероятность того, что оно произойдет, ограничена, и на самом деле она убывает по мере увеличения размеров кольца.

9.4.2. Алгоритм вычисления размера кольца

В этом параграфе представлен алгоритм типа Монте-Карло, вычисляющий размер кольца и оканчивающий работу по завершении обмена сообщениями. Как будет показано, по окончании работы алгоритма для всех процессов p будет выполняться равенство $est_p = N$ с вероятностью, которую можно сделать сколь угодно близкой к единице за счет выбора параметра R . Этот алгоритм сходен с алгоритмом, предложенным Айти и Роде [111], но немного упрощен.

Процесс p вычисляет локальную *оценку* размера кольца, est_p , и при этом гарантируется, что в любой момент времени эта оценка консервативна, т. е., $est_p \leq N$. Вначале $est_p = 2$. Всякий раз, когда процесс p получит информацию, свидетельствующую о том, что оценка est_p не равна размеру кольца, p увеличивает свою оценку.

Чтобы быть уверенным в правильности своей оценки, процесс p порождает маркер и отправляет его на расстояние est_p переходов по кольцу. Если оценка оказывается правильной, p получает обратно свой маркер, и если это происходит, p удовлетворяется результатом (временно) и не предпринимает дальнейших действий. Процесс p получает подобный маркер в случае, когда размер кольца N в действительности превосходит оценку est_p , но процесс, отстоящий от p на (est_p) переходов против часовой стрелки, выработал ту же самую оценку и отправил свой маркер.

Такую ситуацию (в которой процесс p получает не принадлежащий ему маркер) можно распознать с большой вероятностью; каждый процесс выбирает случайным образом метку из множества $\{1, \dots, R\}$ и присоединяет ее к своему маркеру. Метка процесса, выбранная процессом p , остается неизменной, до тех пор пока не увеличится оценка размера сети. Если процесс p получает маркер, выпущенный другим процессом, то с вероятностью $1 - 1/R$ присоединенная к нему метка отличается от метки процесса p , а если процесс получает свой собственный маркер, то его метка наверняка совпадает с меткой процесса.

Процесс p увеличивает локальную оценку размера сети в двух случаях.

1. *Получен маркер, содержащий оценку est , удовлетворяющую неравенству $est > est_p$.* Так как алгоритм обеспечивает консервативность всех оценок, получение такого маркера означает, что $N > est_p$.

2. *Получен маркер, содержащий оценку $est = est_p$, причем этот маркер совершил est переходов, но присоединенная к нему пометка отличается от пометки процесса p .* Этот маркер был выпущен процессом, отстоящим от p на (est) переходов против часовой стрелки, и этот процесс имеет пометку, отличную от пометки процесса p . Следовательно, (est) -й по порядку сосед процесса p отличен от самого p , а это означает, что $N \neq est$.

Маркер, несущий оценку $est < est_p$, изымается (т. е. его распространение прерывается); см. алгоритм 9.5.

```

cons  $R$       : integer    ; (* Определяет вероятность правильного результата *)
var   $est_p$    : integer;
       $lbl_p$    : integer;

begin  $est_p := 2$  ;  $lbl_p := \text{rand}(\{1, \dots, R\})$ ;
      send  $\langle \text{test}, est_p, lbl_p, 1 \rangle$  to  $Next_p$  ;
      while  $true$  do
        begin receive  $\langle \text{test}, est, lbl, h \rangle$  ;
          if  $est > est_p$  then  (* Оценка процесса  $p$  должна быть увеличена *)
            if  $est = h$  then    (*  $est$  также слишком мала *)
              begin  $est_p := est + 1$  ;  $lbl_p := \text{rand}(\{1, \dots, R\})$  ;
                send  $\langle \text{test}, est_p, lbl_p, 1 \rangle$  to  $Next_p$ 
              end
            else  (* отправляет маркер далее и увеличивает  $est_p$  *)
              begin send  $\langle \text{test}, est, lbl, h + 1 \rangle$  to  $Next_p$  ;  $est_p := est$  ;
                 $lbl_p := \text{rand}(\{1, \dots, R\})$ ; send  $\langle \text{test}, est_p, lbl_p, 1 \rangle$  to  $Next_p$ 
              end
            else if  $est = est_p$  then
              if  $h < est$  then send  $\langle \text{test}, est, lbl, h + 1 \rangle$  to  $Next_p$ 
              else (* Этот маркер совершил  $est$  переходов *)
                if  $lbl \neq lbl_p$  then
                  begin  $est_p := est + 1$  ;  $lbl_p := \text{rand}(\{1, \dots, R\})$  ;
                    send  $\langle \text{test}, est_p, lbl_p, 1 \rangle$  to  $Next_p$ 
                  end
                else skip  (* Возможно, что вернулся маркер процесса  $p$  *)
              else (*  $est < est_p$  *) skip
            end
          end
        end
      end

```

Алгоритм 9.5. Вероятностное вычисление размера кольца

Лемма 9.14. *Приведенный алгоритм оканчивает работу по завершении обмена $O(N^3)$ сообщениями. В заключительной конфигурации все переменные est_p равны, и их общее значение ограничено величиной N , т. е. для любой пары процессов p и q выполняется соотношение $est_p = est_q \leq N$.*

Доказательство. Решающий этап обоснования завершаемости алгоритма — это демонстрация того, что оценки действительно остаются консервативными. Заметим, что значение переменной est_p никогда не убывает, а значение переменной lbl_p изменяется только тогда, когда увеличивается значение est_p . Отсюда следует, что, пока циркулирует маркер $\langle \text{test}, est, lbl, h \rangle$, выпущенный процессом p , и выполняется равенство $est = est_p$, будет выполняться и равенство $lbl_p = lbl$. Воспользовавшись этим, можно провести индуктивное обоснование того, что все вычисленные оценки консервативны.

Так как оценки, заложенные в маркеры, вычислены самими процессами, достаточно рассмотреть вычисление оценок, осуществляемое процессами. Здесь можно выделить три случая.

1. Процесс p может увеличить значение est_p до est после получения маркера с оценкой est . А так как эта оценка была вычислена ранее, по предположению индукции можно считать, что она консервативна.

2. Процесс p может увеличить значение est_p до $est + 1$ после получения маркера с оценкой est , если счетчик переходов маркера также показывает est . В таком случае, est консервативна по индуктивному предположению и, кроме того верно соотношение $N \neq est$, поскольку процесс, выпустивший данный маркер, отличен от процесса p и отстоит от него на est переходов.

3. Наконец, процесс p может увеличить est_p до $est + 1$ после получения маркера с оценкой $est = h = est_p$, если его пометка $lbl \neq lbl_p$. Процесс, выпустивший этот маркер, отстоит на est переходов от p . А так как $lbl \neq lbl_p$, этот процесс будет отличен от p . И вновь отсюда следует, что $N \neq est$, а консервативность оценки est влечет неравенство $N > est$.

Каждый процесс порождает менее N различных маркеров, оценки которых пробегают числа от 2 до N , и при этом маркер с оценкой e совершает не более e переходов. Отсюда следует, что число обменов сообщениями ограничено величиной $O(N^3)$. Если процесс p увеличивает значение est_p , например до величины e , то p отправляет процессу $Next_p$ маркер, содержащий значение e . После получения этого маркера переменная est_{Next_p} будет иметь значение, не уступающее e , и поэтому в заключительной конфигурации выполняется неравенство $est_{Next_p} \geq est_p$. Оно выполняется для всех процессов p , и отсюда следует, что все оценки будут равны. А ранее мы показали, что они ограничены величиной N . $\square$

Теорема 9.15. *Алгоритм 9.5 всегда завершает работу, и после завершения работы для всех процессов p с вероятностью, не меньшей $1 - (N-2)(1/R)^{N/2}$, выполняется равенство $est_p = N$.*

Доказательство. Согласно лемме 9.14 работа алгоритма завершается такой конфигураций, в которой все оценки равны. Обозначим общее значение этих оценок символом e и предположим, что $e < N$. Обозначим процессы символами, начиная с p_0 и оканчивая p_{N-1} , в порядке их обхода по кольцу против хода часовой стрелки.

Процесс p_i выбирает пометку lbl_{p_i} . Он порождает маркер $\langle \text{test}, e, lbl_{p_i}, h \rangle$, когда значение est_{p_i} будет равно e . Этот маркер совершит e переходов и достигнет процесса p_{i+e} , который является e -м против хода часовой стрелки соседом процесса p_i ; процесс p_{i+e} , получив этот маркер, не станет увеличивать свою оценку. Это означает, что равенство $lbl_{p_i} = lbl_{p_{i+e}}$ соблюдается для всех i . Пусть $f = \text{НОД}(N, e)$. Тогда, учитывая равенства $lbl_{p_i} = lbl_{p_{i+e}}$, приходим к выводу о том, что всякий раз, когда номера i и j отличаются на величину, кратную f , пометки, выбранные процессами p_i и p_j совпадают, т. е.

$$f | (j - i) \implies lbl_{p_i} = lbl_{p_j}.$$

Все процессы, начиная с p_0 и оканчивая p_{N-1} , оказываются разбитыми на f групп по N/f процессов в каждой так, что все процессы из одной и той же группы имеют одинаковые пометки.

Но поскольку пометки были выбраны случайным образом из множества $\{1, \dots, R\}$, вероятность того, что все процессы из одной группы выбрали одну и ту же пометку, будет равна $(1/R)^{N/f-1}$, а вероятность того, что это произойдет для всех групп, будет равна всего лишь $(1/R)^{f(N/f-1)} = (1/R)^{N-f}$. Для всех возможных значений e , величина f не превосходит $N/2$, и, суммируя вероятности завершения работы со всеми возможными неправильными ответами (от 2 до $N-1$), мы приходим к заключению о том, что вероятность правильного ответа будет не меньше $1 - (N-2)(1/R)^{N/2}$. $\square$

9.5. Упражнения к главе 9

9.5.1

Упражнение 9.1. Пусть ψ обозначает некоторое постусловие. Предположим, что в нашем распоряжении имеются

1. алгоритм A типа Монте-Карло, оканчивающий вычисления по завершении работы процессов и удовлетворяющий постусловию ψ , и
 2. детерминированный алгоритм верификации B , оканчивающий вычисления по завершении работы процессов и проверяющий выполнимость ψ .
- Покажите, как построить алгоритм C типа Лас-Вегас, удовлетворяющий постусловию ψ .

Упражнение 9.2. Пусть ψ обозначает некоторое постусловие. Предположим, что в нашем распоряжении имеется алгоритм A типа Лас-Вегас. Предположим также, что известно математическое ожидание K числа обменов сообщениями в алгоритме A . Требуется построить алгоритм типа Монте-Карло с параметризованной вероятностью неудачи ϵ , удовлетворяющий постусловию ψ . (Это означает, что алгоритм должен всегда завершать работу и выдавать правильный ответ с вероятностью $1 - \epsilon$).

Упражнение 9.3. Постройте детерминированный централизованный алгоритм выработки имен, которому требуется совершить $2|E|$ обменов сообщениями за $O(D)$ единиц времени.

Упражнение 9.4 (см. [8]). Постройте детерминированный алгоритм избрания лидера в клике, если обмен сообщениями — синхронный.

Упражнение 9.5. Постройте детерминированные алгоритмы вычисления МАХ для колец и произвольных сетей заранее неизвестного размера, которые оканчивали бы работу по завершении обмена сообщениями.

Упражнение 9.6. Рассмотрим задачу о выборах для анонимных деревьев заранее неизвестного размера при том условии, что обмен сообщениями асинхронный.

1. Постройте рандомизованный частично корректный алгоритм, который оканчивал бы вычисления по завершении работы процессов с вероятностью 1 и для которого ожидаемая сложность по числу обменов сообщениями составляла бы величину $O(N)$. (На самом деле можно добиться того, чтобы ожидаемая сложность по числу обменов сообщениями была равна $N+1$.)

2. Можно ли построить для этого случая детерминированный алгоритм?

9.5.2

Упражнение 9.7. Докажите, что не существует детерминированного алгоритма подсчета количества процессов, который работал бы корректно для всех анонимных сетей, диаметр которых не превосходит 2.

Идея: воспользуйтесь симметричностью сетей, представленных на рис. 6.20.

Упражнение 9.8. Докажите, что не существует детерминированного алгоритма избрания лидера в кольцах заранее известного четного размера при том условии, что обмен сообщениями — синхронный.

Обобщите доказательство на случай, когда размер сети задается составным числом.

Упражнение 9.9. Пусть задана функция f на целочисленных конечных последовательностях $x = (x_1, \dots, x_k)$: Значение $f(x)$ считается равным *true*, если все числа x_i попарно равны; в противном случае значение $f(x)$ равно *false*. Можно ли вычислить f при помощи детерминированного алгоритма, оканчивающего вычисления по завершении работы процессов, на кольцах заранее неизвестного размера? Можно ли вычислить f при помощи детерминированного алгоритма, оканчивающего вычисления по завершении обмена сообщениями, на кольцах заранее неизвестного размера?

Упражнение 9.10. Тот же самый вопрос, что и в упражнении 9.9, но для случая, когда $g(x)$ принимает значение *true*, если x имеет неубывающий циклический сдвиг, т. е. существует такой циклический сдвиг z последовательности x , что $i < j \implies z_i \leq z_j$.

9.5.3

Упражнение 9.11. Постройте алгоритм типа Монте-Карло, оканчивающий вычисления по завершении работы процессов, для избрания лидера в анонимных кольцах заранее неизвестного размера. Какова сложность построенного алгоритма по числу обменов сообщениями и какова вероятность его успешного завершения?

Можно ли сделать вероятность успеха сколь угодно близкой к единице?

Сеть, процессы которой имеют отличительные признаки, но при этом некоторые из этих признаков могут оказаться одинаковыми, называется *псевдоанонимной сетью*. Псевдоанонимное кольцо (состоящее из процессов, начиная с p_0 и оканчивая p_{N-1}) называется *периодическим*, если существует такое число $k < N$, что для всех i выполняется равенство $id_i = id_{i+k}$.

Упражнение 9.12. Покажите, что существует детерминированный алгоритм избрания лидера, оканчивающий работу по завершении обменов сообщениями, в непериодическом псевдо-анонимном кольце заранее известного размера.

Упражнение 9.13. Покажите, что не существует вероятностного алгоритма избрания лидера, оканчивающего вычисления по завершении работы процессов, в непериодическом псевдоанонимном кольце заранее неизвестного размера, который выдавал бы правильные результаты с положительной вероятностью.

9.5.4

Упражнение 9.14. Пусть функция g определена так же, как в упражнении 9.10. Постройте вероятностный алгоритм избрания лидера, оканчивающий работу по завершении обменов сообщениями и правильно вычисляющий g в кольце заранее неизвестного размера с вероятностью $1 - \epsilon$.

Упражнение 9.15. Можно ли обнаружить завершаемость алгоритма Ай-тай—Роде (алгоритм 9.5) при помощи алгоритма Сафры обнаружения завершения (алгоритм 8.7)?

Можете ли вы предложить для этой цели другой алгоритм обнаружения завершения из числа тех, которые обсуждались ранее?

Г Л А В А 10

МОМЕНТАЛЬНЫЕ СОСТОЯНИЯ СИСТЕМЫ

Рассмотренные нами до сих пор алгоритмы (за исключением тех, что изучались в гл. 8) выполняют задачи, связанные с внутренним устройством сети или с глобальным функционированием сети. В этой главе мы обсудим алгоритмы, задача которых состоит в анализе свойств *вычислений*, порождаемых, как правило, другими алгоритмами. Однако обнаруживается, что следить за вычислениями распределенной системы *изнутри* самой системы — это невероятно трудная задача. При проектировании алгоритмов, работающих с системными вычислениями, важным конструктивным блоком служит процедура расчета и сохранения в памяти отдельной конфигурации этого вычисления, так называемого *моментального состояния системы*.

Согласно определению 2.22, распределенное вычисление описывается различными последовательностями конфигураций, и в этих последовательностях представлены разные совокупности конфигураций. Поэтому совсем неясно, какие из конфигураций рассматриваются в качестве конфигураций того или иного вычисления. Этот вопрос будет поставлен в § 10.5.1, где мы свяжем воедино понятия сечения, согласованного сечения и моментального состояния системы. Эти понятия тесно связаны с понятием причинно-следственной зависимости, введенным в § 2.

Построение моментальных состояний системы объясняется их использованием в целом ряде приложений, и здесь мы рассмотрим три из них.

Во-первых, свойства вычисления, в той мере, в какой они представлены в отдельной конфигурации, можно анализировать в режиме *off-line*, т. е. при помощи алгоритма, который исследует (фиксированное) моментальное состояние всей системы, а не (изменяемые) текущие состояния процессов. К числу таких свойств относятся *устойчивые свойства*; свойство P конфигураций называется устойчивым, если справедливо соотношение

$$P(\gamma) \wedge \gamma \rightsquigarrow \delta \implies P(\delta).$$

Иными словами, если вычисление достигает конфигурации γ , обладающей свойством P , то, начиная с этого момента, свойством P будет обладать каждая последующая конфигурация δ . Следовательно, если будет обнаружено, что свойство P выполняется для моментального состояния некоторой конфигурации системы, то отсюда можно сделать вывод о выполнимости P для всех последующих конфигураций. К устойчивым свойствам относятся завершенность вычисления, блокировка (попадание в тупиковую ситуацию), потеря маркера, а также недоступность объектов в структурах динамической памяти. В § 10.4 мы применим

моментальные состояния системы для решения проблемы обнаружения тупиковых ситуаций.

Во-вторых, моментальные состояния системы можно использовать вместо начальных конфигураций, если требуется перезапустить вычисление ввиду отказа какого-либо процесса. Для этого процесс p следует переустановить в то локальное состояние c_p , которое было запечатлено в моментальном состоянии системы, после чего работа алгоритма будет продолжена.

В третьих, моментальные состояния системы являются полезным средством отладки распределенных программ. Проведя в режиме off-line анализ конфигурации, «выхваченной» из ошибочного выполнения, можно обнаружить причину, по которой программа ведет себя не так, как этого ожидают.

10.1. Начальные сведения

Рассмотрим вычисление S распределенной системы, состоящей из некоторого множества процессов $\mathbb{P}$; обозначим символом Ev множество событий этого вычисления. Мы будем исходить из допущения слабой справедливости, которое заключается в том, что каждое сообщение будет доставлено по назначению спустя конечный промежуток времени. Мы будем предполагать также, что сеть является сильно связной. Локальное вычисление процесса p представляет собой последовательность $c_p^{(0)}, c_p^{(1)}, \dots$ состояний процесса, где $c_p^{(0)}$ — это начальное состояние процесса p . Переход из состояния $c_p^{(i-1)}$ в состояние $c_p^{(i)}$ обусловлен осуществлением события $e_p^{(i)}$ в процессе p (см. рис. 10.1). Таким образом, $Ev = \cup_{p \in \mathbb{P}} \{e_p^{(1)}, e_p^{(2)}, \dots\}$.

На множестве событий процесса p введем *причинно-следственный порядок* при помощи соотношения

$$e_p^{(i)} \preceq_p e_p^{(j)} \iff i \leq j.$$

Каждое событие может быть событием *отправления сообщения*, *приема сообщения* или *внутренним событием* (соответствующие определения представлены в гл. 2). Для упрощения описания алгоритмов и формулировки теорем мы будем всегда придерживаться допущения о том, что вся история обмена информацией с процессом определяется его состоянием. Это означает, что если существует канал связи, ведущий из p в q , то состояние $c_p^{(i)}$ процесса p включает в себя список $sent_{pq}^{(i)}$ сообщений, которые процесс p отправил процессу q при осуществлении событий начиная с $e_p^{(1)}$ и оканчивая $e_p^{(i)}$. Кроме того, состояние $c_q^{(i)}$ процесса q содержит список $rcvd_{pq}^{(i)}$ сообщений, которые процесс q получил от процесса p при осуществлении событий, начиная с $e_q^{(1)}$ и оканчивая $e_q^{(i)}$. В § 10.3.1 будет показано, как можно устранить это допущение, чтобы избежать явного хранения в памяти истории обмена информацией в приложениях наших алгоритмов.

Алгоритм построения моментального состояния системы предназначен для реконструкции глобальной конфигурации, которая образуется из локальных состояний (моментальных состояний) всех процессов. Процесс p запечатлевает свое *локальное состояние*, записывая в память одно локальное состояние c_p^* ,

которое называется *моментальным локальным состоянием процесса p* . Если моментальным локальным состоянием процесса является $c_p^{(i)}$, т. е. p проводит моментальный снимок состояния в промежутке между осуществлением событий $e_p^{(i)}$ и $e_p^{(i+1)}$, то события $e_p^{(j)}$, для которых $j \leq i$, называются *предмоментальными событиями* процесса p , а те события, для которых $j > i$, называются *постмоментальными событиями* процесса p . На временных диаграммах запечатление локального состояния процесса будет отмечаться незакрашенным кружком того состояния процесса, которое стало его моментальным состоянием (см. рис 10.1 и 10.2).

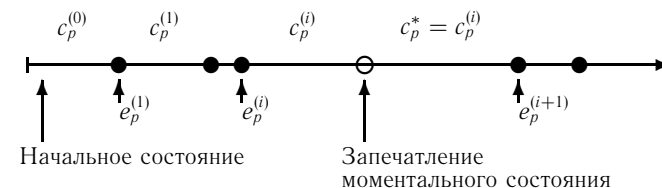


Рис. 10.1. Вычисление процесса p

Глобальное моментальное состояние S^* образуется из моментальных состояний c_p^* всех процессов p , входящих в множество $\mathbb{P}$. Для обозначения этого мы будем использовать запись $S^* = (c_{p_1}^*, \dots, c_{p_n}^*)$. Так как локальные состояния включают в себя истории обменов информацией, моментальное состояние S^* определяет конфигурацию γ^* , если состоянием канала pq считать множество сообщений, отправленных¹⁾ процессом p (согласно состоянию c_p^*), но еще не доставленных процессу q (согласно состоянию c_q^*). Иными словами, состояние канала pq в моментальном состоянии S^* определяется множеством сообщений $sent_{pq}^* \setminus rcvd_{pq}^*$. Конфигурацию, состоящую из моментальных состояний процессов и определенных таким образом состояний каналов, обозначим символом γ^* .

При построении указанной конфигурации появляются некоторые отклонения, если множество $rcvd_{pq}^*$ не является подмножеством $sent_{pq}^*$ (см. рис 10.2). Судя по локальному состоянию $c_{p_1}^*$ в формируемом моментальном состоянии системы, сообщение было отправлено процессом p_1 процессу p_3 , но, как отмечено в локальном состоянии $c_{p_3}^*$, никакого сообщения от p_1 не было получено. Таким образом, в данном моментальном состоянии системы канал p_1p_3 содержит одно сообщение; говорят, что в этом моментальном состоянии системы указанное сообщение пребывает «на этапе пересылки».

Обратимся теперь к сообщению, которое процесс p_1 отправил процессу p_2 . Отправление этого сообщения относится к числу постмоментальных событий, тогда как его получение является предмоментальным событием. Значит, в соответствии с состоянием $c_{p_1}^*$ никакие сообщения по каналу p_1p_2 не отправлялись, тогда как в состоянии $c_{p_2}^*$ отмечено, что по этому каналу было получено некоторое сообщение. Поскольку $rcvd_{p_1p_2}^* \not\subseteq sent_{p_1p_2}^*$, никакого осмысленного состояния

¹⁾По этому каналу. — Прим. перев.

каналу p_1p_2 придать нельзя. Это побуждает нас обратиться к следующему определению.

Определение 10.1. Моментальное состояние S^* называется *осуществимым*, если для всякой пары соседних процессов p и q выполняется включение $rcvd_{pq}^* \subseteq sent_{pq}^*$.

Осуществимость моментального состояния означает, что при построении соответствующей конфигурации в $rcvd_{pq}^*$ не останется ни одного сообщения, которое нельзя удалить из $sent_{pq}^*$. Сообщение, отправленное которого является предмоментальным (постмоментальным) событием, будем называть *предмоментальным* (соответственно, *постмоментальным*) сообщением.

Существует взаимнооднозначное соответствие между моментальными состояниями и конечными *сечениями* на совокупности событий вычисления. Сечением называется совокупность событий, замкнутая влево относительно отношения локальной причинно-следственной зависимости.

Определение 10.2. Сечением на множестве $E\mathcal{V}$ называется такое подмножество $L \subseteq E\mathcal{V}$, для которого выполняется соотношение

$$e \in L \wedge e' \preceq_p e \implies e' \in L.$$

Сечение L_2 считается *более поздним*, чем сечение L_1 , если верно включение $L_1 \subseteq L_2$.

С одной стороны, нетрудно видеть, что для каждого моментального состояния S^* совокупность L всех пред-моментальных событий является конечным сечением. Рассмотрим теперь произвольное конечное сечение L . Для каждого процесса p либо ни одно событие, произошедшее в p , не входит в L (в таком случае будем полагать $m_p = 0$), либо в L существует наибольшее событие $e_p^{(m_p)}$ и при этом все события $e \preceq_p e_p^{(m_p)}$ также содержатся в L . Поэтому L представляет собой в точности множество предмоментальных событий моментального состояния $S^* = (c_{p_1}^{(m_{p_1})}, \dots, c_{p_N}^{(m_{p_N})})$.

Моментальное состояние будет использоваться для извлечения информации о том вычислении, которое оно запечатлевает, но произвольно выбранное моментальное состояние системы дает очень мало информации о таком вычислении. Рассмотрим, например, алгоритм, в основу которого положена передача уникального маркера; к таковым относятся алгоритмы обхода из § 6.6.3. Допустим, что процесс p запечатлел свое состояние, когда он обладал маркером, и спустя некоторое время процесс q также запечатлел свое состояние, когда он, в свою очередь, обладал маркером. В построенной конфигурации два процесса обладают маркером, хотя подобной ситуации не может возникнуть ни в одном вычислении нашего алгоритма.

Естественно, мы бы предпочли, чтобы алгоритм построения моментального состояния реконструировал конфигурацию, которая и «в самом деле» может возникнуть при вычислении. К сожалению, множество реализуемых конфигураций не является инвариантом отношения эквивалентности, как это показано в § 2.3.2,

и поэтому оно не определяется вычислением однозначно. Таким образом, мы будем считать разумным всякий результат работы алгоритма, если он приводит к построению какой-либо конфигурации, которая окажется *возможной* для заданного вычисления (т. е. случится в некоторой реализации этого вычисления).

Определение 10.3. Моментальное состояние S^* называется *значимым* в вычислении C , если существует такая реализация $E \in C$, что γ^* является конфигурацией E .

Мы будем требовать от алгоритма построения моментального состояния такой координации регистрации локальных моментальных состояний, чтобы полученное в результате этого глобальное моментальное состояние оказалось значимым. Своевременность запечатления моментальных состояний обсуждается в § 10.3.2.

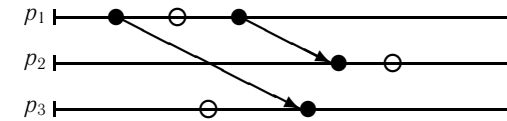


Рис. 10.2. Несогласованное моментальное состояние

Требование осуществимости моментального состояния налагает условия на отношения только между локальными моментальными состояниями соседних процессов, в то время как значимость является глобальным свойством моментального состояния всей системы. Однако далее мы покажем (теорема 10.5), что осуществимость моментального состояния равносильна его значимости, а также что такие моментальные состояния соответствуют *согласованным* сечениям. Сечение называется согласованным, если оно замкнуто влево относительно отношения причинно-следственной зависимости²⁾.

Определение 10.4. *Согласованным сечением* на множестве событий $E\mathcal{V}$ называется такое подмножество $L \subseteq E\mathcal{V}$, для которого выполняется соотношение

$$e \in L \wedge e' \preceq e \implies e' \in L.$$

Теорема 10.5. Пусть S^* — моментальное состояние системы и L — сечение, порожденное S^* . Тогда равносильны следующие три утверждения:

- 1) S^* осуществимо;
- 2) L согласовано;
- 3) S^* значимо.

Доказательство. Мы покажем, что из 1) следует 2), из 2) следует 3), и из 3) следует 1).

²⁾Глобальной — Прим. перев.

```
var takenp : bool  init false ;
```

Для запуска алгоритма:

```
begin запомнить локальное состояние ; takenp := true ;
  forall q ∈ Neighp do send ⟨mkr⟩ to q
end
```

Если был доставлен маркер:

```
begin receive ⟨mkr⟩ ;
  if not takenp then
    begin запомнить локальное состояние ; takenp := true ;
      forall q ∈ Neighp do send ⟨mkr⟩ to q
    end
  end
end
```

Алгоритм 10.4. Алгоритм Ченди—Лэмпорта построения моментального состояния

Лемма 10.6. *Если хотя бы один процесс запустит алгоритм 10.4, то спустя конечный промежуток времени все процессы запечатлеют свое локальное моментальное состояние.*

Доказательство. Поскольку каждый процесс запечатлевает моментальное состояние и отправляет сообщение ⟨mkr⟩ не более одного раза, работа алгоритма построения моментального состояния системы рано или поздно завершится. Если процесс p к тому времени уже запечатлел свое моментальное состояние и процесс q является соседом p , то процесс q также уже успел запечатлеть свое локальное состояние. Это происходит потому, что сообщение ⟨mkr⟩, отправленное процессом p , было получено процессом q и вынудило его запечатлеть моментальное состояние, если только он не сделал этого ранее. Так как по-меньшей мере один процесс запустил алгоритм, хотя бы один процесс запечатлел свое локальное состояние; связность сети приводит к тому, что это успевают сделать все процессы. $\square$

Заметим, что алгоритм должен быть инициирован *хотя бы* одним процессом, но он будет работать правильно и в том случае, если он был запущен произвольным непустым множеством процессов.

Теорема 10.7. *Алгоритм Ченди—Лэмпорта (алгоритм 10.4) Вычисляет значимое моментальное состояние системы спустя конечный промежуток времени, после того как он был запущен хотя бы одним процессом.*

Доказательство. Согласно предыдущей лемме алгоритму требуется конечный промежуток времени для вычисления моментального состояния системы. Остается показать, что построенное моментальное состояние системы является осуществимым, т.е. получение каждого постмоментального (базового) сообщения составляет постмоментальное событие. Рассмотрим постмоментальное сообщение m , отправленное процессом p процессу q . Перед отправлением сообщения m , процесс p запечатлел свое моментальное локальное состояние и отпра-

вил сообщение ⟨mkr⟩ всем своим соседям, включая процесс q . Так как в каналах поддерживается очередность сообщений, процесс q получил сообщение ⟨mkr⟩, до того как получил m , и в соответствии с приведенным алгоритмом процесс q запечатлел свое моментальное состояние сразу по получении этого сообщения или еще ранее того. Следовательно, получение сообщения m относится к числу постмоментальных событий. $\square$

В алгоритме Ченди—Лэмпорта требуется совершить обмен $2|E|$ сообщениями ⟨mkr⟩ и использовать один дополнительный бит памяти (и, конечно же, к этому следует добавить память для хранения моментального состояния) в каждом процессе. Сложность по времени рассмотренного алгоритма составляет величину $O(D)$.

10.2.2. Алгоритм Лая—Янга

Алгоритм Лая—Янга (см. [122]) не опирается на свойство поддержания очередности сообщений в каналах. Поэтому отделить предмоментальные события от постмоментальных при помощи маркеров, как это сделано в алгоритме Ченди—Лэмпорта, уже невозможно. Вместо этого каждое базовое сообщение снабжается пометкой, которая позволяет понять, является ли это сообщение предмоментальным или постмоментальным; для этого процесс p , отправляя сообщение по ходу вычисления C , добавляет к нему значение $taken_p$. Так как содержание сообщений, относящихся к вычислению C , не играет здесь никакой роли, мы будем обозначать такие сообщения просто ⟨mes, c ⟩, где c — это значение переменной, включенное в состав сообщения процессом-отправителем. Алгоритм построения моментального состояния проверяет поступающие сообщения и записывает моментальное локальное состояние в том виде, в каком оно было до поступления первого постмоментального сообщения (см. алгоритм 10.5).

```
var takenp : bool  init false ;
```

Для запуска алгоритма:

```
begin запомнить локальное состояние ; taken := true end
```

Для отправления сообщения по ходу вычисления C :

```
send ⟨mes, takenp⟩
```

Если поступило сообщение ⟨mes, c ⟩;

```
begin receive ⟨mes, c⟩ ;
  if c and not takenp then
    begin запомнить локальное состояние ; taken := true end;
    изменить состояние на то, в котором пребывал процесс
    при получении сообщения в вычислении C
  end
end
```

Алгоритм 10.5. Алгоритм Лая—Янга построения моментального состояния

В алгоритме 10.5 нет обмена контрольными сообщениями, но он не может гарантировать того, что каждый процесс рано или поздно запишет свое состояние; на самом деле бывает и так, что этой записи не происходит. Рассмотрим процесс p , не являющийся инициатором алгоритма построения моментального состояния, и предположим, что ни один из соседей процесса p не отправляет процессу p сообщений, после того как этот сосед запечатлел свое моментальное локальное состояние. В таком случае p никогда не проведет запись своего состояния, и алгоритм построения моментального состояния завершит работу, имея неполное моментальное состояние.

Решение этой проблемы зависит от того, что нам известно о вычислении S ; если мы имеем гарантию, что связь с каждым процессом рано или поздно осуществится, то и полное моментальное состояние всегда можно будет запечатлеть. В противном случае алгоритм можно снабдить блоком, проводящим первоначальный обмен контрольными сообщениями между всеми процессами, как и в алгоритме 10.4. Эти сообщения всего лишь гарантируют, что каждый процесс рано или поздно запишет свое состояние (как это установлено в лемме 10.6), но они не понадобятся при доказательстве осуществимости построенного моментального состояния системы. Так или иначе, вычисление полного моментального состояния системы обеспечено.

Теорема 10.8. *Алгоритм Лая—Янга (алгоритм 10.5) вычисляет только значимые моментальные состояния системы.*

Доказательство. Рассмотрим моментальное состояние, вычисленное алгоритмом, и пусть $m = \langle \text{mes}, c \rangle$ — это постмоментальное сообщение, отправленное процессом p процессу q . Это значит, что $c = \text{true}$, и поэтому q запечатлевает свое состояние таким, каким оно было до получения сообщения m . Таким образом, в состоянии, которое записал процесс q , не учитывается поступление m , и прием сообщения m относится к числу постмоментальных событий. (Следует напомнить, что существенным здесь является то, какое состояние записать, а не то, когда провести эту запись; в нашем случае запись состояния проводится одновременно с получением первого постмоментального события.) $\square$

Использование подавления (замораживания). В алгоритме Ченди—Лэмпорта (алгоритм 10.4) маркеры используются для отделения предмоментальных сообщений от постмоментальных, и поэтому в каналах требуется соблюдение очередности сообщений. В алгоритме Лая—Янга (алгоритм 10.5) в базовые сообщения явно вставляется пометка о том, является ли сообщение предмоментальным или постмоментальным, и, следовательно, здесь процессам должна быть предоставлена возможность реализовать принцип «оседлай поросенка» по отношению к информации, содержащейся в базовом сообщении. Тейлор в работе [186] показал, что если в канале не поддерживается очередность сообщений и «оседлать поросенка» невозможно, всякое решение проблемы построения моментального состояния системы обязано быть *ингибиторным*, т. е. получено за счет временного приостановления базового вычисления. Классификация разных типов подавления и описание необходимых средств подавления при различ-

ных допущениях относительно коммуникационной среды представлены в работе Критчлоу и Тейлора [57].

10.3. Применение алгоритмов построения моментальных состояний

10.3.1. Вычисление состояния канала

До сих пор предполагалось, что состояние каждого процесса включает историю обмена информацией процесса и поэтому состояние всякого канала можно вычислить на основе состояний примыкающих к нему процессов. В большинстве случаев, однако, хранить в явном виде все сообщения, которые когда-либо отправлял или получал процесс, слишком накладно. Мы покажем здесь, как можно восстанавливать состояние канала более эффективно.

Редукция к значимой информации. В зависимости от предназначения моментального состояния системы может оказаться достаточным хранить только ограниченный объем значимых сведений об истории обмена информацией. Например, мы можем вознамериться построить моментальное состояние алгоритма обмена маркерами (наподобие алгоритма Корача—Каттена—Морана), чтобы определить, сколько маркеров все еще являются активными. Чтобы определить количество маркеров в каждом канале, каждому процессу достаточно запомнить, сколько маркеров он отправил и сколько маркеров он получил. Вторым примером может служить моментальное состояние системы для проверки завершенности вычисления (см. гл. 8). Для этого нужно всего лишь помнить о том, пуст канал или нет, а для этого достаточно уметь считать сообщения, а не хранить их в явном виде. (В обоих примерах при записи состояния каждого процесса можно также ограничиться небольшим объемом значимой информации, а именно числом маркеров, которыми обладает процесс, или, соответственно, указанием того, является ли процесс активным или пассивным.)

Явное восстановление состояния канала. Состояние канала можно восстановить в явном виде, опираясь на следующую лемму.

Лемма 10.9. *Если моментальное состояние системы является осуществимым, то $\text{sent}_{pq}^* \setminus \text{rcvd}_{pq}^*$ образует в точности множество тех сообщений, отправленные которых процессом p является предмоментальным событием, а получение этих сообщений процессом q — постмоментальным событием.*

Доказательство. С одной стороны, нетрудно видеть, что сообщение $m \in \text{sent}_{pq}^* \setminus \text{rcvd}_{pq}^*$ отправлено предмоментально, а получено постмоментально. А с другой стороны, если сообщение m отправлено предмоментально и получено постмоментально, то оно по определению включается в sent_{pq}^* , но не включается в rcvd_{pq}^* . Следовательно, такое сообщение принадлежит $\text{sent}_{pq}^* \setminus \text{rcvd}_{pq}^*$. $\square$

Процесс q восстанавливает состояние канала pq путем записи всех предмоментальных сообщений, которые были получены им постмоментально. Коль

скоро все такие сообщения получены процессом q после запечатления его моментального состояния, q приступает к записи сообщений, после того как он записал свое состояние, и запись прекращается, как только все предмоментальные сообщения будут получены.

Алгоритм Ченди—Лэмпорта. Все предмоментальные сообщения от p к q получены перед тем, как сообщение $\langle \mathbf{mkr} \rangle$ было отправлено процессом p процессу q ; даже, более того, *только* предмоментальные сообщения были получены до поступления маркера. Восстановить состояния канала в таком случае будет чрезвычайно просто: состояние канала pq представляет собой совокупность сообщений, полученных процессом q , после того как он сделал запись своего состояния, но до того, как он получил сообщение $\langle \mathbf{mkr} \rangle$ от процесса p .

Алгоритм Лая—Янга. Если в канале не поддерживается очередность сообщений, процесс q может получать предмоментальные и постмоментальные сообщения поочередно; получение постмоментального сообщения не означает, что все предмоментальные сообщения уже были доставлены. Поэтому, хотя и ясно, какие сообщения должны быть записаны процессом q (только те, которые помечены ярлыком *false*, являются предмоментальными), непонятно, в какой же момент этот процесс уже завершил восстановление полного состояния канала и может прекратить запись.

Решение этой проблемы, предложенное Маттерном в работе [144], состоит в том, чтобы заставить процесс p вести подсчет общего числа предмоментальных сообщений, отправленных им процессу q , и уведомлять q об этом числе (либо в отдельном сообщении, либо присоединяя это число к какому-либо постмоментальному сообщению по принципу «оседлать поросенка»). Процесс q подсчитывает число полученных им предмоментальных сообщений (получение которых было связано как с предмоментальными, так и с постмоментальными событиями) и завершает работу по восстановлению состояния канала, как только будет получено достаточно большое число предмоментальных сообщений.

10.3.2. Своевременность моментальных состояний системы

Можно гарантировать, что моментальные состояния системы, которые строятся при помощи алгоритмов, рассмотренных в этой главе, являются значимыми в тех вычислениях, для которых они были построены. Но помимо этого нам хотелось бы, чтобы моментальное состояние было «свежим» в том смысле, что извлеченная из такого состояния информация не должна быть слишком устаревшей. Коль скоро для оценки своевременности информации необходимо понятие времени, мы будем привязывать моментальное состояние системы к конфигурациям, которые возникают при распределенном *выполнении*. Во всяком выполнении вычисленное моментальное состояние системы располагается между той конфигурацией, в которой началась запись состояния, и той, в которой эта запись завершилась, о чем свидетельствует следующая теорема.

Теорема 10.10. *Предположим, что имеется выполнение E , осуществимое моментальное состояние системы S^* , запечатленное по ходу этого выполнения, и соответствующая ему конфигурация системы γ^* . Пусть γ_s — это конфигурация, в которой первый процесс записал свое состояние, а γ_e — это конфигурация, в которой последний процесс записал свое состояние. Тогда справедливо соотношение $\gamma_s \rightsquigarrow \gamma^* \rightsquigarrow \gamma_e$.*

Доказательство. Усилим аргументацию, используемую при доказательстве теоремы 10.5. Построим последовательность событий f , сперва перечислив предмоментальные события выполнения E в порядке их осуществления в E , а затем занумеруем все постмоментальные события выполнения E в порядке их осуществления в E . Такая нумерация не противоречит причинно-следственному порядку в E и поэтому определяет некоторое выполнение F .

Выполнение F содержит конфигурации γ_s , γ^* , и γ_e в указанном порядке, и отсюда следует утверждение теоремы. Выполнение F содержит γ_s , поскольку все события из E , предшествующие γ_s , являются предмоментальными событиями, и, следовательно, f начинается именно с этих событий и именно в том же порядке. Выполнение F содержит γ_e , поскольку совокупность событий, предшествующих γ_e в выполнении E , содержит все предмоментальные события. Значит, f начинается именно этими событиями (хотя, возможно, они следуют в другом порядке), и F содержит γ_e на основании теоремы 2.21. $\square$

10.3.3. Обнаружение устойчивых свойств

Рассмотрим устойчивое свойство P конфигураций; как только выполнение достигает конфигурации, для которой выполняется P , это свойство будет продолжаться выполняться и для каждой последующей конфигурации. К устойчивым относятся следующие свойства:

1. **Завершение вычисления.** Если γ — это заключительная конфигурация и $\gamma \rightsquigarrow \delta$, то $\gamma = \delta$, и поэтому конфигурация δ также является заключительной. Следовательно, задача обнаружения завершения (гл. 8) может быть решена за счет вычисления моментального состояния системы и анализа активных процессов и базовых сообщений в этом моментальном состоянии. Однако специальные решения, предложенные в гл. 8 обычно более эффективны.

2. **Наличие тупиков.** Если в конфигурации γ некоторое множество процессов S оказывается заблокированными, из-за того что все процессы из S пребывают в ожидании³⁾ других процессов из S , то такая же ситуация повторится и в последующих конфигурациях, хотя процессы вне множества S могут изменять свои состояния. Задача обнаружения тупиков рассматривается в § 10.4.

3. **Потеря маркеров.** Рассмотрим алгоритм, в котором маркеры циркулируют между процессами, а сами процессы могут поглощать маркеры. Свойство вида «Имеется не более k маркеров» является устойчивым, так как маркеры могут поглощаться, но не могут порождаться.

³⁾поступления сообщений от — *Прим. перев.*

4. *Наличие «мусора».* В объектно-ориентированной среде программирования создается совокупность объектов, каждый из которых может содержать *ссылку* на другие объекты. Объект считается *доступным*, если можно отыскать путь, ведущий по ссылкам к данному объекту от некоторого предписанного объекта; в противном случае объект объявляется «мусором». Ссылки могут появляться и исчезать, но ссылки, ведущие к «мусорным» объектам, никогда не возникают. Поэтому, как только объект становится «мусором», он будет во веки вечные оставаться таковым. (Проблеме сборки «мусора» посвящено изрядное количество работ, но в этой книге она не рассматривается. Читатель, интересующийся этой задачей, может обратиться к работе [189] и воспользоваться списком литературы из указанной работы.)

Чтобы выполнить предписанные действия, как только предикат P стал истинным, его истинность нужно обнаружить посредством вспомогательного алгоритма, который следит за вычислением и запускает нужные действия, когда обнаружит истинность предиката P .

```

var detect    : bool    init false ;

while notPholds do
  begin вычислить глобальное моментальное состояние  $\gamma^*$  ;
        вычислить  $Pholds := P(\gamma^*)$ 
  end;
detect := true

```

Алгоритм 10.6. Алгоритм обнаружения устойчивого свойства

Весьма общий механизм обнаружения представлен в алгоритме 10.6. В этом алгоритме требуется проводить проверку выполнимости свойства P для конфигурации, соответствующей моментальному состоянию системы, но обычно это осуществляется при помощи сравнительно простой распределенной процедуры. В то время как задача алгоритма обнаружения состоит в том, чтобы отслеживать динамику поведения распределенного вычисления, такая процедура всего лишь анализирует статические данные (которые представлены в конфигурации). Вычисление процедуры $Pholds$ можно провести либо централизованно, либо распределенно. При централизованном вычислении каждый процесс отправляет свое моментальное состояние выделенному процессу, который восстанавливает полную конфигурацию и вычисляет $Pholds$. При распределенном вычислении все процессы участвуют в таком вычислении.

Алгоритм 10.6 обладает свойствами корректности и полноты, о чем свидетельствуют следующие две теоремы; это как раз те свойства, которые требуются от всякого алгоритма обнаружения. Предположим, что алгоритм 10.6 следит за некоторым распределенным алгоритмом, который порождает выполнение E .

Теорема 10.11 (Корректности). *Если алгоритм 10.6 присваивает значение true переменной detect, когда система пребывает в конфигурации δ , то выполняется $P(\delta)$.*

Доказательство. Пусть γ^* — это последнее из вычисленных моментальных состояний системы, а γ_e — конфигурация выполнения E , в которой было запечатлено последнее локальное состояние конфигурации γ^* . Согласно рассматриваемому алгоритму выполняется $P(\gamma^*)$. По теореме 10.10 имеет место соотношение $\gamma^* \rightsquigarrow \gamma_e$, и поскольку переменная *detect* получила значение *true* после завершения построения γ^* , имеет место соотношение $\gamma_e \rightsquigarrow \delta$. В силу устойчивости свойства P будет верно $P(\delta)$. $\square$

Теорема 10.12 (Полноты). *Если выполнение E достигло конфигурации, в которой свойство P выполняется, то спустя конечный промежуток времени алгоритм 10.6 присвоит переменной detect значение true.*

Доказательство. Предположим, что δ — это первая конфигурация выполнения E , которая обладает свойством P . Рассмотрим конфигурацию γ_s , следующую за δ (или совпадающую с δ), в которой впервые после прохождения δ предпринято вычисление моментального состояния системы алгоритмом 10.6. Пусть γ^* — это моментальное состояние системы, вычисление которого началось в конфигурации γ_s . Так как имеет место соотношение $\delta \rightsquigarrow \gamma_s \rightsquigarrow \gamma^*$, верно $P(\gamma^*)$, и алгоритм обнаружения завершает работу после этого тура. $\square$

Алгоритм 10.6 можно использовать для решения всех задач по обнаружению устойчивых свойств, но при этом следует принять во внимание следующие замечания.

1. Для некоторых свойств P (распределенное) вычисление значения $P(\gamma^*)$ может потребовать применения достаточно сложных алгоритмов. Такое случается для некоторых модельных вариантов задачи обнаружения тупиков (один из алгоритмов обнаружения тупиков в заданной конфигурации приведен в § 10.4).

2. В некоторых случаях этим методом нельзя воспользоваться ввиду ограничений по объему памяти. Такое возможно в случае задачи по сбору «мусора»; сохранение в памяти всех ссылок и сообщений моментального состояния системы более чем вдвое увеличивает объем запрашиваемой памяти, и это может оказаться слишком высокой ценой.

Ранние алгоритмы по сбору «мусора» приостанавливали базовое вычисление при проведении оценки значения предиката, служащего индикатором наличия «мусора»; это решение представляется весьма простым, но обычно оно отвергается, поскольку приостановка базового вычисления представляется недопустимой. Сравнительно недавно были разработаны алгоритмы, работающие «на лету»; вместо того чтобы заниматься отдельной конфигурацией, они взаимодействуют с базовым вычислением и не относятся к категории алгоритмов, применяющих вычисление моментального состояния системы (см. [187, глава 5] и [189]).

3. Даже в тех случаях, когда решения, основанные на применении алгоритма 10.6, просты и эффективны, они могут оказаться не самыми простыми и не наиболее эффективными. Это касается, например, задачи обнаружения завершения вычислений, для которой в гл. 8 были приведены совершенно простые решения.

Моментальные состояния системы можно использовать для вычисления *монотонных функций*, определенных на конфигурациях системы; функция f на-

зывается монотонной, если выполняется соотношение

$$\gamma \rightsquigarrow \delta \implies f(\gamma) \leq f(\delta).$$

Если вычислено моментальное состояние системы γ^* и проведена оценка $f^* = f(\gamma^*)$, то для всех последующих конфигураций δ будет выполняться неравенство $f(\delta) \geq f^*$. Примером монотонной функции может служить глобальное виртуальное время (GVТ) в распределенном дискретном симуляторе событий (см., например, [187, гл. 4] или [49]).

10.4. Приложения: обнаружение тупиков

В некоторых распределенных программах процессы могут время от времени быть заблокированы в ожидании сообщений от других процессов. Примерами могут служить транзакции в базах данных, а также процессы, состоящие за право доступа к ресурсам. Тупиковые ситуации, если таковые случаются и не разрешаются должным образом, таят серьезную опасность для распределенных систем, понижая их производительность и сокращая область применения. Прежде всего, процессы, вовлеченные в тупиковую ситуацию, никогда не смогут завершить выполнение своей задачи, и, кроме того, ресурсы, занятые этими процессами, уже никогда не будут доступны другим процессам. Понятно, что грамотная обработка тупиковых ситуаций чрезвычайно важна для реализации распределенных систем

10.4.1. Модель базового вычисления и постановка задачи

Рассмотрим множество процессов $\mathbb{P}$, выполняющих *базовое вычисление*; мы не будем касаться здесь их предназначения и принципов функционирования. Всякий процесс может быть вынужден приостановить свое локальное вычисление, если необходимо дождаться сообщений от других процессов; это моделируется заменой состояния *active* на состояние *blocked*.

Когда процесс p становится заблокированным (действие $\mathbf{B}_p$ в алгоритме 10.7), он отправляет сообщение-запрос семейству процессов, которое мы будем обозначать символом $Reqs_p$. Чтобы снова стать активным (действие $\mathbf{F}_p$), процесс p должен получить разрешение от каких-либо процессов семейства $Reqs_p$, но не обязательно от всех сразу. Подмножество семейства $Reqs_p$, получения разрешения от которых достаточно, чтобы процесс p стал вновь активным, образуют все те процессы, которые удовлетворяют предикату $Free_p$. Запросы, полученные процессом p (действие $\mathbf{R}_p$), образуют множество $Pend_p$, и процесс p откликнется на них, как только станет активным (действие $\mathbf{G}_p$). К каждому запросу и разрешению прикрепляется номер запроса, проставленный процессом p , который позволяет отбросить разрешения, отправленные в ответ на устаревшие запросы (см. $\mathbf{F}_p$).

Мы покажем на нескольких примерах, как применяется условие $Free_p$. Во-первых, если p требует отклика на все сообщения-запросы, то $Free_p$ будет истинно только на множестве $Reqs_p$. Во-вторых, если процессу p достаточно одного-

```

var  $state_p$  : (active, blocked)   init active ;
 $v_p$        : integer             init 0 ;   (* Номер запроса *)
 $Reqs_p$     : set of processes ;          (* Множество запросов *)
 $Grant_p$    : set of processes ;          (* Полученные разрешения *)
 $Pend_p$     : set of requests         init  $\emptyset$  ; (* Ожидаемые разрешения *)
 $Free_p$     : predicate ;              (* Ограничения освобождения *)

```

```

 $\mathbf{B}_p$ : {  $state_p = active$  }
begin определить  $Reqs_p$  и  $Free_p$  ;  $v_p := v_p + 1$  ;
      forall  $q \in Reqs_p$  do send  $\langle \mathbf{req}, v_p \rangle$  to  $q$  ;
       $Grant_p := \emptyset$  ;  $state_p := blocked$ 
end

 $\mathbf{R}_p$ : { Сообщение  $\langle \mathbf{req}, v \rangle$  доставлено от  $q$  }
begin receive  $\langle \mathbf{req}, v \rangle$  from  $q$  ;  $Pend_p := Pend_p \cup \{(q, v)\}$  end

 $\mathbf{G}_p$ : {  $state_p = active \wedge (q, v) \in Pend_p$  }
begin send  $\langle \mathbf{grant}, v \rangle$  to  $q$  ;  $Pend_p := Pend_p \setminus \{(q, v)\}$  end

 $\mathbf{F}_p$ : { Сообщение  $\langle \mathbf{grant}, v \rangle$  доставлено от  $q$  }
begin receive  $\langle \mathbf{grant}, v \rangle$  from  $q$  ;
      if  $state_p = blocked$  and  $v = v_p$  then
        begin  $Grant_p := Grant_p \cup \{q\}$  ;
          if  $Free_p(Grant_p)$  then  $state_p := active$ 
        end
      end
end

```

Алгоритм 10.7. Базовый алгоритм

единственного разрешения, то $Free_p$ будет истинно на всяком непустом множестве. И, наконец, в-третьих, предположим, что процессу p необходимо 1000 мегабайт свободной памяти и он отправляет запрос дискам A (с 900 мегабайтами свободной памяти), B (с 600 мегабайтами свободной памяти), C (с 300 мегабайтами свободной памяти) и D (с 100 мегабайтами свободной памяти). Если процессу доступен диск A , а также любой другой диск, то p может продолжить вычисление. Процесс p может продолжить вычисление и в том случае, когда доступны диски B , C и D . Такой предикат $Free_p(S)$ можно выразить формулой

$$(\#S \geq 2 \wedge A \in S) \vee (\#S \geq 3).$$

Один и тот же процесс p может использовать разные варианты предиката $Free_p$, освобождающие процесс от разных блокировок, но всякий раз при этом должны быть соблюдены следующие условия.

D1. Утверждение $Free_p(\emptyset)$ всегда ложно, а $Free_p(Reqs_p)$ всегда истинно.

D2. Если выполняется утверждение $Free_p(A)$ и $A \subseteq B$, то выполняется и утверждение $Free_p(B)$.

Согласно условию D1 процесс p не может стать активным, не получив никакого разрешения, и обязательно становится активным при получении разрешения от всех процессов из множества $Reqs_p$. Согласно условию D2 процесс p имеет право стать активным, если полученный им набор разрешений включает в себя какой-либо минимально необходимый набор разрешений.

Пусть задан базовый алгоритм; тогда мы можем дать следующее определение таким понятиям, как мертвый процесс и тупиковая конфигурация.

Определение 10.13. Процесс p называется *действующим* в конфигурации γ , если существует такая конфигурация δ , достижимая из γ , в которой процесс p активен; процесс p считается *мертвым*, если он не является действующим. Конфигурация γ называется *тупиковой конфигурацией*, если она содержит мертвый процесс.

Свойство конфигурации быть тупиковой — это *устойчивое* свойство; если процесс p стал мертвым в конфигурации γ_1 и $\gamma_1 \rightsquigarrow \gamma_2$, то он остается мертвым и в конфигурации γ_2 . Последовательность событий базового вычисления, в результате осуществления которой система переходит из конфигурации γ в конфигурацию δ , называется *продолжением* базового вычисления из γ в δ .

Задача обнаружения тупиков состоит в разработке контрольного алгоритма, который можно было бы совместить с базовым вычислением так, чтобы при этом соблюдались следующие три условия.

1. *Невмешательство*. Контрольный алгоритм не оказывает влияния на базовое вычисление.
2. *Полнота*. Если образуется тупиковая ситуация, то она будет обнаружена контрольным алгоритмом.
3. *Корректность*. Алгоритм обнаруживает тупиковую ситуацию только тогда, когда она действительно сложилась.

По ходу работы контрольному алгоритму может понадобиться провести обмен дополнительными сообщениями — мы будем называть их *контрольными сообщениями*, — которые могут быть отправлены как активными, так и заблокированными процессами. Помимо доклада о возникновении тупиковой ситуации, от контрольного алгоритма обычно требуется также указать один или несколько мертвых процессов, используя которые можно было бы устранить этот тупик.

10.4.2. Алгоритм глобальной разметки

В этом параграфе мы предлагаем алгоритм, который называется *алгоритмом глобальной разметки*, для обнаружения тупиков и выявления всех мертвых процессов. Этот алгоритм постоянно вычисляет моментальные состояния системы и применяет к соответствующим им конфигурациям процедуру 10.8. В этой процедуре значения $state_p^*$, v_p^* , $Reqs_p^*$, $Grant_p^*$, $Pend_p^*$, $Free_p^*$ и $(sent_{qp}^* \setminus rcvd_{qp}^*)$ считаются заданными константами (они вычисляются алгоритмом построения моментального состояния). В алгоритме 10.8 для выявления действующих процессов моделируется обмен разрешениями и освобождения процессов в продолжении вычисления из конфигурации γ^* .

Переменной $alive_p$ первоначально присвоено значение $false$, но, как только процесс p обнаружит, что он является действующим в рассматриваемом моментальном состоянии, это значение будет изменено на $true$. Это происходит в том случае, когда p пребывает в активном состоянии в данном моментальном состоянии системы, а также если множество сообщений $\langle grant, v_p^* \rangle$ и $\langle Alive \rangle$, полученных к этому моменту, удовлетворяют предикату $Free_p^*$. Переменная $GrRec_p$ сохраняет множество разрешений (для текущего запроса), которые либо уже были получены, либо находятся на этапе пересылки в заданном моментальном состоянии системы, а также сообщения $\langle Alive \rangle$, которые были получены процессом p .

```

var alive : bool    init false ;
GrRec    : set of  $\mathbb{P}$   init  $Grant_p^* \cup \{q : \langle grant, v_p \rangle \in sent_{qp}^* \setminus rcvd_{qp}^*\}$  ;

```

```

Mp(* Начать разметку в активном процессе *)
{  $\neg alive_p \wedge (state_p^* = active \vee Free_p^*(GrRec_p))$  }
begin alivep := true ;
  forall  $q \in Pend_p^* \cup \{q : \langle req, v \rangle \in (sent_{qp}^* \setminus rcvd_{qp}^*)\}$ 
  do send  $\langle Alive \rangle$  to  $q$ 
end

```

```

Pp(* Распространить разметку на освобожденные процессы *)
{ An  $\langle Alive \rangle$  message arrives from  $q$  }
begin receive  $\langle Alive \rangle$  from  $q$  ;  $GrRec_p := GrRec_p \cup \{q\}$  ;
  if not alivep and  $Free_p^*(GrRec_p)$  then
    begin alivep := true ;
      forall  $q \in Pend_p^* \cup \{q : \langle req, v \rangle \in (sent_{qp}^* \setminus rcvd_{qp}^*)\}$ 
      do send  $\langle Alive \rangle$  to  $q$ 
    end
  end
end

```

Алгоритм 10.8. Алгоритм глобальной разметки

Теорема 10.14. Алгоритм глобальной разметки всегда завершает свою работу, и после завершения переменная $alive_p$ имеет значение $true$ в том и только том случае, когда процесс p является действующим в конфигурации γ^* .

Доказательство. Процесс p присваивает значение $true$ переменной $alive_p$ (и отправляет сопутствующие этому сообщения $\langle Alive \rangle$) не более одного раза, и поэтому проводится обмен лишь конечным числом сообщений.

Алгоритм разметки моделирует возможные продолжения вычислений базового алгоритма, начинающиеся из конфигурации γ^* , причем активными становятся лишь те процессы, у которых переменной $alive$ присваивается значение $true$, а сообщениями-разрешениями служат сообщения $\langle Alive \rangle$. В моделируемом продолжении вычисления всякий процесс немедленно отправляет разрешение всем отложенным запросам, как только становится активным. Процесс p , выполняю-

щий действие $\mathbf{M}_p$, является активным в конфигурации γ^* . Этот процесс может также стать активным в результате получения одного из тех сообщений, которые в конфигурации γ^* пребывают на этапе пересылки, и поэтому в этих случаях он считается действующим в γ^* . Процесс p , присвоивший переменной $alive_p$ значение $true$ в результате выполнения действия $\mathbf{P}_p$, также становится активным в моделируемом продолжении вычисления, так как он получил разрешающее сообщение, которое было отправлено в этом продолжении; и это также означает, что процесс является действующим в γ^* . Следовательно, все процессы p , у которых $alive_p = true$ в момент завершения работы алгоритмом разметки, являются действующими в γ^* .

Осталось доказать обратное, а именно что если процесс p является действующим в конфигурации γ^* , то переменной $alive_p$ будет присвоено значение $true$. Предположим, что p является действующим процессом в конфигурации γ^* и $(f_1, \dots, f_l)$ — это продолжение вычисления из γ^* , приводящее к конфигурации δ , в которой p становится активным процессом. Обратимся к последовательности конфигураций $(\gamma_0, \gamma_1, \dots, \gamma_l)$, соответствующих данному продолжению, где $\gamma_0 = \gamma^*$ и $\gamma_l = \delta$.

Воспользовавшись индукцией по i , покажем, что если γ_i — это первая конфигурация (в последовательности, начинающейся γ_0 и заканчивающейся γ_l), в которой процесс q является активным, то переменной $alive_q$ будет присвоено значение $true$ по ходу разметки.

Случай $i = 0$. Коль скоро процесс q является активным в γ^* , он может выполнить действие $\mathbf{M}_q$, присвоив тем самым переменной $alive_q$ значение $true$.

Случай $i > 0$. Процесс q оставался заблокированным во всех конфигурациях начиная с γ_0 и оканчивая γ_{i-1} , но стал активным при осуществлении события f_i . Очевидно, f_i — это событие приема сообщения $\langle \mathbf{grant}, v_q^* \rangle$, которое расширило набор $Grant_q$ до набора G , удовлетворяющего предикату $Free_q^*(G)$. Мы покажем, что для каждого $r \in G$ процесс r будет рано или поздно включен в состав $GrRec_q$ алгоритмом разметки. Рассмотрим два случая.

1. В конфигурации γ^* сообщение $\langle \mathbf{grant}, v_q^* \rangle$ было на этапе пересылки от процесса r к процессу q . Тогда процесс r уже был включен в состав $GrRec_q$, когда проводилась инициализация алгоритма разметки; первоначально значение $GrRec_q$ полагалось равным $Grant_q^* \cup \{r : \langle \mathbf{grant}, v_q^* \rangle \in sent_{rq}^* \setminus rcvd_{rq}^*\}$.

2. Сообщение $\langle \mathbf{grant}, v_q \rangle$ было отправлено при осуществлении одного из событий последовательности $f_1, \dots, f_{i-1}$. Тогда либо $(q, v_q) \in Pend_r$, либо сообщение $\langle \mathbf{req}, v_q \rangle$ находится на этапе пересылки к процессу r в конфигурации γ_i ; при этом процесс r является активным в одной из конфигураций последовательности $\gamma_0, \dots, \gamma_{i-2}$. По индуктивному предположению алгоритм разметки присваивает переменной $alive_r$ значение $true$. Как только это случится, процесс r отправляет сообщение $\langle \mathbf{Alive} \rangle$ процессу q , и после получения этого сообщения r заносится в $GrRec_q$.

Если сразу после инициализации выполняется включение $G \subseteq GrRec_q$, то можно выполнять действие $\mathbf{M}_q$, и в результате этого переменная $alive_q$ примет значение $true$. В противном случае переменной $alive_q$ будет присвоено значение $true$ позднее, когда будет доставлено сообщение $\langle \mathbf{Alive} \rangle$, пополняющее множество G .

В любом случае переменной $alive_p$ будет присвоено значение $true$. $\square$

Завершение работы алгоритма 10.8 явно не обозначено, но его можно выявить при помощи какого-нибудь алгоритма обнаружения завершения вычисления (см. гл. 8). Как только завершение работы алгоритма глобальной разметки будет зафиксировано, мертвыми в конфигурации γ^* будут объявлены все те и только те процессы, у которых $alive_p = false$.

10.4.3. Обнаружение тупиков в ограниченных моделях

Модель базового вычисления, которая использовалась в этом параграфе, является наиболее общей моделью при изучении проблемы обнаружения тупиков, и для этой модели известно совсем немного алгоритмов решения указанной задачи.

Один из алгоритмов, также осуществляющий моделирование продолжения вычисления базового алгоритма, был предложен Бржезинским, Хелари, Рейнелом в работе [37]. В этом алгоритме все контрольные сообщения сосредоточены в кольце, которое встраивается в коммуникационную сеть; по этому кольцу циркулирует маркер, и в нем указано множество процессов, о которых еще не стало известно, являются ли они действующими. Поскольку вся информация сконцентрирована в маркере, обнаружить завершение работы такого алгоритма разметки значительно легче: стоит маркеру пройти полный круг, не совершив изменения приписанного ему множества, и завершение работы алгоритма будет зафиксировано.

Задачу обнаружения тупиков часто изучают при наличии более строгих допущений, касающихся активизации процессов и допускающих описание тупиковых ситуаций в теоретико-графовых терминах. Обзор некоторых моделей и алгоритмов можно найти в статье Кнаппа [117]. Наиболее часто рассматриваются модели, в которых процессу, чтобы стать активным, необходимо получить разрешение от всех процессов системы (AND-модель) или хотя бы от одного процесса системы (OR-модель).

Тупиковая ситуация на уровне распределения ресурсов: AND-модель. Распределенная база данных состоит из совокупности файлов, рассредоточенных по большому числу сайтов (компьютеров). Система управления базой данных разрешает пользователям доступ к этим данным для чтения или для модификации. Доступ к данным открывается структурным методом посредством *транзакций* в базе данных, которые обычно направляют данные на разных сайтах. Ввиду разбросанности элементов данных нужно проследить за тем, чтобы избежать определенного чередования шагов, предпринимаемых различными транзакциями. Поэтому корректность операций над данными обычно обеспечивается посредством *блокирования* тех элементов данных, с которыми оперирует транзакция. И конечно же, если запрашиваемые данные уже заблокированы другой транзакцией, то нашей транзакции приходится ожидать.

Чтобы стать активной, транзакция должна получить доступ ко всем заблокированным элементам, которые ей требуются, т. е. условие освобождения имеет

вид $Free_p(G) \equiv (G = Req_{s_p})$. Ограниченную модель тупиковых ситуаций, в которой все запросы должны быть удовлетворены, прежде чем процесс станет активным, называют *AND-моделью*.

Можно показать, что в AND-модели наличие тупиковой ситуации равносильно существованию цикла в *графе ожидания*. Графом ожидания называется граф, вершинами которого являются процессы и при этом дуга pq проводится в том случае, когда процесс p заблокирован в ожидании процесса q . Алгоритмы поиска циклов в таком графе были предложены, в частности, в работах Ченди, Мисры и Хааса [50], Минаска и Мюнца [146], а также Митчелла и Меритта [150].

Тупиковая ситуация на уровне обмена информацией: OR-модель. Процесс, взаимодействующий с другими процессами в распределенном алгоритме, может перейти в заблокированное состояние, в котором единственно возможными событиями могут быть только события взаимосвязи с другими процессами. Осуществление всякого события взаимосвязи переводит этот процесс в другое состояние, в котором он может продолжить свое вычисление. Следовательно, чтобы стать активным, такому процессу достаточно, чтобы был удовлетворен хотя бы один его запрос, т. е. предикат $Free_p$ считается истинным на всяком непустом подмножестве семейства Req_{s_p} . Ограниченная модель тупиковых ситуаций, в которой одного-единственного разрешения достаточно, чтобы процесс стал активным, называется *OR-моделью*.

В OR-модели наличие тупиковой ситуации равносильно существованию узла в графе ожидания. Алгоритмы для этой модели были предложены, в частности, в работах Ченди, Мисры и Хааса [50] и Натараджана [153].

10.5. Упражнения к главе 10

10.5.1

Упражнение 10.1. Будем рассматривать регистрацию моментального локального состояния процесса p как еще одно внутреннее событие a_p . Докажите, что

$$S^* \text{ является значимым } \iff \forall p, q: a_p \parallel a_q.$$

10.5.2

Упражнение 10.2. Дайте полное описание алгоритма Лая—Янга, включив в него механизм, принуждающий завершать построение моментальных состояний и состояний каналов.

Упражнение 10.3. Профессор Пржлвичковский пишет: «После того как я ознакомился с главой 10, я сумел лучше понять алгоритмы из главы 8. Например, в алгоритме Сафры (алгоритм 8.7) обработку маркеров процессом p нужно понимать как вычисление моментального состояния процесса p . В построенном моментальном состоянии системы все процессы являются пассивными, так как маркер обрабатывается только пассивными процессами. Поэтому для вычисления значения $Pholds$ требуется всего лишь проверить, пусты ли все каналы. Для

этого в маркере указывается суммарное значение счетчиков сообщений. Однако мне неясна та роль, которую играют окраски *white* и *black*, а также как удается обеспечить значимость моментальных состояний системы.»

Не могли бы Вы помочь профессору?

Г Л А В А 11

ВОСПРИЯТИЕ НАПРАВЛЕНИЯ И ОРИЕНТАЦИЯ

В сетях, имеющих регулярную структуру, наподобие торов или гиперкубов, в соединительных линиях обычно указывается их направление. Мы обсудим здесь результаты сравнительно недавних исследований, оценивающих ту пользу, которую приносит подобная разметка; эту разметку мы будем называть *восприятием направления* или сокращенно SoD¹⁾. Восприятие направления усиливает модель и дает процессам возможность более эффективно обмениваться информацией друг с другом, а также использовать в своих алгоритмах топологические особенности сетей связи.

Как было установлено, восприятие направления позволяет понизить сложность некоторых задач, однако тот класс задач, для решения которых применима эта возможность, оказывается необычайно мал. На самом деле, есть много задач для которых известны эффективные или простые алгоритмы, использующие разметку дуг. Тем не менее, во многих нетривиальных случаях хорошие нижние оценки, которые известны для неразмеченных вариантов задачи, оказываются недостижимыми, и, более того, для неразмеченных графов удалось построить несколько невероятно эффективных алгоритмов.

В этой главе предпринята попытка проследить усилия, инициированные Санторо в работе [168], с целью выявления тех вариантов задачи о выборах и задачи ширококестельного распространения сообщений, в которых восприятие направления приносит выгоду.

Мы определим восприятие направления для некоторых специальных классов сетей и покажем, что задача о выборах на кольцах может быть решена более эффективно при наличии восприятия направления, если в нашем распоряжении будут хорды. Мы покажем, что, располагая восприятием направления, избрание лидера на гиперкубах и кликах можно осуществить с линейной сложностью, но при этом такой же сложности можно достичь при помощи рандомизованных алгоритмов, в которых не используется SoD. Мы обсудим также алгоритмы формирования SoD в сетях, где его первоначально не было.

Обзор главы. В § 11.6.1 вводятся некоторые общие понятия и ставятся задачи, которые будут обсуждаться в последующих параграфах. В § 11.1.1 даются строгие определения восприятия направления, а в § 11.1.2 приводятся примеры, иллюстрирующие его применение. В § 11.1.3 описано, как эффективно провести ширококестельное отправление сообщения, используя *униформное* восприятие направления. В § 11.6.2 приведены некоторые результаты, касающиеся

выборов на простых кольцах и кольцах с хордами с учетом восприятия направления; при этом будет показано, что восприятие направления позволяет уменьшить сложность избрания лидера на кликах.

В § 11.6.3 изучаются преимущества, которые дает восприятие направления на гиперкубах. Несколько лет назад стали известны очень эффективные алгоритмы для размеченных гиперкубов; алгоритмы для неразмеченных гиперкубов были менее эффективны, и исследования были сосредоточены на доказательстве соответствующих нижних оценок. Не так давно, однако, были предложены более эффективные алгоритмы для неразмеченных гиперкубов, свидетельствующие о том, что повышение эффективности достигается за счет особенностей топологии, а не за счет разметки дуг.

В § 11.4 обсуждается ряд вопросов, связанных со сложностью алгоритмов, включая алгоритмы формирования восприятия направления в неразмеченных сетях. В конце главы (§ 11.5) мы проведем обсуждение открытых вопросов.

11.1. Введение и определения

Как обычно, мы будем моделировать сеть процессов графом с N вершинами и m дугами. Дуги, примыкающие к каждому процессу, имеют локальные имена, и поэтому процесс может различать, по какой дуге к нему поступила информация, и может выбирать, по какой дуге ему отправлять информацию. Процессоры имеют разные отличительные признаки, но этими признаками являются числа, которые не имеют специального истолкования и не привязаны к топологии сети. Мы будем иметь дело с асинхронными сетями.

11.1.1. Определения и характеристические свойства восприятия направления

Хотя восприятию направления было уделено немало внимания в последние годы, простое определение этого понятия так и не было согласовано. В работе [190] были определены некоторые частные случаи, а в работе [86] был выделен ряд классов восприятия направления.

Определения из теории групп. Мы будем использовать методы теории групп, и поэтому начнем с того, что напомним некоторые понятия из теории групп. Тем читателям, кто незнаком с теорией групп, некоторые из определений могут показаться загадочными, но мы, по мере возможности, снабдим наши определения конкретными примерами.

Коммутативной, или абелевой, *группой* называется множество G с выделенным *нулевым* элементом 0 и бинарной операцией $+$, которые удовлетворяют следующим требованиям.

1. *Замкнутость.* Для любых элементов $x, y \in G$ выполняется соотношение $(x + y) \in G$.

2. *Тождественность.* Для любого элемента $x \in G$ выполняются равенства $0 + x = x + 0 = x$.

¹⁾От англ. Sense of Direction. — Прим. перев.

3. *Обратимость.* Для любого элемента $x \in G$ существует такой элемент $y \in G$, что $x + y = y + x = 0$.

4. *Ассоциативность.* Для любых элементов $x, y, z \in G$ выполняется равенство $(x + y) + z = x + (y + z)$.

5. *Коммутативность.* Для любых элементов x, y выполняется равенство $x + y = y + x$.

Вследствие закона ассоциативности скобки при проведении суммирования могут быть опущены. Мы будем использовать запись $-x$ для элемента, обратного элементу x . Если $s \in \mathbb{Z}$, то запись $s \cdot x$ будет обозначать сумму, состоящую из s элементов x .

Количество элементов в множестве G называется *порядком*; при изучении восприятия направления порядок $\text{ord}(G)$ будет полагаться конечным. В конечных группах для каждого элемента x существует такое положительное число k , что $k \cdot x = 0$; наименьшее из подобных положительных чисел называется *порядком* элемента x , и оно всегда является делителем порядка группы.

Для произвольного набора элементов $g_1, \dots, g_k$ рассмотрим множество всех тех элементов группы, которые представимы в виде суммы этих элементов g_i :

$$S = \left\{ x : \exists s_1, \dots, s_k : x = \sum_{i=1}^k s_i \cdot g_i \right\}.$$

Это множество само является группой; оно называется *подгруппой, порожденной элементами* $g_1, \dots, g_k$, и обозначается записью $\langle g_1, \dots, g_k \rangle$. Для элемента $y \in G$ множество $T = S + y = \{y + x : x \in S\}$ называется *орбитой* подгруппы S . Все орбиты подгруппы S имеют одинаковый размер, причем две орбиты $S + y_1$ и $S + y_2$ либо совпадают, либо не пересекаются. Таким образом, орбиты подгруппы S разбивают группу G на непересекающиеся множества.

Группа называется *циклической*, если она порождается единственным элементом, т. е. существует такой элемент g , что $G = \langle g \rangle$. Порождающий элемент не единствен (исключение составляет группа порядка 2), но мы обычно выбираем некоторый фиксированный порождающий элемент и обозначаем его 1; запись i при этом будет обозначать элемент $i \cdot 1$. Циклическую группу порядка k будем обозначать $\mathbb{Z}_k$.

Таким образом, $\mathbb{Z}_k$ — группа сложения натуральных чисел по модулю k : ее элементами являются числа, начиная с 0 и оканчивая $k-1$. Другая группа, заслуживающая особого внимания, — это группа $(\mathbb{Z}_k)^d$, элементами которой являются векторы длины d , а в качестве групповой операции выступает покомпонентное сложение по модулю k .

Сети и разметки. В дальнейшем мы будем полагать, что G — это коммутативная группа и для соседних процессов p и q запись $\mathcal{L}_p(q)$ обозначает имя, которое используется в процессе p для обозначения связи pq . В основе идеи использования теоретико-группового восприятия направления лежит соответствие между топологией сети и структурой группы.

Определение 11.1. Разметка дуг $\mathcal{L}$ считается *восприятием направления* (на основе G), если все дуги помечены элементами группы G и существует такое инъективное отображение $\mathcal{N}$ множества вершин сети в множество G , что для любой пары соседних процессов p и q выполняется равенство $\mathcal{N}_q = \mathcal{N}_p + \mathcal{L}_p(q)$.

Если задано восприятие направления $\mathcal{L}$, то та разметка вершин сети, которая упомянута в приведенном определении, называется *свидетельствующей разметкой* или просто *свидетельством* для $\mathcal{L}$. Процессоры осведомлены о разметке связей $\mathcal{L}$, но при этом не предполагается и не требуется, чтобы свидетельствующая разметка вершин была известна процессам; эта разметка *не является* составной частью восприятия направления. Нетрудно показать (см. [192]), что свидетельствующая разметка вершин не единственна; всякую заданную свидетельствующую разметку можно модифицировать, добавив к пометке каждой вершины один и тот же элемент группы в качестве слагаемого. Поэтому количество свидетельствующих разметок для восприятия направления в точности равно порядку группы $\text{ord}(G)$.

Восприятие направления можно охарактеризовать и без упоминания разметки вершин, а именно при помощи свойств замкнутых путей (см. упражнение 11.1).

Обратим внимание на то, что SoD не использует элемент 0 для разметки дуг (ввиду того, что соседние вершины помечены по-разному) и что SoD удовлетворяет свойству *антисимметричности* $\mathcal{L}_p(q) + \mathcal{L}_q(p) = 0$, или, что то же самое, $\mathcal{L}_p(q) = -\mathcal{L}_q(p)$. Обычно получается так, что порядок группы равен числу вершин сети, и, следовательно, свидетельствующая разметка является биекцией. Это свойство никак не учитывается в большинстве алгоритмов, которые мы опишем здесь, но часто оно неявно подразумевается.

Для заданной сети, состоящей из N процессов, и группы G порядка N нетрудно построить восприятие направления. Вначале припишем всем вершинам различные элементы группы G , а затем возьмем в качестве разметки каждой дуги $\mathcal{L}_p(q)$ элемент $\mathcal{N}_q - \mathcal{N}_p$. В результате получим восприятие направления.

Как мы увидим в дальнейшем, полученная таким образом разметка позволяет значительно уменьшить сложность отдельных задач, однако большинство разметок, которые оказались особенно эффективными, обладают еще одним дополнительным качеством — *униформностью*.

Униформность. Процессоры в параллельных компьютерах часто соединены так, что образуется заранее известная симметрическая структура (т. е. граф выглядит одинаково «с точки зрения» любого процесса); как раз симметрия и осведомленность о топологии подразумеваются в униформном восприятии направления.

Определение 11.2. Восприятие направления называется *униформным*, если каждый процесс обладает одним и тем же набором локальных пометок.

Обозначим символом L общее множество пометок на дугах. Тогда некоторые свойства униформного восприятия направления следуют немедленно из определения.

1. Если $g \in L$, то $-g \in L$: связующая дуга, помеченная элементом g на одном конце, помечена элементом $-g$ на другом конце.

2. Множество L порождает G , поскольку для каждого $g \in G$ существует такой путь P , что $SUM_L(P) = g$.

3. Для всякого элемента $g \in G$ кратчайший путь P , для которого выполняется равенство $SUM_L(P) = g$, может быть вычислен локально на основе обобщенного варианта задачи о размене монеты. Достаточно вычислить минимальную последовательность пометок из L , дающих в сумме g , и, поскольку у каждого процесса есть такие пометки, эта последовательность задает путь в сети.

Мы будем описывать сеть с равномерным восприятием направления, просто указывая множество пометок, используемых в каждом процессе, и опуская при этом обратные элементы.

Примеры. В научных публикациях упоминаются несколько примеров восприятия направления, и большинство из них подпадают под наше определение. В работе Флокчини и др. [86] определено *хордовое* восприятие направления: это всякое восприятие направления над группой $\mathbb{Z}_N$. Особого внимания заслуживает равномерное хордовое SoD, в котором 1 (или всякий иной порождающий элемент) содержится среди пометок. *Хордовое кольцо* размера N похоже на обыкновенное кольцо, но только из каждой вершины исходят «срезающие пути», соединяющие ее с вершинами, отстоящими на несколько шагов далее по кольцу.

Определение 11.3. *Хордовым кольцом* $C_N(c_1, \dots, c_k)$ называется сеть, в которой существует равномерное хордовое восприятие направления, и при этом

$$L = \{-c_k, \dots, -c_1, -1, 1, c_1, \dots, c_k\},$$

где k называется числом хорд.

Требование, чтобы порождающий элемент группы использовался в качестве пометки, очень существенно. Так, например, $k \times l$ -тор с циклическим возвратом и взаимно простыми k и l имеет равномерное восприятие направления, но не является при этом хордовым кольцом. Мы дадим альтернативные определения в терминах теории групп ряду уже известных топологий.

Определение 11.4. Тором размер $n \times n$ называется сеть, для которой существует равномерное SoD над группой $(\mathbb{Z}_n)^2$ и при этом $L = \{(0, 1), (0, -1), (1, 0), (-1, 0)\}$. (Для удобства эти пометки обозначаются N, S, E, W .)

Определение 11.5. *Гиперкубом размерности n* называется сеть, для которой существует равномерное восприятие направления над группой $(\mathbb{Z}_2)^n$ с n пометками, обозначаемыми $i = (0, \dots, 1, \dots, 0)$, где $i = 0, \dots, n - 1$.

Определение 11.6. Кликой называется сеть, для которой существует равномерное SoD над группой $\mathbb{Z}_N$ и при этом $L = \mathbb{Z}_N \setminus \{0\}$.

11.1.2. Применение восприятия направления

Мы приведем описание нескольких наиболее важных методов, в основу которых положено применение восприятия направления. Далее в этой главе для иллюстрации и пояснения этих методов будет представлено полное описание алгоритмов.

Сравнение путей. Если заданы два пути π_1 и π_2 , исходящие из одной и той же вершины, то пометки, приписанные дугам этих путей, позволяют узнать, оканчиваются ли эти пути в одной и той же вершине или нет. Действительно, все, что нам нужно, — это просуммировать пометки на каждом пути; вершины, в которые ведут эти пути, будут совпадать в том и только том случае, когда вычисленные суммы будут равны.

Восприятие направления, таким образом, равносильно осведомленности о соседях, и благодаря этому становится возможен обход сети в глубину с использованием $O(N)$ обменов сообщениями, как это было установлено в §6.4.3. Для того чтобы это можно было проделать, восприятие направления не обязательно должно быть равномерным.

Возможность сравнивать пути настолько присуща восприятию направления, что Флокчини и др. в статье [86] предложили считать это свойство *определением* восприятия направления. В предложенном ими подходе разметка непосредственно сопровождается построением явной транслирующей функции, позволяющей вычислять относительное местоположение конечной точки пути. С одной стороны, это определение является более общим, так как оно описывает более обширный класс разметок. Однако преимущества такого подхода ограничены: этот обширный класс включает в себя осведомленность о соседях (согласно нашему определению это качество *не является* примером восприятия направления) и к тому же некоторые разметки чрезвычайно редко применяются при проектировании распределенных алгоритмов. С другой стороны, теоретико-групповой подход дает возможность получить компактное представление транслирующей функции (как сложение элементов групп), позволяя тем самым сосредоточить внимание на более углубленных методах применения восприятия направления.

Сжатие путей. Если восприятие направления является равномерным, то оно дает нам полные сведения о топологии сети. Действительно, в таком случае восприятие направления невозможно рассматривать в отрыве от полной осведомленности о топологии. Равномерное восприятие направления позволяет провести эффективную маршрутизацию в любую вершину, относительное местоположение которой известно, при помощи обобщенного варианта задачи о размене монеты, упомянутой нами ранее.

Структура сети. Равномерное восприятие направления позволяет управлять общим ходом алгоритма, руководствуясь теоретико-групповым представлением структуры сети. В качестве примеров будут рассмотрены алгоритм широковещательного распространения сообщений (алгоритм 11.2) и один из вариантов алгоритма избрания лидера на гиперкубе (§11.3.2).

11.1.3. Широковещательная рассылка сообщений при наличии восприятия направления

Ввиду того, что лейтмотивом этой главы является сопоставление сложности широковещательного распространения сообщений при наличии и при отсутствии восприятия направления, мы покажем здесь, что восприятие направления позволяет проводить широковещательную связь с использованием $O(N)$ обменов сообщениями. Если же информация о топологии недоступна, то для широковещательной рассылки сообщений требуется совершить хотя бы по одному обмену сообщениями по каждому каналу связи (теорема 6.6), и, следовательно, сложность по числу обменов сообщениями составляет величину $\Omega(|E|)$.

Применение восприятия направления для поиска в глубину. Как было отмечено Менсом и Санторо, алгоритм поиска в глубину с использованием осведомленности о соседях (§ 6.4.3) можно легко обобщить на случай сетей с восприятием направления. Напомним, что алгоритм 6.17 проводит перемещение маркера, в котором содержится список имен ранее посещенных процессов. Отправление маркера по листовидным дугам удается полностью устранить, так как процесс, передающий маркер, будет проверять, содержится ли имя соседа в указанном списке.

```
var  $father_p$  : process  init  $undef$  ;
```

Только для инициатора, выполнить один раз:

```
begin  $father_p := p$  ; choose  $q \in Neigh_p$  ;
      send  $\langle tlist, \{0\} \rangle$  to  $q$ 
end
```

Для каждого процесса после получения сообщения $\langle tlist, L \rangle$ по каналу λ :

```
begin forall  $x \in L$  do  $x := x + \lambda$  ;
  if  $father_p = undef$  then  $father_p := \lambda$  ;
  if  $\exists q \in Neigh_p \setminus L$ 
    then begin choose  $q \in Neigh_p \setminus L$  ;
              send  $\langle tlist, L \cup \{0\} \rangle$  to  $q$ 
            end
  else if  $p$  is initiator
    then decide
  else send  $\langle tlist, L \cup \{0\} \rangle$  to  $father_p$ 
end
```

Алгоритм 11.1. Алгоритм поиска в глубину с SoD

В алгоритме 11.1 показано, как использовать восприятие направления вместо осведомленности о соседях. Список, передаваемый вместе с маркером, содержит не *имя* процесса (мы не пользуемся предположением о том, что оно известно), а указание местоположения процесса относительно той вершины, в которой

пребывает маркер. В алгоритме применяется способность проводить «сравнение путей»: процесс q заносит себя в список, поместив туда элемент 0. После того как этот список пройдет путь π , сумма пометок на котором составляет элемент σ , и достанется, например, процессу p , элемент 0 превратится в $-\sigma$. Заметим здесь, что $\mathcal{L}_q(p) = \sigma$ и $\mathcal{L}_p(q) = -\sigma$, и поэтому в этом списке можно обнаружить пометку дуги pq .

Единственное дополнительное действие, которое нужно внести в алгоритм 6.17, состоит в том, что по получении указанного списка все относительные позиции в нем уточняются за счет прибавления к каждой из них пометки того канала, по которому этот список был доставлен.

Этот алгоритм можно использовать не только для широковещательной рассылки сообщений, но также и для проведения выборов лидера по алгоритму Каттена и др. (§ 7.5.4), и в таком случае мы получим алгоритм, требующий $O(N \log N)$ обменов сообщениями. Тем самым, достигнут первый успех на пути применения восприятия направления, поскольку для задач широковещательной рассылки сообщений и избрания лидера *без SoD* у нас есть точные нижние оценки (теоремы 6.6 и 7.15). Мы подчеркиваем, что восприятие направления не обязано быть равномерным, и поэтому возможности сравнивать пути вполне достаточно для получения этого результата.

Следствие 11.7. В тех сетях, в которых отсутствует осведомленность о топологии, наличие восприятия направления уменьшает сложность широковещательной рассылки сообщений с величины $\Theta(m)$ до величины $\Theta(N)$, а сложность проведения выборов лидера с величины $\Theta(N \log N + m)$ до величины $\Theta(N \log N)$.

Структурный алгоритм для равномерного SoD. Равномерное восприятие направления позволяет процессам извлекать из сети остоное дерево «на лету» и вести по этому дереву широковещательную рассылку сообщений, используя в точности $N - 1$ обменов сообщениями. Эта величина оптимальна, так как каждый из $N - 1$ процессов, помимо инициатора рассылки, должен быть ознакомлен с рассылаемой информацией за счет получения сообщения.

Вначале мы приведем простой пример этого алгоритма: предположим, что сеть представляет собой $n \times n$ -тор, дуги которого помечены «Север», «Восток», и т. п. Широковещательная связь проводится так. Инициатор отправляет сообщение на Восток, и сообщение совершает $n - 1$ переходов в восточном направлении. Инициатор, а также каждая вершина, в которую поступает сообщение с Запада, отправляет сообщение на Север, и сообщение совершит в точности $n - 1$ шагов в этом направлении.

Легко видеть, что такая стратегия срабатывает и при этом происходит в точности $N - 1$ обменов сообщениями; если добавить к ним $N - 1$ подтверждающих сообщений, то алгоритм будет проводить широковещательную рассылку сообщений с обратной связью (PIF). Если бы мы отправили сообщение на один шаг дальше на Восток, то его следующим адресатом стал бы сам инициатор, что излишне. Если бы мы отправили сообщение на один шаг дальше на Север, его

получил бы процесс, который сам же и отправил его на Север, а это тоже излишне.

Этот пример показывает, каким образом равномерное восприятие направления дает информацию о структуре всей сети в целом и как эта информация используется в алгоритме. Теперь мы дадим общее описание этого алгоритма, полагая, что в основу сети положена произвольная группа. Расположим пометки начиная с g_1 и оканчивая g_k (опуская обратные элементы) в произвольном порядке и определим величины $n_1, \dots, n_k$ следующим образом. Прежде всего, величину n_1 положим равной порядку элемента g_1 . Далее положим n_2 равной порядку g_2 в группе $G/\langle g_1 \rangle$; эта величина будет равна наименьшему натуральному числу, для которого g_1 является делителем $n_2 \cdot g_2$. И для каждого i в качестве n_i выбирается порядок элемента g_i в группе $G/\langle g_1, \dots, g_{i-1} \rangle$; это наименьшее число, для которого элемент $n_i \cdot g_i$ можно представить в виде суммы элементов $\{g_1, \dots, g_{i-1}\}$. Числа n_i обладают следующим свойством: произведение всех этих чисел равно N , и каждый элемент g из группы G можно представить *единственным* образом в виде суммы

$$g = \sum_{i=1}^k s_i \cdot g_i, \quad \text{где } 0 \leq s_i < n_i.$$

Для инициатора:

```
for i = 1 to k
  do if  $n_i > 1$  then send  $\langle \text{info}, n_i - 1 \rangle$  via link  $g_i$ 
```

После получения сообщения $\langle \text{info}, s \rangle$ по каналу $-g_j$:

```
if  $s > 1$  then send  $\langle \text{info}, s - 1 \rangle$  via link  $g_j$ ;
for i = j + 1 to k
  do if  $s_i > 1$  then  $\langle \text{info}, s_i - 1 \rangle$  via link  $g_i$ 
```

Алгоритм 11.2. Широковещательная рассылка с равномерным восприятием направления

В алгоритме 11.2 информация отправляется на $n_1 - 1$ переходов в направлении g_1 и достигает всех процессов, отличие которых от инициатора описывается элементом группы, кратным g_1 . Инициатор и все процессы, получающие сообщение по направлению $-g_1$, отправляют его по направлению g_2 на $n_2 - 1$ переходов; таким образом, сообщение достигает всех процессов, отличие которых от инициатора выражается элементом группы, представляющим собой сумму элемента, кратного g_1 , и элемента, кратного g_2 . В общем случае инициатор и все процессы, получившие сообщение по направлениям начиная с $-g_1$ и оканчивая $-g_{i-1}$, отправляют это сообщение далее по направлению g_i на $n_i - 1$ переходов. Поскольку каждый процесс, отличный от инициатора, получает сообщение в точности один раз, в алгоритме 11.2 происходит в точности $N - 1$ обменов сообщениями.

В случае необходимости можно добавить $N - 1$ обратных сообщений и превратить этот алгоритм в PIF-алгоритм.

В случае необходимости неинициаторы могут запоминать те каналы, по которым к ним было доставлено сообщение; в результате будет образовано остовное дерево сети, корнем которого будет инициатор. Это дерево можно использовать для отправления инициатору уведомляющих сообщений (см. § 11.3.2). Глубина остовного дерева, которое строится рассматриваемым алгоритмом, равна $\sum_{i=1}^k (n_i - 1)$; в случае необходимости глубину можно изменить, избрав другой порядок расположения элементов $g_1, \dots, g_k$ в списке пометок L . Мы не будем здесь заниматься исследованием этого вопроса, который связан с временной сложностью широковещательного распространения информации.

11.2. Выборы на кольцах и хордовых кольцах

Задача избрания лидера на кольцах и хордовых кольцах имеет долгую историю (см. § 7.5.2). В § 11.2.1 мы опишем решение, Франклином в работе [90], которое было улучшено Аттьей и др. в работе [10] за счет использования хорд, добавленных (равномерно) к кольцу. В их алгоритме большую роль играет способность сжимать пути, и, как будет показано в §§ 11.2.2 и 11.2.3, линейной сложности можно достичь, добавив к кольцу совсем немного хорд. В этом алгоритме, однако, используется одна лишь способность сжимать пути, не затрагивающая особенностей структуры сети. В § 11.2.4 мы продемонстрируем, что для получения линейного алгоритма понадобится еще меньше хорд, если структура сети также будет принята в расчет.

11.2.1. Алгоритм Франклина

В алгоритме Франклина проводится разделение процессов на *активные* и *релейные*, а между турами устанавливается преемственность. Перед началом первого тура все процессы считаются активными. Если в начале тура имеется *более одного* активного процесса, то в этом туре в худшем случае половина из них, а в лучшем случае все кроме одного станут релейными процессами, и затем начнется следующий тур. Тот тур, в котором есть только один активный процесс, является завершающим; этот оставшийся процесс избирается лидером. Таким образом, число туров не превышает величины $\lceil \log N \rceil + 1$, и, поскольку в каждом туре используется в точности $2N$ сообщений, сложность алгоритма по числу сообщений в худшем случае приближенно равна $2N \log N$.

В каждом туре активные процессы обмениваются своими именами с ближайшими активными соседями справа и слева. Если «ближайшим активным соседом» к какому-либо процессу является он сам, то это означает, что этот процесс и есть единственный активный процесс; тогда он становится лидером. В противном случае он переходит в следующий тур только в том случае, если его имя превосходит имена обоих его «активных соседей» справа и слева. Это условие соблюдается по меньшей мере для одного процесса (процесса с самым старшим именем), но не более, чем для половины из них (поскольку оно нарушается для

```

statep := active ;
while statep = active do
  begin send (name, p) to left ; (name, p) to right ;
    receive (name, x) from right ; receive (name, y) from left ;
    if x = p and y = p then statep := elected ;
    if x > p or y > p then statep := relay
  end

```

Алгоритм 11.3. Алгоритм Франклина избрания лидера на кольце

обоих соседей всякого «уцелевшего» процесса). Активный процесс, ближайший активный сосед которого имеет более старшее имя, становится релейным процессом. Если релейный процесс получает сообщение от одного из своих соседей, то он просто передает его другому соседу; в алгоритме 11.3 это не указано.

11.2.2. Усовершенствование Атты

Число сообщений, отправляемых *активными* процессами в алгоритме, линейно относительно размеров кольца, и в алгоритме 11.4 релейными процессами уничтожается как можно больше сообщений.

Так как число активных процессов после каждого тура сокращается вдвое, *сумма по всем $\log N$ турам* числа активных процессов, участвующих в выборах, ограничена величиной $2N$. Поскольку всякий активный процесс отправляет два сообщения на каждом туре, число сообщений, отправляемых активными процессами, линейно и равно $4N$. Однако сообщениям приходится преодолевать все большее и большее расстояние, ввиду того что после i -го тура между двумя соседними активными процессами расположено не менее 2^i релейных. Таким образом, ретрансляция сообщений пассивными процессами вносит решающий вклад в сложность по числу сообщений.

В кольцевых сетях это неизбежно; даже если активный процесс осведомлен о том, насколько он отдален от своего активного соседа, сообщение должно быть отправлено по кольцу, ибо это кратчайший путь. В алгоритме Атты сохранена принципиальная схема алгоритма Франклина, но при этом предполагается, что активные процессы располагают сведениями о расстоянии, отделяющем их от ближайших активных соседей, и им разрешается использовать хорды, а также сжатие путей, для того чтобы быстро доставить сообщение $\langle \text{name}, p \rangle$. Хорды не применяются, для того чтобы изменить основные правила соревнования, заложенные в алгоритме.

Теперь мы перейдем к рассмотрению хордового кольца с восприятием направления. В начале каждого тура активный процесс осведомлен об относительном местоположении ближайшего активного напарника как слева, так и справа, и он отправляет им свое имя, но используя при этом хорды. Предполагается, что сжатие осуществляется при помощи базовых операций $\text{Send}(\text{message}, g)$ и $\text{Receive}(\text{message}, g)$. Первая из операций отправляет сообщение message по

пути, для которого $\text{SUM}_L = g$, а вторая операция обеспечивает прием такого сообщения. Одной из возможных реализаций этих операций является *жадная маршрутизация*, при которой сообщение всегда отправляется по самой большой хорде, не дающей «перелета» (см. алгоритм 11.5). Конечно, можно применять и более изощренный способ выбора дуг, но это может повлиять на сложность получившегося в результате алгоритма избрания лидера лишь незначительно.

```

statep := active ; Leftp := -1 ; Rightp := +1 ;
while statep = active do
  begin Send((name, p), Left) ; Send((name, p), Right) ;
    Receive((name, x), Right) ; Receive((name, y), Left) ;
    if p ≥ x and p ≥ y then
      begin Send((pos, 0), Left) ; Send((pos, 0), Right);
        Receive((pos, r), Right) ; Right := Right + r ;
        Receive((pos, l), Left) ; Left := Left + l ;
        if Leftp = 0 then statep := elected
      end
    else
      begin (* ретранслировать сообщение (pos, ·) *)
        Receive((pos, r), Right) ; Send((pos, r + Right), Left);
        Receive((pos, l), Left) ; Send((pos, l + Left), Right)
      end
    end
  end

```

Алгоритм 11.4. Алгоритм Атты избрания лидера на хордовом кольце

Первоначально относительные координаты левого и правого активного соседа задаются числами -1 и $+1$ соответственно. После завершения тура уцелевшие активные процессы определяют относительные координаты ближайших активных процессов, отправляя сообщение $\langle \text{pos}, \cdot \rangle$ по цепочке процессов, которые стали релейными именно после этого тура. Лидерство устанавливается в конце того тура, в котором уцелеет единственный процесс, за счет того что этот процесс обнаружит, что он является своим собственным соседом ($\text{Left} = 0$).

Новый алгоритм отличается от алгоритма Франклина только тем, как осуществляют взаимосвязь активные процессы; суммарное количество активных процессов по всем турам остается, как и прежде, равным $2N$. Теперь число сообщений, отправляемых активными процессами в каждом раунде, возросло до 4, и общее число операций Send на протяжении работы алгоритма достигло величины $8N$.

Вначале мы обсудим особый случай, когда сеть является кликой; алгоритм для этого случая был предложен еще ранее в работе Луи и др. в работе [134]. Так как сеть является вполне связной, операция Send доставляет сообщение непосредственно, на это затрачивается одна передача, и общая сложность по числу сообщений будет ограничена $8N$ сообщениями. И это вновь благоприятно для нас, поскольку Корач и др. в работе [119] установили нижнюю оценку $\Omega(N \log N)$ для задачи о выборах на кликах без восприятия направления.

Следствие 11.8. При наличии восприятия направления в клике сложность задачи о выборах уменьшается с величины $\Theta(N \log N)$ до величины $\Theta(N)$.

В случае хордового кольца общего вида Аттъя ввел *функцию стоимости сообщения* F : это такая наименьшая монотонная и выпуклая функция, что выполнение операции $Send(m, g)$ требует использования не более $F(g)$ обменов сообщениями. (Функцию F можно представлять как стоимость отправления сообщения процессу g , удаленному на заданное число переходов по кольцу.) В худшем случае в i -м туре могут принять участие $N/(2^{i-1})$ активных процессов, отстоящих друг от друга на 2^{i-1} переходов, и на нем придется использовать $\frac{N}{2^{i-1}} \times 4 \times F(2^{i-1})$ передач сообщений.

Лемма 11.9. В худшем случае в алгоритме 11.4 используется $4N \cdot \sum_{i=0}^{n-1} \frac{F(2^i)}{2^i}$ передач сообщений.

Лемма устанавливает, как общая сложность проведения выборов зависит от стоимости отправления сообщений на кольцо. В особых случаях сложность проведения выборов на кольце (где $F(g) = g$) и клике (где $F(g) = 1$) будет составлять величину $O(N \log N)$ и $O(N)$, соответственно. Аттъя и др. показали, что логарифмического числа хорд достаточно, чтобы ограничить эту сумму константой, например для хордового кольца $C_N(2, 3, 4, \dots, \lg N)$. Мы воздержимся здесь от воспроизведения выкладок.

procedure $Send(m, g)$:

begin if $g = 0$
 then deliver message m to waiting $Receive$
 else begin $ch :=$ the largest of g_i such that $ch \leq g$;
 send $\langle m, g - g_i \rangle$ through link ch
 end
end

Upon receipt of $\langle m, g \rangle$:

$Send(m, g)$

Алгоритм 11.5. Стратегия жадной маршрутизации

11.2.3. Минимизация числа хорд

Результат Аттъя устанавливает, что сложность проведения выборов на хордовых кольцах линейна; теперь сосредоточим внимание на задаче минимизации числа хорд, необходимого для достижения такой сложности. В этом параграфе мы установим асимптотику числа хорд, которое необходимо алгоритму Аттъя, используя метод «восстановления графа по формуле». Представленный здесь

результат отражает общую тенденцию в изучении алгоритмов, когда исследования становятся все менее «алгоритмическими», а основное внимание уделяется анализу. Действительно, коль скоро алгоритмическая сторона задачи о выборах разработана и отношения между хордами и сложностью задачи оказались «спрятаны» в формулах леммы 11.9), наши усилия сосредотачиваются на рабочих формулах.

Сумма геометрической прогрессии. Коль скоро геометрическая прогрессия (знаменатель которой меньше 1) имеет ограниченную сумму, мы вначале попробуем выяснить, сколько хорд необходимо, чтобы ограничить дробь $\frac{F(2^i)}{2^i}$ величиной вида c^i , где $c < 1$.

Рассмотрим последовательность хорд $L = \{2, 4, 16, 256, \dots\}$, в которой охват каждой последующей хорды выражается величиной, составляющей квадрат от аналогичного показателя предыдущей; последовательность оканчивается, когда размах очередной хорды превосходит N . Размах хорд выражается формулой 2^{2^i} , а их число равно $\log \log N$.

Лемма 11.10. $F(d) \leq 2 \cdot \sqrt{d}$.

Доказательство. Проведем индукцию по d , используя *жадную* маршрутизацию, в которой построение пути, составляющего сумму d , начинается с *наибольшей хорды* l , подходящей для d , и далее пойдем по пути длины $d - l$. Вначале $F(1) = 1 \leq 2\sqrt{1}$.

Предположим теперь, что $d > 0$ и наше соотношение справедливо для всякого $d' < d$. Выбор хорд проводится так, что самая длинная хорда, подходящая для d имеет такую длину l , что $\sqrt{d} < l \leq d$. В результате мы получаем неравенства

$$F(d) \leq 1 + F(d - l) \leq 1 + 2\sqrt{d - l} < 1 + 2\sqrt{d - \sqrt{d}} < 2\sqrt{d}. \quad \square$$

Таким образом, дробь $\frac{F(2^i)}{2^i}$ ограничена величиной $2 \cdot \left(\frac{1}{\sqrt{2}}\right)^i$, ряд, представленный в лемме 11.9 ограничивается суммой, и эта сумма ограничена сверху. Мы показали, что выборы с линейной сложностью возможны при наличии $O(\log \log N)$ хорд. Оказывается, что $\Omega(\log \log N)$ также и необходимо (если применяется жадная маршрутизация), чтобы ограничиться суммированием геометрической прогрессии!

Лемма 11.11. Если существует такая константа $c < 1$, что при использовании жадной маршрутизации выполняется неравенство $\frac{F(2^i)}{2^i} < c^i$, то число хорд составляет величину $\Omega(\log \log N)$.

Доказательство. Для каждого $i < \log N$ имеет место неравенство $F(2^i) < c^i \cdot 2^i$, и отсюда следует, что задействованы хорды, размах которых не меньше $(1/c)^i$. В соответствии со стратегией жадной маршрутизации длина такой хорды не превышает 2^i . Следовательно, для каждого i есть хорда, размах которой лежит в отрезке $[(1/c)^i \dots 2^i]$. Заметим, что при $i' = i \cdot \log 1/c$ отрезки для i и i'

не пересекаются. Значит, для $\log \log N$ различных значений $i_r = (\log 1/c)^r$ (где $r < \log \log N$) мы получаем семейство непересекающихся отрезков $[(1/c)^{i_r} \dots 2^{i_r}]$, в каждом из которых укладывается размах некоторой хорды. $\square$

Применение других стратегий маршрутизации позволит, пожалуй, улучшить лишь постоянный множитель; доказательство этого тезиса остается открытым вопросом. Построение F по заданному L связано с задачей о размене монеты, в которой требуется выплатить оговоренную сумму, используя лишь монеты определенного достоинства. Общая стратегия маршрутизации предусматривает использование результата решения задачи о размене монеты.

Более медленно убывающие суммы. Мы убедились, что геометрическая прогрессия в соотношении из леммы 11.9 приводит к линейной сложности, но при этом требуется проведение $\Theta(\log \log N)$ хорд. Поэтому мы выберем последовательность, которая убывает еще медленнее, но при этом обладает ограниченной суммой, а именно, квадратично-гармоническую последовательность $\sum (1/i^2)$.

Рассмотрим семейство хорд $L = \{36, 64, 256, 65536, \dots\}$, в котором $g_{i+1} = 2^{\sqrt{g_i}}$; 36 является наименьшим числом, для которого эта формула дает большую степень 2. Так как $g_i = \log^2(g_{i+1})$, для каждого d существует хорда, размах которой попадает в промежуток между $\log^2 d$ и d .

Лемма 11.12. *В L содержится менее $2 \log^* N$ хорд.*

Доказательство. Так как $g_{i+2} = 2^{\sqrt{2^{\sqrt{g_i}}}} = 2^{(\sqrt{2})^{\sqrt{g_i}}} > 2^{g_i}$, последовательность величин $g_0, g_2, g_4, g_6, \dots$ возрастает быстрее, чем последовательность $2, 2^2, 2^{2^2}, 2^{2^{2^2}}, \dots$, и ее члены будут превышать N спустя $\log^* N$ шагов. $\square$

Для маршрутизации сообщений на короткие дистанции ($d < 36$) нужны короткие хорды, но их будет всего лишь некоторое фиксированное число.

Лемма 11.13. *Справедливо неравенство $F(d) \leq 2 \cdot \frac{d}{\log^2 d}$.*

Доказательство. Для больших значений d алгоритм жадной маршрутизации сначала выберет хорду l , размах которой превышает $\log^2 d$. В результате будут получены следующие соотношения:

$$F(d) = 1 + F(d - l) \leq 1 + 2 \cdot \frac{d - l}{\log^2(d - l)} < 1 + 2 \cdot \frac{d - \log^2 d}{\log^2(d - \log^2 d)} < 2 \cdot \frac{d}{\log^2 d}. \quad \square$$

Проведенные вычисления показывают, что для того, чтобы провести выборы с линейной сложностью, достаточно $\Theta(\log^* N)$ хорд. Теперь перейдем к соответствующей нижней оценке числа хорд. Доказательство основывается на вычислении сумм ряда, элементы которого убывают еще медленнее, но при этом сами суммы не ограничены: это гармонический ряд $\sum (1/i)$. Известно, что первые $\log n$ членов этого ряда дают в сумме величину, составляющую $\Theta(\log \log N)$, однако для того, чтобы ее достичь, требуется $\Omega(\log^* N)$ хорд!

Лемма 11.14. *Если применяется жадная маршрутизация и верно неравенство $\frac{F(2^i)}{2^i} < 1/i$, то имеет, по меньшей мере, $\log^* N$ хорд.*

Доказательство. Доказательство проводится так же, как была обоснована предыдущая нижняя оценка: для каждого i должна найтись хорда, размах которой укладывается в отрезок между i и 2^i . $\square$

Подведем итоги:

- 1) $O(\log^* N)$ хорд достаточно для сходимости квадратично-гармонического ряда (лемма 11.13);
- 2) $\Omega(\log^* N)$ хорд требуется для расходимости гармонического ряда (лемма 11.14).

Чтобы получить линейную сложность, суммы из леммы 11.9 должны быть ограничены, и, следовательно, должны убывать быстрее, нежели суммы гармонического ряда; для этого требуется, по меньшей мере, $\Omega(\log^* N)$ хорд. Отсюда следует, что $\Theta(\log^* N)$ хорд необходимо и достаточно для того чтобы алгоритм Атты имел линейную сложность.

11.2.4. Однохордовый линейный алгоритм

В этом параграфе будет представлен алгоритм избрания лидера на хордовом кольце, имеющий линейную сложность и требующий всего лишь одной хорды. Мы уже убедились в том, что алгоритм Атты этого сделать не может: в нем используется лишь способность сжимать пути, а структура сети никакой роли не играет.

Рассмотрим теперь хордовое кольцо с одной-единственной хордой размаха t , и при этом будем полагать, что величина t приближенно равна $\sqrt{N}$. Топология хордового кольца с единственной хордой, размах которой близок к $\sqrt{N}$, напоминает тор, и наш алгоритм будет представлять собой адаптацию алгоритма Петерсона (см. [160]) для торов.

В начале каждого тура всякий активный процесс пытается отыскать, или *увидеть*, другой процесс, который считается активным в том же самом туре; процесс видит самое большее еще один процесс, но его могут видеть многие процессы. Есть два способа продвижения какого-либо активного процесса в следующий тур.

Первый из них весьма похож на продвижение по принципу «локального максимума» из алгоритма Франклина: если процесс замечает другой процесс с младшим именем и его также *замечает* процесс с младшим именем, то наш процесс переходит в следующий тур. Не более половины процессов могут перейти в следующий тур, руководствуясь таким правилом: ведь если какой-либо процесс перешел в следующий тур, то хотя бы один младший процесс, который его заметил, в следующий тур переведен не был.

Было бы слишком накладно заставлять каждый активный процесс заниматься поиском в сети, до тех пор пока он не заметит другой активный процесс; поэтому активные процессы обшаривают только ограниченное подмножество сети, и поиск может завершиться так, что никакого другого процесса заметить не удастся. Второе правило продвижения указывает, что в таком случае процесс, проводивший поиск, переходит в следующий тур!

Главное достоинство этого алгоритма состоит в организации процедуры поиска таким образом, чтобы 1) область поиска была достаточно велика и позволяла лишь немногим процессам перейти в следующий тур по второму правилу и 2) поиск был достаточно эффективным и позволял получить в итоге небольшую сложность. Теперь представим себе, что мы очерчиваем все большие и большие квадраты на бесконечной решетке и замечаем, что число точек на границе растет гораздо медленнее, чем число точек во внутренней области.

Квадраты и границы, процедуры поиска. Для каждого элемента $g \in \mathbb{Z}_N$ назовем l -квадратом элемента g множество элементов

$$S_{g,l} = \{g + i \cdot 1 + j \cdot t : 0 \leq i \leq l, 0 \leq j \leq l\}.$$

Множество $S_{g,l}$ — это множество (не более чем) $(l+1)^2$ процессов, доступных из g за счет пересечения не более l 1-дуг и не более l t -дуг. Активный процесс занимается поиском другого активного процесса в l -квадрате, но не отправляя сообщений всем процессам из этого квадрата, ибо это было бы слишком дорого.

Назовем *границей* такое множество точек $B_{g,l}$, для которых хотя бы одно из указанных нестрогих неравенств становится равенством, т. е.

$$B_{g,l} = \{g + i \cdot 1, g + i \cdot 1 + l \cdot t, g + i \cdot t, g + l \cdot 1 + i \cdot t : 0 \leq i \leq l\},$$

а *внутренней областью* назовем множество $I_{g,l} = S_{g,l} \setminus B_{g,l}$. На границе $B_{g,l}$ располагаются (самое большее) $4l$ процессов, и, кроме того, если $S_{g,l}$ и $S_{h,l}$ пересекаются, то $B_{g,l}$ и $B_{h,l}$ также пересекаются.

Процессор, участвующий в i -м туре, начинает поиск других процессов, отправляя зонд-маркер по границе l -квадрата. Этот маркер пытается совершить l переходов в направлении 1, l переходов в направлении t , l переходов в направлении -1 и l переходов в направлении $-t$. Размер квадрата зависит от номера тура; в i -м туре длина стороны l_i выбирается равной α^i (где α — константа, немногим превосходящая 1). Для обхода всей границы требуется совершить обмен $4l$ сообщениями, но этот обход может быть прерван по следующим причинам.

Если маркер p посетит процесс, в котором уже побывал другой маркер, выпущенный в более высоком туре, обход прерывается; процесс p никогда не получит обратно свой зонд и не перейдет в последующий тур. Если маркер впервые попадает в процесс, в котором уже побывал (отличный от него) маркер, выпущенный в том же туре (скажем, процессом q), то процесс p «замечает» процесс q . Если имя процесса q младше, чем имя процесса p , то процесс p должен быть об этом уведомлен, так как способность заметить процесс q очень важна для продвижения процесса p в следующий тур по первому правилу. Если же имя процесса q старше имени процесса p , то p утрачивает шанс быть переведенным в следующий тур, но процесс q должен быть проинформирован об этом, так как тот факт, что процесс p заметил процесс q , мог бы быть очень важен для продвижения q в следующий тур по первому правилу. Таким образом, либо маркер возвращается обратно к процессу p , либо маркер-преследователь отправляется к процессу q по границе, которую обходит зонд-маркер, выпущенный процессом q , с тем чтобы оповестить q о том, что его заметил процесс с младшим именем. И коль скоро для

процесса q достаточно, чтобы его оповестили об одном процессе с младшим именем, которому удалось заметить q , процесс, отправивший маркер-преследователь процессу q , уже не будет отправлять больше маркеров-преследователей, относящихся к процессу p .

На основе приведенного описания процедуры обработки маркеров можно сформулировать следующие локальные свойства, определяющие продвижение процесса p в следующий тур. Во-первых, зонд-маркер процесса p возвращается с оповещением о том, что удалось заметить процесс с младшим именем, *и, кроме того*, маркер-преследователь прибывает в p с сообщением о том, что какой-то процесс с младшим именем заметил p . Во-вторых, зонд-маркер процесса p возвращается с уведомлением о том, что никакого процесса с младшим именем заметить не удалось.

Мы покажем, что в каждый тур проходит по меньшей мере один процесс. Если ни одному из процессов не удалось заметить других, то в следующий тур проходят все процессы. Допустим, что хотя бы одному процессу удалось заметить другой процесс. Пусть процесс r — это процесс с самым старшим именем из числа тех, которые были замечены. Тогда либо процессу r не удалось заметить другие процессы, и он проходит в следующий тур согласно второму правилу, либо процесс r заметил процесс с младшим именем и r проходит в следующий тур по первому правилу.

Некоторые выкладки Наш алгоритм похож на алгоритм Петерсона из работы [160], и здесь будут присутствовать те же самые выкладки. Обозначим символом A_i количество активных процессов, которые приняли участие в i -м туре; очевидно, $A_0 \leq N$. Вначале мы получим верхнюю оценку величины A_i для всех туров, в которых размер квадратов меньше N , т. е. для всех туров, номера которых удовлетворяют неравенству $i \leq \log_\alpha \sqrt{N}$.

Лемма 11.15. *Для туров, номера которых удовлетворяют неравенству $i \leq \log_\alpha \sqrt{N}$ имеет место оценка $A_i \leq \frac{N}{\alpha^{2i}(2 - \alpha^2)}$.*

Доказательство. Как мы уже убедились, каждый процесс, перешедший в следующий тур по первому правилу, был замечен другим процессом, который в следующий тур не перешел. Поэтому не более половины процессов, которым удалось заметить какой-либо другой, были переведены в следующий тур по этому правилу.

Некоторые процессы могли перейти в следующий тур по второму правилу (за счет того, что их не заметили другие процессы), но при этом требуется, чтобы для двух процессов p и q , перешедших в следующий тур благодаря этому правилу, квадраты $S_{p,l}$ и $S_{q,l}$ не пересекались. Действительно, если бы эти квадраты налагались друг на друга, то пересекались бы и их границы, по которым двигались зонды-маркеры, запущенные процессами p и q . Один из таких маркеров достиг бы точки пересечения первым, лишив другой маркер возможности продолжить обход так, чтобы не был замечен ни один процесс. Следовательно, количество процессов, переведенных в следующий тур по второму правилу, меньше N/l_i^2 .

Исходя из этого, а также принимая во внимание выбор $l_i = \alpha^i$, сделанный нами ранее, мы приходим к заключению о том, что

$$A_{i+1} \leq \frac{A_i + N/\alpha^{2i}}{2}.$$

Применяя индукцию по i , можно убедиться в том, что отсюда следует оценка

$$A_i \leq \frac{N}{\alpha^{2i}(2 - \alpha^2)}$$

□

Из этой леммы вытекает линейная оценка числа обменов сообщениями, а также константная оценка количества процессов, которым удалось пробиться в $\log_\alpha \sqrt{N}$ -й тур.

Следствие 11.16. *На протяжении первых туров, начиная с 0-го тура и оканчивая $\log_\alpha \sqrt{N}$ -м туром, совершается менее $\frac{8\alpha}{(\alpha - 1)(2 - \alpha^2)} N$ обменов сообщениями.*

Доказательство. В лемме 11.15 уже установлена оценка числа активных процессов, участвующих в i -м туре. Оценка числа обменов сообщениями проводится так. Зонд-маркер всякого активного процесса p может совершить $4\alpha^i$ переходов, и это заносится на счет процесса p . Маркеры-преследователи, направляющиеся к p , также заносят свой вклад на счет процесса p . Каждый процесс, который был посещен зонд-маркером, запущенным процессом p , может отправить самое большее один маркер-преследователь, и поэтому на счет процесса p можно занести не более $4\alpha^i$ переходов, совершенных маркерами-преследователями. Числом

Таким образом, на долю каждого из $\frac{N}{\alpha^{2i}(2 - \alpha^2)}$ процессов, участвующих в i -м туре, приходится не более $8\alpha^i$ обменов сообщениями, и общее число обменов сообщениями в i -м туре ограничено величиной $\frac{8N}{2 - \alpha^2} \cdot (1/\alpha)^i$. Эти величины образуют геометрическую прогрессию, сумма которой по всем турам равна $\frac{8N}{2 - \alpha^2} \cdot \frac{\alpha}{\alpha - 1}$. □

Завершаемость работы алгоритма. Из леммы 11.15 можно заключить, что число процессов, успешно выдержавших первые $\log_\alpha \sqrt{N}$ туров, ограничено сверху константой $1/(2 - \alpha^2)$. Чтобы избрать лидера из этих оставшихся процессов, наш алгоритм переходит к другому этапу работы и начинает работать, как алгоритм Ченя—Роберта (§7.2.1). Всякий процесс, выдержавший $\log_\alpha \sqrt{N}$ -й тур, отправляет маркер по всему кольцу, т. е. сообщение совершает N переходов в направлении $+1$. Этот маркер поглощается всяким другим процессом с более старшим отличительным признаком, который также участвует в этом туре; маркер также прерывает активность всех процессов, пребывающих на первом этапе работы алгоритма. На втором этапе работы алгоритма гарантируется, что только процесс с наибольшим отличительным признаком, сумевший пройти $\log_\alpha \sqrt{N}$ -й тур, получит свой маркер обратно; этот процесс и избирается лидером.

Теорема 11.17. *Описанный алгоритм позволяет избрать лидера с использованием $O(N)$ обменов сообщениями на хордовом кольце с одной хордой (имеющей размах $\sqrt{N}$).*

Доказательство. Коль скоро на первом этапе работы алгоритма не все процессы оказываются убитыми, хотя один процесс перейдет ко второму этапу работы алгоритма. На втором этапе гарантируется избрание лидером в точности одного процесса, и этим обосновывается корректность построенного алгоритма.

Согласно следствию 11.16 на первом этапе совершается $O(N)$ обменов сообщениями, а на втором этапе происходит только $O(N)$ обменов сообщениями, поскольку на этом этапе число процессов, отправивших маркер по кольцу, ограничено константой. □

11.3. Вычисления в гиперкубах

В этом параграфе мы исследуем вопрос о том, позволяет ли восприятие направления уменьшить сложность ширококвещательного распространения сообщений и проведения выборов в гиперкубах. Долгое время предполагалось, что ответ на этот вопрос должен быть утвердительным, потому что известные алгоритмы, в которых используется восприятие направления, превосходили по качеству наилучшие известные алгоритмы, работающие в неразмеченных гиперкубах. Тем не менее, ввиду отсутствия соответствующих нижних оценок для случая неразмеченных гиперкубов строгого обоснования получить не удавалось.

В последние годы были предложены невероятно эффективные алгоритмы для неразмеченных гиперкубов, ликвидировавшие разрыв в известных оценках сложности между случаями размеченных и неразмеченных гиперкубов. В §11.3.1 рассматривается случай, когда в сети отсутствует восприятие направления и нет никакой осведомленности о топологии сети; т. е. мы рассматриваем алгоритмы, которые работают корректно в произвольной сети, и изучаем их сложность при работе в гиперкубе. В §11.3.2 мы обратимся к тому случаю, когда наличествует как осведомленность о топологии, так и восприятие направления, и продемонстрируем, насколько элегантными, простыми и эффективными могут быть полученные алгоритмы.

Трудной частью исследований является случай, когда есть осведомленность о топологии сети, но нет восприятия направления: это значит, что от алгоритма требуется всего лишь правильная работа в гиперкубах, но при этом никакой разумной разметки ребер не предполагается. Было установлено, что ширококвещательную рассылку сообщений можно провести с использованием линейного числа обменов сообщениями; в §11.3.3 представлен алгоритм ориентации гиперкуба, а в §11.3.4 введен метод трафаретов и алгоритм ширококвещательной рассылки сообщений. В §11.3.5 обсуждаются некоторые методы, используемые в современных алгоритмах проведения выборов лидера для неразмеченных гиперкубов. Случай, когда есть восприятие направления, но нет осведомленности о топологии, нами не рассматривается.

11.3.1. Базис: отсутствие осведомленности о топологии сети

При отсутствии восприятия направления и осведомленности о топологии сети, для проведения как широковещательной рассылки, так и выборов лидера требуется, чтобы были задействованы все дуги сети для передачи хотя бы одного сообщения. Действительно, обязательное условие правильной работы алгоритма на всяком графе позволяет воспользоваться методом «вставки вершины», как это было сделано в теоремах 6.6 и 7.15. Следовательно, такие алгоритмы при выполнении на гиперкубе будут использовать по меньшей мере $\Omega(N \log N)$ обменов сообщениями.

С другой стороны, широковещательную рассылку можно осуществить при помощи потока или алгоритма эха (алгоритм 6.4), а выборы можно провести, используя алгоритм Галладжера и др. алгоритм 7.10/7.11/7.12). Таким образом, обе задачи можно решить с использованием $O(N \log N)$ обменов сообщениями.

Теорема 11.18. *Избрание лидера и широковещательная рассылка сообщений в гиперкубах без осведомленности о топологии сети требует $\Theta(N \log N)$ обменов сообщениями.*

11.3.2. Алгоритм посредничества

В алгоритме посреднического избрания лидеров, применяющемся для гиперкубов с восприятием направления, используется рекурсивная структура гиперкубического графа. Чтобы избрать лидера в гиперкубе размерности $d + 1$, этот алгоритм вначале проводит выборы в каждой грани размерности d , образованной всеми порождающими элементами, за исключением последнего, а затем избирает лидером одного из этих двух победителей. Во избежание путаницы между лидерствами на различных этапах работы алгоритма вершина называется *d-лидером*, если ей удалось одержать победу в выборах в одной из d -мерных граней.

Базу для работы алгоритма создают выборы в грани размерности, и она проста; сеть состоит в точности из одной вершины, которая становится 0-лидером непосредственно.

Турнир. Вершина, ставшая d -лидером (для $d < n$), приглашается на турнир с d -лидером смежной по направлению d грани. Турнир между двумя вершинами, которые могут сообщаться друг с другом прямо, организовать просто. Каждая вершина отправляет другой вершине сообщение, в котором содержится ее имя; вершина, которой будет доставлено более старшее имя, чем ее собственное, становится *нелидером*, а вершина, которой будет доставлено более младшее имя, чем ее собственное, становится *лидером*.

Турнир между двумя d -лидерами проводится точно так же, за исключением затруднения, касающегося того, что вершины не знают, как добраться до лидера на другой (или даже на их собственной) грани. В качестве первого шага вершина p , ставшая d -лидером, отправляет извещение о турнире $\langle \text{tour}, p, d \rangle$, в котором содержится ее имя и номер этапа d , по дуге, исходящей из нее в направлении d ; это сообщение получает одна из вершин другой d -мерной грани. При этом

на вершину, принявшую указанное сообщение, которую будем теперь называть *входной* вершиной, возлагается обязанность доставить это сообщение своему d -лидеру.

Трудность доставки рассматриваемого сообщения d -лидеру состоит в том, что входная вершина не осведомлена об относительном местоположении своего d -лидера. Для d -лидера было бы слишком накладно оповещать о своем местоположении все вершины своего d -мерного куба, чтобы входная вершина могла доставить сообщение за d шагов. Такое оповещение обошлось бы в $2^d - 1$ обменов сообщениями, и это привело бы весь алгоритм проведения выборов к общей сложности $O(N \log N)$. Точно так же, было бы накладно обязывать входную вершину рассылать извещение о турнире широковещательно по всему d -кубу; на это тоже потребовалось бы $2^d - 1$ обменов сообщениями.

В нашем решении используется сочетание этих двух основных стратегий: d -лидер объявляет о своем статусе всем вершинам $\lfloor d/2 \rfloor$ -мерной грани, которую в дальнейшем будем называть *строкой* лидера. Входная вершина осуществляет широковещательную рассылку извещения о турнире по грани размерности $\lfloor d/2 \rfloor$, которая называется ее *столбцом*. Так как каждая строка пересекается с каждым столбцом в точности в одной точке (это будет показано далее), найдется одна вершина, которая получит оба сообщения: и оповещение от своего d -лидера, и извещение о турнире. Эту вершину будем называть *вершиной-посредником*. Вершина-посредник переправит извещение о турнире далее своему d -лидеру по основному дереву, которое проведет оповещение лидера.

Определение 11.19. Пусть задан гиперкуб размерности d .

Строкой называется грань в гиперкубе, которую образуют порождающие элементы начиная с 0 и оканчивая $\lfloor d/2 \rfloor - 1$. *Столбцом* называется грань в гиперкубе, которую образуют порождающие элементы начиная с $\lfloor d/2 \rfloor$ и оканчивая $d - 1$.

Турнир между двумя лидерами проводится следующим образом.

1. d -лидер p отправляет сообщение $\langle \text{tour}, p, d \rangle$ по каналу d ;
2. d -лидер широковещательно информирует свою строку о своем статусе, используя алгоритм 11.2; вершины строки сохраняют в памяти номер канала, по которому они получили оповещение, и в результате образуется остовное дерево этой строки; (алгоритм широковещательной рассылки имеет простой вид, потому что порождающие элементы не зависят друг от друга, и каждое значение n_i равно 1; счетчик переходов в сообщении можно подавить);
3. входная вершина (т.е. вершина, получившая извещение $\langle \text{tour}, p, d \rangle$ по каналу d) широковещательно информирует об этом свой столбец, используя все тот же алгоритм 11.2.
4. процесс q , не являющийся d -лидером, но расположенный в строке лидера (это означает, что q уже получил сообщение $\langle \text{ann}, d \rangle$) и получивший сообщение $\langle \text{tour}, p, d \rangle$, отправляет это сообщение своей родительской вершине;
5. d -лидер q после получения сообщения $\langle \text{tour}, p, d \rangle$ сравнивает значения p и q . Если $q > p$, то процесс q становится $(d + 1)$ -лидером; в противном случае q становится нелидером.

Коммуникационная сложность. Условимся обозначать записью $T(d)$ количество обменов сообщениями, совершенных на турнире между двумя d -лидерами, а записью $E(n)$ — количество обменов сообщениями, которое используется при работе алгоритма избрания лидера (на гиперкубе размерности n). Так как для проведения выборов требуется проведение выборов в двух кубах меньшей размерности, а также проведение турнира, мы получаем следующее рекурсивное соотношение

$$E(n) = \begin{cases} 0, & \text{если } n = 0, \\ 2 \cdot E(n-1) + T(n-1) & \text{в противном случае.} \end{cases}$$

Для анализа сложности по числу обменов сообщениями нам понадобится уже известная формула суммы бесконечной убывающей геометрической прогрессии (со знаменателем $\alpha < 1$) и ее производная по α :

$$\sum_{c=0}^{\infty} \alpha^c = \frac{1}{1-\alpha} \quad \text{и} \quad \sum_{c=0}^{\infty} c \cdot \alpha^{c-1} = \frac{1}{(1-\alpha)^2}.$$

Подсчитывая число обменов сообщениями на каждом шаге турнира между двумя d -лидерами, мы обнаружим, что:

шаг 1: 2 сообщения;

шаг 2: $2 \cdot (2^{\lfloor d/2 \rfloor} - 1)$ сообщений;

шаг 3: $2 \cdot (2^{\lceil d/2 \rceil} - 1)$ сообщений;

шаг 4: at most $2 \cdot \lfloor d/2 \rfloor$ сообщений;

В сумме мы имеем $T(d) \leq 2 \cdot (2^{\lfloor d/2 \rfloor} + 2^{\lceil d/2 \rceil} + \lfloor d/2 \rfloor - 1)$.

Для четных значений d мы имеем $\lfloor d/2 \rfloor = \lceil d/2 \rceil = d/2$, а для нечетных значений d выполняются равенства $\lfloor d/2 \rfloor = (d/2) - \frac{1}{2}$ и $\lceil d/2 \rceil = (d/2) + \frac{1}{2}$; на основании этого мы получаем

$$\begin{aligned} T(2c) &= 2 \cdot (2^c + 2^c + c - 1) = 4 \cdot 2^c + 2c - 2, \\ T(2c+1) &= 2 \cdot (2^c + 2^{c+1} + c - 1) = 6 \cdot 2^c + 2c - 2. \end{aligned}$$

Введем функцию $F(n) = E(n)/2^n$ и, воспользовавшись рекурсивным соотношением для E , получим соотношение

$$F(n) = \begin{cases} 0, & \text{если } n = 0, \\ F(n-1) + \frac{T(n-1)}{2^n} & \text{в противном случае.} \end{cases}$$

Оно позволяет нам записать $F(n)$ в виде конечной суммы, которую можно оценить сверху суммой бесконечного ряда:

$$F(n) = \sum_{d=0}^{n-1} \frac{T(d)}{2^{d+1}} < \sum_{d=0}^{\infty} \frac{T(d)}{2^{d+1}}.$$

Теперь воспользуемся выражением для T , которое было выведено нами для четных и нечетных значений d , сортируем слагаемые числителя на показательные,

линейные и постоянные и получим в итоге следующую цепочку равенств

$$\begin{aligned} \sum_{d=0}^{\infty} \frac{T(d)}{2^{d+1}} &= \sum_{c=0}^{\infty} \frac{T(2c)}{2^{2c+1}} + \sum_{c=0}^{\infty} \frac{T(2c+1)}{2^{2c+2}} = \\ &= \sum_{c=0}^{\infty} \frac{4 \cdot 2^c + 2c - 2}{2^{2c+1}} + \sum_{c=0}^{\infty} \frac{6 \cdot 2^c + 2c - 2}{2^{2c+2}} = \\ &= \sum_{c=0}^{\infty} \left(\frac{4 \cdot 2^c}{2^{2c+1}} + \frac{6 \cdot 2^c}{2^{2c+2}} \right) + \sum_{c=0}^{\infty} \left(\frac{2c}{2^{2c+1}} + \frac{2c}{2^{2c+2}} \right) - \\ &\quad - \sum_{c=0}^{\infty} \left(\frac{2}{2^{2c+1}} + \frac{2}{2^{2c+2}} \right) = \\ &= 3 \frac{1}{2} \cdot \left(\sum_{c=0}^{\infty} \frac{1}{2^c} \right) + \frac{3}{8} \cdot \left(\sum_{c=0}^{\infty} \frac{c}{4^{c-1}} \right) - \\ &\quad - \frac{3}{2} \cdot \left(\sum_{c=0}^{\infty} \frac{1}{4^c} \right) = \\ &= 3 \frac{1}{2} \cdot 2 + \frac{3}{8} \cdot \frac{16}{9} - \frac{3}{2} \cdot \frac{4}{3} = \\ &= 7 + \frac{2}{3} - 2 = 5 \frac{2}{3}. \end{aligned}$$

Отсюда следует, что $E(n) < 5 \frac{2}{3} \cdot N$.

11.3.3. Алгоритм потока по многим путям

Теперь мы представим алгоритм, вычисляющий восприятие направления в гиперкубе размерности n , и этот алгоритм послужит нам опорой при разработке алгоритма широкоэвентального распространения сообщений, имеющего линейную сложность. В алгоритме, который мы будем рассматривать, предполагается, что есть процесс, назначенный на роль инициатора алгоритма (лидера), и каналы связи, примыкающие к лидеру, помечены произвольным образом числами $0, \dots, n-1$. Разметка каналов инициатора однозначно определяет восприятие направления, о чем свидетельствует следующая теорема (которую мы приводим здесь без доказательства).

Теорема 11.20. Пусть задана вершина w и разметка $\mathcal{P}$ дуг, исходящих из w , числами, начинающимися с 0 и оканчивающимися $n-1$. Тогда существует единственная ориентация $\mathcal{L}$, удовлетворяющая условию $\mathcal{L}_w(v) = \mathcal{P}(v)$ для каждого соседа вершины w .

Наш алгоритм вычисляет в точности эту ориентацию и, кроме того, единственную свидетельствующую разметку вершин, в которой лидер помечен набом-

ром $(0, \dots, 0)$. В алгоритме используются три типа сообщений. Лидер отправляет каждому из своих соседей метку того канала, который их соединяет, в сообщениях $\langle \mathbf{dmn}, i \rangle$. Нелидеры отправляют свои метки вершин другим процессам в сообщениях $\langle \mathbf{iam}, nla \rangle$. Нелидеры информируют своих соседей о метках каналов связи в сообщениях $\langle \mathbf{labl}, i \rangle$.

```

begin  $num\_rec_p := 0$  ;  $dis_p := 0$  ;  $lbl_p := (0, \dots, 0)$  ;
  for  $l = 0$  to  $n - 1$  do (* Отправление сообщений на этапе 1 *)
    begin send  $\langle \mathbf{dmn}, l \rangle$  по каналу  $l$  ;
       $\pi_p[l] := l$ 
    end ;
  while  $num\_rec_p < n$  do (* Прием сообщений на этапе 2 *)
    begin receive  $\langle \mathbf{labl}, l \rangle$  ; (* обязательно по каналу  $l$  *)
       $num\_rec_p := num\_rec_p + 1$ 
    end
end

```

Алгоритм 11.6. Ориентация гиперкуба (Инициатор)

Рассматриваемый алгоритм состоит из двух процедур — алгоритм 11.6 (для лидера) и алгоритм 11.7 (для нелидеров). Работа алгоритма разбита на два этапа: на первом этапе поток сообщений расходится от лидера, а на втором этапе поток сообщений направлен к лидеру. *Предшественником* вершины p называется такой ее сосед q , для которого выполняется неравенство $d(q, w) < d(p, w)$, а *последователем* вершины p называется такой сосед q вершины p , для которого выполняется неравенство $d(q, w) > d(p, w)$. В гиперкубе у вершины p нет соседей q , для которых выполнялось бы равенство $d(q, w) = d(p, w)$, и всякая вершина, расположенная на расстоянии d от лидера, имеет d предшественников и $n - d$ последователей.

Лидер запускает алгоритм, отправляя сообщение $\langle \mathbf{dmn}, i \rangle$ по каналу связи, помеченному i . Когда процессу p , не являющемуся лидером, становится известно расстояние dis_p от него до лидера и, кроме того, ему доставлены все сообщения от его предшественников, процесс p может приступить к вычислению пометки nla_p своей вершины. Он отправляет эту пометку в сообщении $\langle \mathbf{iam}, nla_p \rangle$ своим последователям. Чтобы убедиться в том, что p может осуществить это, рассмотрим вначале случай, когда p получает сообщение $\langle \mathbf{dmn}, i \rangle$ по каналу l . Так как сообщение было отправлено лидером, $dis_p = 1$ и все его остальные соседи являются его последователями. Процессор p помечает свою вершину двоичным набором, в котором в позиции i ставится 1, а во всех остальных позициях ставится 0. (В таком случае меткой канала l становится i .) Далее p отправляет сообщение $\langle \mathbf{iam}, nla_p \rangle$ по всем каналам $k \neq l$.

Теперь обратимся к случаю, когда процесс p получает сообщение $\langle \mathbf{iam}, label \rangle$. Из этого сообщения извлекается расстояние d от отправителя сообщения до лидера (оно равно количеству единиц в наборе $label$). Так как сообщения $\langle \mathbf{iam}, label \rangle$ отправляются только последователям, отправитель этого сообщения является

```

begin  $num\_rec_p := 0$  ;  $dis_p := n + 1$  ;  $lbl_p := (0, \dots, 0)$  ;
  forall  $l$  do  $nei_p[l] := nil$  ;
  while  $num\_rec_p < dis_p$  do (* Прием сообщений на этапе 1 *)
    begin receive  $msg$  по каналу  $l$  ;  $num\_rec_p := num\_rec_p + 1$  ;
      (*  $msg$  — это сообщение одного из двух типов  $\langle \mathbf{dmn}, i \rangle$  или  $\langle \mathbf{iam}, nla \rangle$  *)
      if  $msg$  is  $\langle \mathbf{dmn}, i \rangle$  then
        begin  $dis_p := 1$  ;
           $nei_p[l] := (0, \dots, 0)$  ;  $lbl_p[i] := 1$ 
          (* Теперь в наборе  $lbl_p = (0, \dots, 1, \dots, 0)$  есть только одна 1 *)
        end
      else
        begin  $dis_p := 1 + \#$  of 1's in  $nla$  ;
           $lbl_p := (lbl_p \text{ or } nla)$  ;
           $nei_p[l] := nla$ 
        end
      end ;
    (* Отправление сообщений на этапе 1 *)
    forall  $l$  with  $nei_p[l] = nil$  do
      send  $\langle \mathbf{iam}, lbl_p \rangle$  по каналу  $l$  ;
    while  $num\_rec_p < n$  do (* прием сообщений на этапе 2 *)
      begin receive  $\langle \mathbf{labl}, i \rangle$  по каналу  $l$  ;
         $num\_rec_p := num\_rec_p + 1$  ;  $\pi_p[l] := i$ 
      end ;
    (* Отправление сообщений на этапе 2 *)
    forall  $l$  with  $nei_p[l] \neq nil$  do
      begin  $\pi_p[l] :=$  бит, которым различаются  $lbl_p$  и  $nei_p[l]$  ;
        send  $\langle \mathbf{labl}, \pi_p[l] \rangle$  по каналу  $l$ 
      end
    end
end

```

Алгоритм 11.7. Ориентация гиперкуба (Неинициатор)

предшественником процесса p , и поэтому $dis_p = d + 1$. Отсюда становится ясно, сколько предшественников имеет процесс, и p ожидает, пока не будут получены все dis_p сообщений $\langle \mathbf{iam}, label \rangle$. После этого процесс p определяет пометку своей вершины, вычислив покомпонентную дизъюнкцию тех пометок, которые были им получены, и отправляет ее тем из своих соседей, от которых не было доставлено сообщений $\langle \mathbf{iam}, label \rangle$, так как именно они являются его последователями.

На первом этапе каждый процесс p , не являющийся лидером, вычисляет свою пометку. На втором этапе каждый процесс p , не являющийся лидером, узнает от своих последователей ориентацию каналов, связывающих его с ними, и вычисляет ориентацию своих каналов, связывающих процесс p с его предшественниками. Эта информация отправляется по каждому каналу в сообщении $\langle \mathbf{labl}, i \rangle$. Процессор отправляет сообщения $\langle \mathbf{labl}, i \rangle$ своим предшественникам, как только он получает подобные сообщения от всех своих последователей, и после этого пре-

крашает вычисления. Лидер завершает работу, когда от всех его соседей будет получены сообщения $\langle \text{labl}, i \rangle$.

В процессе p используются следующие переменные: num_res_p для учета количества полученных сообщений; dis_p для обозначения расстояния между процессом и лидером (это значение вычисляется, как только прибывает первое сообщение, а первоначально оно полагается равным $n + 1$); lbl_p для обозначения пометки вершины, вычисленной процессом p ; $\text{nei}_p[0..n - 1]$ для хранения пометок вершин, в которых располагаются предшественники процесса p ; $\pi_p[0..n - 1]$ для хранения данных, обеспечивающих восприятие направления.

Лемма 11.21. *Описанный алгоритм завершает работу в каждом процессе.*

Доказательство. Воспользовавшись индукцией по d , нетрудно проверить, что все процессы, отстоящие от лидера на расстояние, не превышающее d , рано или поздно отправят все сообщения, которые необходимо отправить на первом этапе. Для $d = 0$ лишь один лидирующий процесс отстоит от лидера (т. е. от самого себя) на расстояние d , и он отправляет все сообщения, не дожидаясь вначале поступления других сообщений. Предположим, что все процессы, отстоящие от лидера на расстояние d , отправили все свои сообщения на первом этапе работы алгоритма. Рассмотрим процесс p , удаленный от лидера на расстояние $d + 1$. Коль скоро все предшественники процесса p в конце концов сумели на первом этапе отправить сообщения процессу p , одно из этих сообщений рано или поздно будет доставлено процессу p , и тогда этот процесс выполнит присваивание $\text{dis}_p := d + 1$. Когда на первом этапе процесс p получит сообщения от всех своих $d + 1$ предшественников, он сам отправит своим последователям сообщения, которые нужно распространять на первом этапе.

Подобным же образом устанавливается, что все процессы отправляют свои сообщения на втором этапе и завершают работу. $\square$

Лемма 11.22. *Разметка π , образовавшаяся по завершении работы алгоритма, является ориентацией. Для любой пары соседних процессов p и q , соединенных каналом l , наборы lbl_p и lbl_q отличаются в точности в одной позиции с номером $\pi_p[l]$ (который также равен $\pi_q[l]$).*

Доказательство. По теореме 11.20 существует единственная ориентация $\mathcal{L}$ и такая свидетельствующая разметка вершин $\mathcal{N}$, что $\mathcal{L}_w(p) = \mathcal{P}_w(p)$ и $\mathcal{N}(w) = (0, \dots, 0)$.

На первом этапе все процессы вычисляют разметку вершин $\mathcal{N}$, и в этом можно удостовериться, воспользовавшись индукцией по расстоянию от лидера. В вершине w значение lbl_w полагается равным набору $(0, \dots, 0)$, который и составляет значение $\mathcal{N}(w)$. Сосед p процесса w строит набор lbl_p , полагая b_i равным 1, если канал, соединяющий w и p , помечен номером i в процессе w ; в противном случае b_i полагается равным 0. Таким образом, $\text{lbl}_p = \mathcal{N}(p)$.

Предположим теперь, что во всех вершинах q , находящихся на расстоянии d от w , вычисляется $\text{lbl}_q = \mathcal{N}(q)$. Рассмотрим вершину p , находящуюся на расстоянии $d + 1$ от w . Набор $\mathcal{N}(p)$ содержит $d + 1$ единиц и $n - d - 1$ нулей. У

вершины p есть $d + 1$ предшественников, и набор $\mathcal{N}(q)$ для предшественника q отличается от $\mathcal{N}(p)$ только тем, что в одной из позиций вместо 1 стоит 0. Поэтому покомпонентная дизъюнкция $d + 1$ пометок $\mathcal{N}(q)$ действительно дает набор $\mathcal{N}(p)$.

После первого этапа для предшественника q процесса p , соединенного с ним каналом l , выполняется равенство $\text{nei}_p[l] = \text{lbl}_q$. На втором этапе процесс p вычисляет значение $\pi_p[l]$, полагая его равным номеру той позиции, в которой отличаются наборы lbl_p и $\text{nei}_p[l]$, и поэтому lbl_p и lbl_q отличаются только в позиции $\pi_p[l]$. Номер этой позиции используется процессом q для пометки канала связи с p , после того как q получит от p сообщение $\langle \text{labl}, \pi_p[l] \rangle$. $\square$

Подведем итоги. Алгоритм 11.6/11.7 ориентирует гиперкуб с выделенным лидером, совершая $2 \cdot |E|$ обменов сообщениями. Сложность алгоритма по числу обменов сообщениями асимптотически оптимальна (см. следствие 11.27).

11.3.4. Эффективные алгоритмы для гиперкубов на основе трафаретов

Алгоритм 11.6/11.7 можно использовать для решения других задач на помеченных гиперкубах (см. [67]) и при этом можно даже сократить число обменов сообщениями, избавив некоторые процессы от необходимости отправлять сообщения. Это осуществляется при помощи *трафаретов*, представляющих собой подмножества двоичных наборов длины n . После вычисления пометки своей вершины процесс будет проверять, не содержится ли эта пометка в трафарете (считается, что трафарет определен заранее и доступен всем процессам). Процессор займется рассылкой сообщений *в том и только том случае*, если пометка содержится в трафарете, точно так же, как это описано в алгоритме 11.6/11.7.

Истолкование трафаретного алгоритма. Далее мы будем считать, что M является подмножеством $\mathbb{Z}_2^n$, т. е. некоторым множеством двоичных наборов длины n , и мы будем обозначать символом M_k совокупность всех строк множества M , в которых содержится в точности k единиц. Вполне понятно, что трафарет должен удовлетворять некоторым требованиям, и мы еще должны внести дополнительные изменения в сам алгоритм, но эти требования мы разъясним прямо сейчас. Во-первых, строка $00 \dots 00$ будет всегда содержаться в M , и соседи инициатора будут способны вычислить свои пометки, получив сообщение $\langle \text{dmn}, i \rangle$. Другие вершины трафарета должны быть способны вычислить свои пометки, получив сообщения по меньшей мере от двух предшественников. Таким образом, для строки $v \in M_k$ (где $k \geq 2$) в множестве M_{k-1} должны быть хотя бы два предшественника x и y .

Поясним теперь техническую подробность: так как x и y — предшественники вершины v , вершина v является общим последователем для x и y . Но x и y также имеют и общего предшественника, скажем, w . Существуют в точности два процесса, которые получают сообщения как от x , так и от y , и этими процессами являются v и w . Мы потребуем, чтобы общий предшественник процессов x и y вычислял свою пометку, *до того как x и y отправят сообщения*, исходя из предположения, что w также содержится в трафарете.

Определение 11.23. Трафаретом называется подмножество $M \subseteq \mathbb{Z}_2^n$, удовлетворяющее следующим требованиям:

- 1) $0^n \in M$;
- 2) для всякого набора $v \in M_k$, где $k \geq 2$, существуют наборы $x, y \in M_{k-1}$, являющиеся предшественниками набора v ;
- 3) Если набор v и два его предшественника x и y содержатся в M , то общий предшественник наборов x и y также содержится в M .

Пусть задан (общезвестный) трафарет M . Тогда алгоритм работает следующим образом.

1. Инициатор отправляет сообщение $\langle \mathbf{dmn}, i \rangle$ по такому каналу i , что $00 \dots 1 \dots 00 \in M$ (единица занимает i -ю позицию в наборе).

2. Процессор, получивший это сообщение, вычисляет свою пометку $lbl_p = 00 \dots 1 \dots 00$ и (учитывая, что эта пометка содержится в M) отправляет сообщение $\langle \mathbf{iam}, lbl_p \rangle$ по всем остальным каналам.

3. После получения *двух* сообщений $\langle \mathbf{iam}, l \rangle$, в которых содержатся строки с одинаковым количеством единиц, процесс может вычислить пометку v , которая является общим последователем этих двух строк.

Если эта пометка *не содержится* в M , то все последующие сообщения игнорируются.

Если эта пометка содержится в M , то следует ожидать поступления подтверждения $\langle \mathbf{iam}, l \rangle$ от всех предшественников набора v , которые содержатся в M , а затем отправить сообщение $\langle \mathbf{iam}, v \rangle$ по всем каналам.

Отметим, что описанный алгоритм вычисляет меньше информации, чем «полный» алгоритм 11.6/11.7, проводящий ориентацию гиперкуба: новый алгоритм вычисляет только такие пометки процессов, которые содержатся в M , и вычисляет пометки каналов только в том случае, когда оба процесса, которые сообщаются по этому каналу, помечены наборами из M . Но в нем также используется и меньшее число обменов сообщениями; оно ограничено величиной $n \cdot |M|$, так как лишь процессы, помеченные наборами из M , отправляют сообщения и каждый из них может отправить не более n сообщений.

Маленькие трафареты для отправления сообщений и широковещательной рассылки. Теперь мы станем свидетелями того же эффекта, который нам довелось наблюдать в § 11.2.3: настраивая наш алгоритм, мы не будем заниматься его алгоритмическими особенностями, а вместо этого выполним упражнение по комбинаторике, связанное со свойствами трафаретов.

Первой задачей, которой мы займемся, станет задача *FarSend* отправления инициатором сообщения в заданную вершину, отстоящую от него на расстояние d (не ограничивая общности, мы будем полагать, что она помечена набором $1^d 0^{n-d}$). Этот адрес следует истолковывать с точки зрения «частного» восприятия направления, свойственного инициатору, или, иначе говоря, исходя из локальной разметки каналов, примыкающих к инициатору, которая вовсе не обязана совпадать с ориентацией других вершин. Напоминаем, что вначале ни одна вершина,

кроме инициатора, не осведомлена ни о своей пометке, ни о пометках примыкающих к ней каналов!

Лемма 11.24. Существует такой трафарет размера $1 + \frac{d(d+1)}{2}$, который содержит набор $1^d 0^{n-d}$.

Доказательство. Множество наборов $M = \{0^i 1^k 0^{n-i-k} : i+k \leq d\}$ удовлетворяет всем предъявленным к нему требованиям. $\square$

В результате, задача *FarSend* обойдется в всего лишь в $O(d^2 n)$ обменов сообщениями на неразмеченных гиперкубах. В работе [67] было установлено, что размер этого трафарета оптимален. Обозначим трафарет для задачи *FarSend* отправления сообщения в вершину, помеченную набором v , символом F_v .

Задачу широковещательной рассылки сообщений можно было бы решить с любым трафаретом, который *покрывает* всю сеть так, чтобы каждая вершина либо содержалась в трафарете, либо имела в трафарете смежную с ней вершину. Однако для того чтобы добиться линейной сложности, трафарет должен содержать не более $O(N/n)$ вершин, а нам не известно, существует ли покрывающий трафарет такого размера. Поэтому в алгоритме широковещательной рассылки сообщений используется процедура *FarSend*, для того чтобы переправить сообщение в некоторое разреженное множество выделенных вершин, из которых оно затем распространяется потоком на ограниченное число переходов.

Так как для доставки сообщения в заданную точку посредством процедуры *FarSend* требуется совершить $O(n^3)$ обменов сообщениями, количество таких «вершин-глашатаев» в нашем алгоритме ограничено величиной N/n^3 .

Предложение 11.25. Существует такое множество $C \subseteq \mathbb{Z}_2^n$, что

- 1) $|C| = O(\frac{N}{n^3})$;
- 2) для каждого набора $x \in \mathbb{Z}_2^n$ выполняется неравенство $d(x, C) \leq 3$.

В алгоритме широковещательной рассылки сообщений используется именно то множество, которое описано в данном утверждении; элементы этого множества будут называться *глашатаи*. На первом этапе каждый глашатай получает сообщение от инициатора посредством процедуры *FarSend*. На втором этапе каждый глашатай распространяет поток сообщений на расстояние 3, т.е. отправляет сообщение каждому из своих соседей, те отправляют сообщение своим соседям, а последние отправляют сообщение своим соседям еще на один переход дальше.

Заметим, что вследствие второго свойства множества C каждая вершина получит сообщение на втором этапе работы алгоритма. Теперь разберемся со сложностью алгоритма. Чтобы достичь одного глашатая, нужно провести $O(n^3)$ обменов сообщениями. С учетом размера множества C это означает, что число обменов сообщениями, которые проводятся на первом этапе, ограничено линейной функцией. Так как у каждого глашатая есть n соседей, у которых есть (менее чем) n^2 своих соседей и из каждой такой вершины на втором этапе отправляется n сообщений, число обменов сообщениями, которые проводятся на втором этапе, также ограничено линейной функцией.

Еще одно приложение: FarOrient. Трафаретный алгоритм можно применять для ориентации одной вершины относительно другой; предположим, что нам нужно, чтобы вершина v провела ориентацию выделенной вершины, удаленной от нее на расстояние d . Это значит, что выделенная вершина должна провести такую повторную разметку своих дуг, чтобы новая разметка оказалась согласованной с разметкой дуг вершины v . Вершина u запускает трафаретный алгоритм с трафаретом M , в котором представлены все соседи вершины v . Так как процесс v получит сообщение $\langle \text{iam}, l \rangle$ по всем примыкающим к нему каналам, он сможет вычислить направление каждого канала. В статье [67] было показано, что для этого существует оптимальный трафарет, размер которого близок к величине $n \cdot d$. Следовательно, задачу *FarOrient* можно решить, проведя $O(n^3)$ обменов сообщениями.

11.3.5. Выборы в неразмеченных гиперкубах

В основу наиболее эффективного алгоритма избрания лидера в гиперкубах заложен принцип слияния деревьев в том виде, в котором он используется в алгоритме Галладжера—Хамблета—Спиры (§ 7.3.2). Для уменьшения его сложности можно воспользоваться топологическими свойствами гиперкуба. В этом параграфе мы попробуем дать описание наиболее значимых методов, но полное описание такого алгоритма здесь не приводится (см. [68]).

Напомним, что сложность по числу обменов сообщениями алгоритма GHS составляет величину $\Theta(N \log N + m)$, и это оптимальная сложность (если отсутствует осведомленность о топологии сети). Два слагаемых в приведенной формуле отражают два разных аспекта этого алгоритма. Одной из составных частей алгоритма является поиск дуги, которая выводит за пределы текущего поддерева: в алгоритме для каждой дуги проверяется возможность слияния с ее помощью с другим поддеревом. Число успешных попыток слияния ограничено линейной функцией, тогда как в большинстве случаев обнаруживается, что дуга приводит к вершине, относящейся к тому же самому поддереву. Вот эти *неудачные* попытки и образуют слагаемое m в формуле оценки сложности алгоритма. Другая часть алгоритма занимается слиянием двух поддеревьев размера n_1 и n_2 в единое поддерево размера $n_1 + n_2$. В алгоритме GHS проводится изменение разметки меньшего дерева, и за счет этого осуществляется слияние. Для изменения разметки требуется $\min(n_1, n_2)$ обменов сообщениями, и общие затраты на проведение всех слияний оцениваются величиной $\Theta(N \log N)$.

Чтобы уменьшить затраты на проведение выборов, мы должны ограничить количество дуг, проверка которых не приносит удачи, а также сократить затраты на слияние поддеревьев. На первом этапе работы алгоритма Добрева в вершинах начинается построение поддерева, высота которого ограничена константой 5, методом поиска в ширину. Особенности топологии гиперкуба позволяют положить предел неудачным попыткам поиска подходящей дуги. Действительно, у вершины, находящейся на расстоянии 5 от корня, есть в точности пять соседей на четвертом ярусе дерева, и поэтому она будет «обнаружена» в точности пять раз. Таким образом, BFS-деревья высоты 5 можно построить с приемлемыми затратами.

Если дерево содержит все вершины, расположенные на расстоянии 5 и менее от некоторого центра, можно воспользоваться другой стратегией расширения. Так как у нас имеется не более $O(N/n^5)$ деревьев, вполне приемлемо затрачивать n^5 обменов сообщениями на одно слияние. В процессе, расположенном в центральной точке дерева, само дерево представлено списком относительных позиций, но выраженных с точки зрения «частного» восприятия направления, т. е. того восприятия направления, которое образуется в результате распространения локальной разметки дуг, примыкающих к центру дерева, на весь граф.

Если все дерево представлено в таком формате, то процесс в его центре может выбирать ту точку, в которой это дерево должно быть расширено: эта точка должна быть удалена на расстояние 5 от дерева. Если таких точек нет, центральная точка дерева становится лидером; из условий выбора следует, что нигде в гиперкубе нет шара радиуса 5, который принадлежал бы другому поддереву. Проход к выбранной точке прокладывает процедура *FarOrient*, и уже из этой точки дерево может быть расширено либо за счет применения алгоритма BFS для пяти ярусов, либо за счет слияния с другим поддеревом.

Чтобы провести слияние двух деревьев, список позиций, который содержится в центре одного из деревьев, адаптируется к восприятию направления, свойственному центральной точке другого дерева; это можно сделать при помощи процедуры *FarOrient*, которая устанавливает согласованную ориентацию пары вершин друг относительно друга. Полученный таким образом алгоритм совсем непросто; его анализ до сих пор является предметом исследования.

11.4. Сложностные аспекты

Основные темы исследований, связанные с восприятием направления и осведомленностью о топологии, были разработаны в предыдущих параграфах. В этом параграфе мы рассмотрим ряд других, менее взаимосвязанных тем, которые играют важную роль в современных исследованиях.

11.4.1. Ориентация клики или произвольного графа

Убедившись в том, что восприятие направления позволяет уменьшить сложность некоторых задач, можно задаться следующим вопросом: возможно ли ввести восприятие направления в тех сетях, где оно не задано. Рискуя внести путаницу, связанную с тем, что предложенное название может быть отнесено к теоретико-графовым операциям, устанавливающим направление на ребрах неориентированного графа, мы тем не менее назовем задачу определения восприятия направления задачей *ориентации* сети. В этом параграфе мы обоснуем нижнюю оценку сложности по числу обменов сообщениями решения задачи ориентации графа и представим несколько алгоритмов ориентации сети.

Нижняя оценка сложности задачи ориентации. Поскольку ориентация сети предполагает нанесение пометки на каждую дугу, для этого требуется воспользоваться каждой дугой, о чем свидетельствует следующая теорема.

Теорема 11.26. При выполнении всякого алгоритма ориентации происходит обмен по меньшей мере $t - \frac{1}{2}N$ сообщениями.

Доказательство. Пусть задана разметка дуг $\mathcal{P}$. Тогда запись $\mathcal{P}^{u,v,\omega}$ (где v и ω — это соседи вершины u) будет обозначать разметку, которая задается следующими соотношениями: $\mathcal{P}_u^{u,v,\omega}(v) = \mathcal{P}_u(\omega)$, $\mathcal{P}_u^{u,v,\omega}(\omega) = \mathcal{P}_u(v)$, а все прочие пометки наносятся разметкой $\mathcal{P}^{u,v,\omega}$ так же, как они наносятся разметкой $\mathcal{P}$. (Иначе говоря, $\mathcal{P}^{u,v,\omega}$ получается из $\mathcal{P}$, если поменять местами $\mathcal{P}_u(\omega)$ и $\mathcal{P}_u(v)$.) Если $\mathcal{L}$ представляет собой ориентацию, то $\mathcal{L}_u[v] = -\mathcal{L}_v[u]$, но при этом значение $\mathcal{L}_u^{u,v,\omega}[v]$ будет уже другим. Поэтому $\mathcal{L}^{u,v,\omega}$ не является ориентацией.

Рассмотрим выполнение алгоритма построения ориентации, начинающего работу с разметки $\mathcal{P}$ и завершающего работу с перестановкой π_v для каждой вершины (где $\mathcal{L} = \pi(\mathcal{P})$ представляет собой ориентацию). Предположим, что в ходе этого выполнения некоторая вершина u не отправляла и не получала никаких сообщений от двух своих соседей v и ω . Так как вершина u не выходила на связь ни с v , ни с ω , точно такое же выполнение алгоритма возможно и тогда, когда первоначальная разметка имела вид $\mathcal{P}^{u,v,\omega}$. При этом все процессы вычислят ту же самую перестановку. Однако $\mathcal{L}' = \pi(\mathcal{P}^{u,v,\omega}) = \mathcal{L}^{u,v,\omega}$ не образует ориентации, и поэтому алгоритм некорректен.

Значит, при любом выполнении каждая вершина должна выйти на связь со всеми своими соседями, кроме, быть может, одного. $\square$

Следствие 11.27. Для проведения ориентации клики на N вершинах требуется совершить $\Omega(N^2)$ обменов сообщениями. Для проведения ориентации n -мерного гиперкуба требуется совершить $\Omega(n \cdot 2^n)$ обменов сообщениями. Для проведения ориентации $(n \times n)$ -тора требуется совершить $\Omega(n^2)$ обменов сообщениями.

Проведение ориентации клики. В алгоритме ориентации клики (алгоритм 11.8) каждому процессу присваивается уникальный номер из множества $0, \dots, N-1$; каждый процесс вычисляет свой ранг на множестве всех имен, после того как ему доставят имена всех его соседей. Ранг каждого соседа вычисляется аналогично, и в качестве пометки канала берется разность ранга процесса и ранга его соседа.

В алгоритме существенно используется тот факт, что все вершины имеют разные отличительные признаки. Но если в сети есть лидер, то алгоритм можно приспособить и для этого случая. Тогда работа будет начинаться с того, что лидер отправит всем своим соседям номера начиная с 1 и оканчивая $N-1$. Эти номера можно будет использовать в качестве отличительных признаков. Раздача номеров обходится дешево (N сообщений) по сравнению с вычислением разметки каналов, ведь согласно теореме 11.26 для этого потребуется совершить $\Omega(n^2)$ обменов сообщениями.

Хордовое восприятие направления в произвольных сетях. В произвольной сети вершины можно перенумеровать при помощи маркера, который будет совершать обход сети, используя, например, стратегию обхода в глубину. Но сделать это можно лишь при условии, что одна из вершин будет выделена в качестве

```

begin for  $l = 1$  to  $N - 1$  do send  $\langle \text{name}, p \rangle$  via  $l$ ;
    num_rec_p := 0;
    while num_rec_p <  $N - 1$  do
        begin receive  $\langle \text{name}, n \rangle$  по каналу  $l$ ;
            num_rec_p := num_rec_p + 1; nei_p[l] :=  $n$ 
        end;
    (* Вычисление пометки вершины *)
    lbl_p :=  $\#\{k : \text{nei}_p[k] < p\}$ ;
    for  $l = 1$  to  $N - 1$  do
        begin (* Вычисление пометки соседа и пометки канала связи *)
            ll :=  $\#\{k : \text{nei}_p[k] < \text{nei}_p[l]\}$ ;
            if  $p < \text{nei}_p[l]$  then ll := ll + 1;
             $\pi_p[l] := (ll - \text{lbl}_p) \bmod N$ 
        end
    end
end

```

Алгоритм 11.8. Ориентация клик

инициатора. Для совершения обхода нужно совершить $2m$ обменов сообщениями (по одному сообщению на каждое из двух направлений в каждом канале связи), и всем вершинам будет присвоен индивидуальный номер в зависимости от порядка, в котором маркер посещает эти вершины. При таком обходе каждый процесс узнает не только свой собственный номер, но и номера всех соседей. В результате процессы могут вычислить хордовое восприятие направления. Но для этого нужно знать количество вершин сети. Если это число заранее не известно, то его можно вычислить при обходе, а затем поручить инициатору разослать это число широковещательно. Это обойдется в $N-1$ дополнительных обменов сообщениями.

11.4.2. Битовая сложность и алгоритм потока по многим путям

Теперь мы перейдем к более подробным оценкам сложности и рассмотрим битовую сложность задачи ориентации на гиперкубе. В теореме 11.26 уже была установлена оценка числа обменов сообщениями, но ее можно распространить и на битовую сложность.

Лемма 11.28. В ходе проведения ориентации n -мерного гиперкуба каждая вершина отправляет по каналам связи по меньшей мере $\Omega(n \log n)$ битов.

Доказательство. Как уже было отмечено, всякая вершина должна отправить не менее $n-1$ сообщений, чтобы уметь различать n примыкающих к ней каналов связи. Чтобы можно было различать эти каналы по-настоящему, все эти сообщения должны быть попарно различными. А это значит, что каждое из них должно содержать, по меньшей мере, $\Omega(\log n)$ битов. $\square$

В сообщениях, используемых в алгоритме потока по многим путям (§ 11.3.3), содержатся пометки вершин, занимающие n битов. Сейчас мы покажем, что этот алгоритм можно так реализовать, что понадобятся лишь сообщения, содержащие $O(\log n)$ битов.

В алгоритме не нужна вся информация, содержащаяся в сообщении $\langle iam, label \rangle$; это сообщение слишком избыточно. Достаточно всего лишь передать количество единиц в пометке, наименьший номер позиции, которую занимает единица, и суммарно по модулю n номеров всех тех позиций, в которых стоят единицы. Для вершины, помеченной набором $label$, определим ее параметры wgt (вес), low (нижний индекс) и ixs (суммарный индекс) следующими соотношениями: $wgt(label) = \# \{i : b_i = 1\}$; $low(label) = \min \{i : b_i = 1\}$; $ixs(label) = (\sum_{b_i=1} i) \bmod n$. Набор $smv(label) = (wgt(label), low(label), ixs(label))$ будем называть *сводкой*. Сводкой вершины будем называть сводку ее пометки.

Лемма 11.29. Пусть задана вершина p на расстоянии $d + 1 \geq 2$ от w . Тогда

- 1) расстояние $dis_p = d + 1$ можно извлечь из сводки одного из предшественников вершины p ;
- 2) сводку p можно вычислить на основе $d + 1$ сводок предшественников вершины p ;
- 3) пометку вершины p можно вычислить на основе сводки самой вершины p и $d + 1$ сводок предшественников вершины p ;
- 4) пометку предшественника q вершины p можно вычислить на основе пометки вершины p и сводки вершины q .

Доказательство.

1) Вычислить расстояние dis_p нетрудно, так как значение $wgt(\mathcal{N}(q))$ равно $d(q, w)$.

2) Пусть заданы $d + 1$ сводок предшественников вершины p . Тогда в d из $d + 1$ сводки значение low равно $low(\mathcal{N}(p))$, а в одной сводке значение параметра low будет превосходить $low(\mathcal{N}(p))$ (это сводка того предшественника, пометку которого можно получить из $\mathcal{N}(p)$ путем замены *первой* по порядку единицы в наборе на нуль). Это дает значение $low(\mathcal{N}(p))$, а также позволяет определить суммарный индекс ixs_0 пометки той вершины, которая отличается от $\mathcal{N}(p)$ в позиции low . Таким образом, $ixs(\mathcal{N}(p)) = (ixs_0 + low(\mathcal{N}(p))) \bmod n$. Этим завершается вычисление набора $smv(\mathcal{N}(p))$.

3) Те $d + 1$ позиций, которые заняты единицами в наборе $\mathcal{N}(p)$ можно вычислить по формуле $ixs(\mathcal{N}(p)) - ixs(\mathcal{N}(q)) \bmod n$, перебрав всех $d + 1$ предшественников q вершины p .

4) Для всякого предшественника q вершины p , пометка $\mathcal{N}(q)$ образуется заменой единицы на нуль в том разряде пометки $\mathcal{N}(p)$, номер которого равен $(ixs(\mathcal{N}(p)) - ixs(\mathcal{N}(q))) \bmod n$. $\square$

Из леммы 11.29 следует, что алгоритму построения ориентации достаточно отправлять сводку пометки вершины вместо всей пометки, и поэтому этот алгоритм можно так реализовать, что битовая сложность каждого сообщения составит величину $O(\log n)$.

11.4.3. Алгоритм случайного блуждания Вервейджа

Теперь мы приведем рандомизованный алгоритм класса Монте-Карло для решения задачи о выборах; этот алгоритм был предложен в работе Вервейджа [197]. Он работает на сетях произвольной топологии и *не использует* восприятия направления, поскольку в основу его положено параллельное случайное блуждание.

Первоначально каждый процесс пребывает в состоянии *sleep*. После того как он переходит в состояние *awake* (а это происходит либо спонтанно, либо в результате получения сообщения в состоянии *sleep*), процесс запускает маркер, в который помещает свой отличительный признак и счетчик переходов. Маркеры совершают *случайные блуждания* по сети и изымаются из обращения теми процессами, которым уже к тому моменту нанес визит маркер с более старшим опознавательным признаком. Если маркер совершил K шагов и не был изъят из обращения, то последний процесс, которого он достиг, избирается лидером (см. алгоритм 11.9).

cons K (* Параметр безопасности *)

var $state : (sleep, awake, leader)$ **init** $sleep$;
 $maxid_p$; (* Отличительный признак,
наибольший из увиденных *)

Чтобы начать выборы:

```

begin  $i := p$  ;  $s := 0$  ;  $maxid_p := i$  ;
    if  $s < K$ 
    then send  $\langle walk, i, s + 1 \rangle$  по случайно выбранному ребру
    else  $state_p := leader$ 
end

```

Если получен маркер $\langle walk, i, s + 1 \rangle$:

```

begin if  $state_p = sleep$  then начать выборы;
    receive  $\langle walk, i, s \rangle$  ;  $maxid_p := \max(maxid_p, i)$  ;
    if  $i = maxid_p$ 
    then if  $s < K$ 
    then send  $\langle walk, i, s + 1 \rangle$  по случайно выбранному ребру
    else  $state_p := leader$ 
    else (* Изъять маркер *) skip
end

```

Алгоритм 11.9. Алгоритм избрания лидера (для процесса p)

Как же это работает? Маркер, несущий самый старший отличительный признак, нельзя изъять. Следовательно, этот маркер совершит K шагов и приведет к избранию лидером один из процессов; назовем это избрание *притязанием настоящего лидера*. Это означает, что алгоритм потерпит неудачу лишь в том

случае, когда будет избрано несколько лидеров, но не тогда, когда ни одного лидера *вообще не будет* избрано.

Алгоритм терпит неудачу (т. е. избирает более одного лидера), если какой-либо маркер, не являющийся максимальным, успешно совершает K переходов, не посетив ни один процесс, в котором побывал более старший маркер. Притязания такого лидера назовем *лживыми*. Вероятность возникновения лживых притязаний убывает с ростом значения K , но, поскольку маркеры совершают не менее K переходов, с увеличением значения K возрастает и сложность по числу обменов сообщениями.

Если $K = 0$, то каждый запущенный маркер²⁾ приводит к лживым притязаниям. Таким образом, вероятность неудачного исхода велика (1), хотя сложность по числу обменов сообщениями очень мала (0). С другой стороны, если K стремится к ∞ , то вероятность ошибки стремится к 0, поскольку не может быть положительной величиной вероятность того, что какой-нибудь маркер, кроме самого старшего, способен совершить бесконечно много шагов, избегнув при этом посещения процесса, в котором побывал маркер с более высоким отличительным признаком.

К счастью выбор значения K не очень критичен для правильной работы алгоритма или для его сложности. Во-первых, даже для малых значений K сложность по числу обменов сообщениями оценивается снизу функцией, линейно зависящей от N , так как каждый процесс может запустить свой маркер. А во-вторых, выбрав значение K большим, чем это необходимо для того, чтобы добиться желаемой надежности, мы приходим к тому, что маркеры будут совершать лишние переходы, но при этом будет высока вероятность того, что эти переходы будет делать лишь один старший маркер! Из проведенного обсуждения следует, что асимптотика математического ожидания числа обменов сообщениями не зависит от K , как только она станет превосходить минимальное значение на некоторую величину, линейно растущую с ростом N .

Математический анализ сложности алгоритма и необходимого значения параметра K неизвестен; это остается открытой проблемой. Далее мы вкратце поясним, в чем состоит трудность и почему полный конкурентный анализ возможен лишь при наличии допущений о согласовании времени.

Допущения о согласовании времени. Оптимальное значение K и производительность алгоритма зависят от относительной скорости движения маркеров. Мы обсудим это для случая, когда сеть представляет собой клику, т. е. каждый переход маркера приводит его в случайно выбранную вершину.

В одном из предельных вариантов самый старший маркер движется бесконечно медленно по сравнению со вторым по старшинству маркером. Тогда второй маркер будет изъят только в том случае, когда попадет в самый старший процесс. В случае клики, для того чтобы это произошло с вероятностью, близкой к 1, необходимо, чтобы значение K возрастало линейно с увеличением размера клики.

²⁾За исключением того, у которого действительно самый старший отличительный признак. — Прим.перев.

Если рассматривать сложность по числу обменов сообщениями, то наихудшим будет такое согласование времени, когда всякий маркер достается вершине лишь после того, как все маркеры с более младшими отличительными признаками уже изъятые. Тогда каждый маркер изымается только после того, как он достается процессу с более старшим отличительным признаком. При этом ожидаемое число шагов, которое придется сделать маркеру с $(i + 1)$ -м по старшинству отличительным признаком, составляет величину $\Omega(N/i)$. Отсюда следует, что для вполне асинхронной клики сложность по числу обменов сообщениями составит величину $\Omega(N \log N)$.

С другой стороны, рассмотрим случай, когда наша клика синхронизована и к любому моменту времени каждый из уцелевших маркеров совершил одинаковое число шагов. После $O(\sqrt{N})$ шагов самый старший маркер уже посетил $O(\sqrt{N})$ процессов, а после следующих $O(\sqrt{N})$ шагов второй маркер посетит один из этих процессов с вероятностью, близкой к 1. Значит, чтобы изъять второй маркер с высокой вероятностью, достаточно, чтобы выполнялось условие $K = O(\sqrt{N})$, и вероятность лживых притязаний на лидерство младшими маркерами будет совсем мала.

Экспериментальный анализ сложности. Проводилось моделирование рассматриваемого алгоритма с целью экспериментального установления его сложности (см. [197]) на кликах и гиперкубах, и было обнаружено, что сложность по числу обменов сообщениями составляет величину, близкую к $3N$. Как видно, эта величина значительно лучше той сложности, в пользу которой мы приводили доводы в ходе обсуждения вполне асинхронного случая. Это означает, что для получения аналитических оценок сложности должны быть приняты во внимание допущения о согласовании времени (например, допущение Архимеда).

11.5. Заключение и открытые вопросы

Мы убедились в том, что восприятие направления позволяет эффективно использовать особенности топологии сети и осуществлять более тесное взаимодействие между процессами. Изучение свойств восприятия направления является одной из тем ежегодного научного форума SIROCCO (Structural Information and Communication Complexity). Труды SIROCCO публикуются издательством Карлетонского университета.

11.5.1. Использование восприятия направления

Как было отмечено в § 11.1.2, восприятие направления полезно по нескольким причинам. Возможность сравнивать пути дает возможность обойти нижнюю оценку сложности $\Omega(m)$ для алгоритмов ширококестельной рассылки сообщений и проведения выборов, поскольку отпадает необходимость отправлять сообщения по дуге, чтобы проверить, не достиг ли уже алгоритм этой вершины. Сравнение путей открывает возможность для ширококестельной рассылки

сообщений с линейной сложностью и для проведения выборов со сложностью $O(N \log N)$ в произвольных сетях.

Метод сжатия путей дает возможность достигать по кратчайшим путям вершин, местоположение которых заранее известно. В § 11.6.2 показано, насколько это полезно для алгоритмов избрания лидера на хордовых кольцах, где степень сжатия пути можно регулировать за счет выбора различных размещений хорд. Было установлено, что сжатие путей может привести к алгоритмам избрания лидера, имеющим линейную сложность, даже в случае, когда в распоряжении имеется более чем скромное количество хорд. Для применения метода сжатия путей требуется, чтобы восприятие направления было равномерным.

В § 11.6.2 метод сжатия путей сравнивается также еще с одним новым качеством, которое открывает восприятие направления. Речь идет об использовании сведений о топологии сети, которые неявно содержатся в равномерном восприятии направления. Было показано, что если эти сведения использовать в алгоритме избрания лидера, то для проведения выборов с линейной сложностью потребуется еще меньше хорд.

11.5.2. Понижение сложности

Задача	Сеть	Нет SoD Нет TA	Нет SoD TA	SoD и TA	Улучшение из-за
ВС	Произвольное	m	(1)	N	Да: SD
	хорд. кольцо (2)	$c \cdot N$	неизвестна	N	Да (3)
	Тор	N	N	N	Нет
	Гиперкуб	$N \cdot n$	N	N	Да: TA
	Клика	N^2	N	N	Да: TA
ЕЛ	Произвольная	$N \log N + m$	(1)	$N \log N$	Да: SD
	хорд. Кольцо (2)	$N \log N$	N	N	Да: TA
	Тор	$N \log N$	N	N	Да: TA
	Гиперкуб	$N \log N$	??	N	Да: TA или SD
	Клика	N^2	$N \log N$	N	Да: TA+SD

Замечания:

- (1) Топологическая осведомленность для этого класса не нужна.
- (2) Величина c обозначает число хорд; предполагается, что $c \leq \log N$.
- (3) Неизвестно, можно ли добиться улучшения, используя только TA.

Таблица 11.10. Сведения о структуре и сложность

В таблице 11.10 представлена сводка многих открытий, о которых рассказывалось в этой главе. Здесь приведены оценки сложности по числу обменов сообщениями задачи широкоэвентальной рассылки сообщений и задачи о выборах для различных классов сетей в зависимости от допущений, которые принимаются во внимание в алгоритмах решения. Первая колонка соответствует случаю,

когда ни осведомленность о топологии, ни восприятие направления в расчет не принимаются. Ввиду того что алгоритм «не ведает» о том, в какой топологией он работает, для решения обеих задач требуется, чтобы сообщения были отправлены по каждой дуге. Вторая колонка соответствует случаю, когда предполагается осведомленность о топологии сети. Это качество позволяет алгоритмам использовать специальные свойства графа сети, но при этом направления дуг остаются неизвестными. В третьей колонке представлены оценки для случая, когда имеется как осведомленность о топологии, так и восприятие направления.

В самой правой колонке сообщается о том, верно ли, что информация о структуре сети способствует понижению сложности задачи. Как мы видим, во всех случаях, за исключением широкоэвентальной рассылки сообщений на торе (которую можно провести с линейной сложностью по числу обменов сообщениями, даже не зная о том, что сеть является тором), информация о структуре сети понижает сложность задачи по числу обменов сообщениями. В большинстве случаев, однако, понижение оказывается возможным за счет осведомленности о топологии сети. За вычетом сетей произвольной топологии, задача о выборах в кликах остается единственной задачей, для которой строго обосновано, что ориентация дуг приводит к понижению сложности по числу обменов сообщениями.

Если даже не принимать в расчет асимптотику сложности по числу обменов сообщениями, восприятие направления дает целый ряд других преимуществ: алгоритмы становятся проще, вспомогательные константы имеют меньшую величину, улучшается сложность по времени.

11.5.3. Современные исследования

Здесь приведены некоторые из актуальных направлений исследований в этой области.

1. Доказать леммы 11.11 и 11.14 для случая стратегии маршрутизации общего вида.
2. Для каких значений размаха хорды t можно адаптировать однохордовый алгоритм (см. § 11.2.4), сохраняя при этом линейную сложность по числу обменов сообщениями?
3. Оценить математическое ожидание сложности алгоритма выбора методом случайного блуждания (для клика, гиперкубов и торов).
4. Обобщить результаты этой главы для восприятий направления, основанных на некоммутативных группах. Такое обобщение представляло бы интерес, например, для кубических связанных циклических сетей, определяемых при помощи группы подстановок.

11.6. Упражнения к главе 11

11.6.1

Упражнение 11.1. Пусть $\mathcal{L}$ — это разметка дуг элементами группы G . Обозначим записью $SUM(P)$ сумму всех элементов группы, помечающих путь P . Докажите эквивалентность следующих двух утверждений:

1. $\mathcal{L}$ является восприятием направления;
2. путь P является циклом тогда и только тогда, когда $SUM(P) = 0$.

11.6.2

Курсовая работа 11.2. Покажите, что $\Omega(\log \log N)$ хорд необходимы в случае произвольной стратегии маршрутизации, чтобы сумма, фигурирующая в лемме 11.9, стала суммой геометрической прогрессии. (Сравните это утверждение с леммой 11.11.)

11.6.3

Упражнение 11.3. Какова сложность по времени алгоритма посредничества (§ 11.3.2)?

ГЛАВА 12

СИНХРОНИЗАЦИЯ В СЕТЯХ

В этой главе мы исследуем вопрос о том, какое влияние оказывает на теорию распределенных вычислений допущение о существовании глобального времени, к которому могут обращаться процессы. Структура глобального времени составляет часть той физической действительности, которая нас окружает; эта действительность включает в себя процессы распределенной системы, и использование свойств времени может принести пользу для проектирования распределенных систем. Результаты, представленные в этой главе, свидетельствуют о том, что, учитывая свойства времени, мы можем уменьшить коммуникационную сложность распределенных алгоритмов, если длительность обработки данных и передачи сообщений ограничена.

Однако проблемы, которые были неразрешимы, когда свойства времени не учитывались, остаются таковыми и после явного введения времени. Таким образом, введение синхронизации позволяет улучшить сложностные характеристики распределенных систем, но не расширяет их вычислительные возможности. Такое положение дел коренным образом отличается от ситуации, складывающейся в системах, подверженных отказам; здесь введение синхронизации приводит к настоящему прорыву в решении многих задач. В гл. 14 и 15 будет показано, что с использованием синхронизации мы получаем возможность обработать детерминированно гораздо больший класс неисправностей. Нам уже приходилось обращаться к синхронизации в § 3.3.2, чтобы справиться с отказами, возникающими при передаче данных.

В этой главе мы будем говорить о времени и почти всегда будем иметь дело с некоторой идеализированной моделью времени, когда работа всей сети разбивается на дискретные шаги, именуемые *тактами*; об этом пойдет речь в § 12.1.1. Каждый процесс выполняет локальные вычисления на протяжении каждого такта, и всякое сообщение, которое было отправлено в некотором такте, будет обязательно доставлено по назначению до начала следующего такта. Такую глобально синхронизованную операцию часто называют *жестко синхронизованной* операцией. Если есть доступ к часам и есть верхняя оценка задержки передачи сообщения, то жестко синхронизованную операцию довольно просто реализовать (см. § 12.1.3).

Сеть, работающую в режиме жесткой синхронизации, обычно называют *синхронной сетью*. Синхронную сеть не следует смешивать с сетью, в которой осуществляется синхронный обмен сообщениями, как это было определено в § 2.1.3. В последнем случае глобальная синхронизация отсутствует. И лишь только в тех случаях, когда два процесса осуществляют взаимосвязь, они действительно вре-

менно синхронизуют свои действия по передаче сообщения; но они не проводят синхронизации своих действий с другими процессами, и эта синхронность немедленно исчезает сразу после завершения передачи сообщения.

Глобально синхронизованные сети обладают способностью решать распределенные задачи, совершая меньшее число обменов сообщениями, чем вполне асинхронные сети. Чтобы продемонстрировать это, мы построим синхронный алгоритм для решения задачи избрания лидера и покажем, что сложность этого алгоритма оказывается меньше доказанной нами ранее нижней оценки сложности асинхронных алгоритмов, решающих ту же самую задачу (см. § 12.6.2). То, что синхронные алгоритмы не позволяют решать более широкий класс задач по сравнению с асинхронными алгоритмами, будет установлено за счет введения так называемых *алгоритмов-синхронизаторов*, при помощи которых синхронные сети моделируются асинхронными (см. § 12.6.3).

Чаще всего проектировать распределенные алгоритмы для синхронных сетей менее затруднительно, из-за того что здесь меньше неопределенных моментов, которые нужно принимать в расчет. Всякий асинхронный алгоритм можно спроектировать так: сначала разработать синхронный алгоритм, а затем применить к нему метод моделирования, описанный в § 12.6.3. Несмотря на издержки, вызванные введением синхронизатора, получившийся в результате алгоритм может, тем не менее, соперничать с алгоритмами, спроектированными непосредственно для асинхронных сетей. В § 12.6.4 мы рассмотрим в качестве примера применение синхронизаторов для построения деревьев поиска в ширину.

Даже если часы недоступны, некоторые слабые формы допущения о согласовании времени могут быть вполне оправданы. Например, можно полагать, что отношение быстродействий самого быстрого и самого медленного компонентов системы ограничено сверху. Такое допущение, известное под названием *допущение Архимеда* (см. [198], обсуждается вкратце в § 12.6.5. Мы приведем некоторые аргументы в пользу того, что лучше всего, вообще говоря, полагаться на вполне асинхронные алгоритмы и не использовать допущения о согласовании времени. Алгоритмы, разработанные в рамках допущения Архимеда, устроены так, что их корректность не зависит от допущений о согласовании времени, зато при этом возможен выигрыш в сложности, который обусловлен этими допущениями.

12.1. Начальные сведения

12.1.1. Синхронные сети

В *синхронных* распределенных системах функционирование каждого процесса складывается из последовательности (возможно, бесконечной) дискретных шагов, именуемых *тактами*. За один такой такт всякий процесс вначале отправляет несколько (ноль или более) сообщений, затем принимает несколько (ноль или более) сообщений и, наконец, выполняет локальные вычисления (изменяет свое состояние). Сообщения достигают адресатов в том же самом такте, в котором они были отправлены, т. е. если процесс p отправил сообщение процессу q в i -м такте своего функционирования, то это сообщение будет получено про-

цессом q в i -м такте его функционирования еще до того, как q выполнит свои вычисления в этом такте.

Такты можно представлять как импульсы (биения) глобальных часов. Вычисления выполняются по тактам, и сообщение, отправленное в одном такте, будет обязательно доставлено по назначению до начала следующего такта. Конфигурация системы после выполнения i тактов определяется набором, в котором представлены состояния всех процессов после выполнения i -го такта.

Чтобы формально определить синхронные системы, обозначим символом $\mathcal{M}$, как это было сделано в гл. 2, множество сообщений, а символом $\mathbb{M}$ — совокупность всех мультимножеств сообщений. Определение процесса в составе синхронной системы отлично от определения процесса, входящего в асинхронную систему, поскольку теперь отправлению и приему целого набора сообщений соответствует один-единственный переход системы. Чтобы избежать путаницы, мы будем называть процессы синхронной системы *синхронными процессами*.

Определение 12.1. *Синхронным процессом* называется четверка $p = (Z, I, \mathcal{MG}, \vdash)$, где

- 1) Z — множество состояний,
- 2) I — подмножество Z *начальных состояний*,
- 3) $\mathcal{MG}: Z \rightarrow \mathbb{M}$ — *функция генерации сообщений*, и
- 4) $\vdash$ — отношение переходов $Z \times \mathbb{M} \times Z$. Мы будем использовать запись $(c, M) \vdash d$ для обозначения $(c, M, d) \in \vdash$.

Всякий процесс начинает свое вычисление в состоянии $c_0 \in I$. Когда процесс начинает очередной такт работы в состоянии $c \in Z$, он отправляет набор сообщений $\mathcal{MG}(c)$. Еще до начала следующего такта процесс примет весь набор сообщений (обозначим его M), отправленных ему в текущем такте, и перейдет в состояние d , удовлетворяющее отношению $(c, M) \vdash d$.

Согласно приведенному определению набор сообщений, которые отправляет процесс, полностью определяется тем состоянием, в котором он пребывает в начале такта; весь внутренний недетерминизм процесса укладывается в отношение $\vdash$. Можно также ввести недетерминизм и в функцию $\mathcal{MG}$ (тогда это уже будет отношение, а не функция), но с равным успехом этот недетерминированный выбор можно возложить и на отношение $\vdash$.

Синхронной системой называется совокупность $\mathbb{P}$ синхронных процессов, а конфигурацией синхронной системы называется набор, состоящий из состояний всех ее процессов. Конфигурация, в которой каждый процесс находится в начальном состоянии, называется начальной конфигурацией. Изменение состояний всех процессов осуществляется синхронно за один глобальный переход всей системы, а изменение состояния каждого процесса происходит под влиянием того набора сообщений, которые были отправлены этому процессу в текущей конфигурации.

Определение 12.2. *Синхронной системой*, состоящей из совокупности (синхронных) процессов $\mathbb{P} = (p_1, \dots, p_N)$ (где каждый процесс p_i имеет вид $(Z_i, I_i, \mathcal{MG}_i, \vdash_i)$), называется такая система переходов $S = (\mathcal{C}, \rightarrow, \mathcal{I})$, что

- 1) $\mathcal{C} = \{(c_1, c_2, \dots, c_N) : \forall i, c_i \in Z_i\}$;

2) $\rightarrow$ обозначает отношение перехода, которое определяется следующим образом:

$(c_1, \dots, c_N) \rightarrow (d_1, \dots, d_N)$ тогда и только тогда, когда для каждого процесса $p_i \in \mathbb{P}$ выполняется соотношение

$$(c_i, \{m \in (\cup_{p_j \in \mathbb{P}} \mathcal{MG}_j(c_j)) : m \text{ адресовано процессу } p_i\}) \vdash_i d_i;$$

3) $\mathcal{I} = \{(c_1, c_2, \dots, c_N) : \forall i, c_i \in I_i\}$.

Синхронным вычислением системы называется такая максимальная последовательность конфигураций $\gamma_0, \gamma_1, \dots$, что ее первый член γ_0 является начальной конфигурацией и для всех $i \geq 0$ выполняется отношение $\gamma_i \rightarrow \gamma_{i+1}$. В качестве сложности по числу обменов сообщениями в таком вычислении будет выступать совокупное число сообщений, которые были отправлены процессами, а в качестве сложности по времени будет выступать индекс последней конфигурации (при условии, что вычисление оказалось конечным).

12.1.2. Повышение эффективности за счет синхронизации

На протяжении этой главы в тех случаях, когда это удобно, мы будем неявно предполагать, что состояния процессов, фигурирующих в рассматриваемых алгоритмах, включают в себя счетчик тактов, который отсчитывает количество тактов, выполненных процессом. В этом параграфе мы обсудим ряд парадигм, которые могут быть использованы в синхронных сетях, для того чтобы уменьшить коммуникационную сложность распределенных алгоритмов.

Кодирование с использованием времени. Можно сократить количество битов в сообщениях, которыми обмениваются процессы, до двух битов в каждом сообщении. Для этого содержимое сообщения будет «кодироваться» тем промежутком времени (количеством тактов), которое проходит между отправлением двух битов.

Предположим, что процесс p должен отправить сообщение процессу q . Здесь мы предполагаем, что содержимое сообщения можно представить в виде натурального числа m . Чтобы передать значение m , используются два сообщения, $\langle \text{start} \rangle$ и $\langle \text{stop} \rangle$. Процесс p отправляет сообщение $\langle \text{start} \rangle$ в том самом i -м такте, в котором инициируется передача сообщения, а сообщение $\langle \text{stop} \rangle$ отправляет в $(i + m)$ -м такте. Процесс q , получивший сообщения $\langle \text{start} \rangle$ и $\langle \text{stop} \rangle$, скажем, в тактах a и b соответственно, принимает тем самым сообщение (натуральное число $m = b - a$) в b -м такте. Кодирование при помощи времени приводит к тому, что на передачу сообщений расходуется много времени: ведь процесс q принимает сообщение спустя m тактов после того, как процесс p начал передачу. Передачу сообщения можно сделать более экономной по времени за счет отправления большего числа битов (см. упражнение 12.1).

Неявные сообщения. В асинхронных системах процесс может получать информацию от соседа только за счет получения от него какого-нибудь сообщения. Если же процесс в синхронной системе не получил сообщений от некоторого

соседа в некотором такте, то это означает, что сосед в этом такте не отправлял указанному процессу сообщений. В результате процесс получает некоторую информацию о состоянии, в котором находится его сосед.

Рассмотрим в качестве примера, алгоритм Туэга для задачи отыскания кратчайших путей между всеми парами вершин (алгоритм 4.6). На этапе обработки опорной вершины w каждый процесс должен выяснить, какие из его соседей являются его сыновними вершинами в дереве T_w . Для этого по каждому каналу проводится обмен сообщениями $\langle \text{ys}, w \rangle$ или $\langle \text{nys}, w \rangle$. В синхронных сетях необходимо отправлять только сообщения $\langle \text{ys}, w \rangle$; если от соседа не будет получено никакого сообщения (в том такте, когда проводится обмен такими сообщениями), то этот сосед не является сыновней вершиной. Сделав обмен сообщениями $\langle \text{nys}, w \rangle$ неявным, мы уменьшили сложность вычисления сыновних вершин для каждого процессора. Теперь вместо $2|E|$ обменов сообщениями требуется совершить не более $N - 1$ обменов. (Отправления сообщений $\langle \text{nys}, w \rangle$ можно избежать и в асинхронных системах, но за счет усложнения алгоритма и увеличения сложности по объему памяти.)

Избирательная задержка. Нам уже пришлось сталкиваться с распределенными алгоритмами, запускающими несколько альтернативных вычислений, одно из которых гарантирует получение желаемого результата, тогда как результат вычисления прочих альтернатив просто игнорируется. Примером тому служит алгоритм избрания лидера, который строится за счет применения принципа угасания (§ 7.3.1) к централизованному волновому алгоритму.

Чтобы понизить сложность по числу обменов сообщениями, можно уменьшить быстроту, с которой выполняется каждое такое альтернативное вычисление, и коэффициенты замедления могут быть разными. В асинхронных системах нельзя избежать (по крайней мере, в худшем случае) такой ситуации, когда безуспешное альтернативное вычисление проводится до полного завершения. Поэтому коммуникационная сложность становится равной произведению числа альтернативных вычислений и сложности каждого отдельного альтернативного вычисления. В случае синхронных систем число шагов, совершенных безуспешным альтернативным вычислением, ограничено временем выполнения успешной альтернативы, которое следует разделить на коэффициент замедления этого альтернативного вычисления. В § 12.2.2 мы займемся анализом избирательной задержки на основе тщательно отобранных коэффициентов замедления.

12.1.3. Асинхронные сети с ограниченной задержкой

Допущение, лежащее в основе синхронных сетей и гласящее, что все изменения состояний процессов глобально синхронизованы, является слишком строгим. Более умеренным было бы допущение о том, что процессы снабжены часами (не обязательно согласованными по времени), и время доставки сообщений ограничено сверху. Мы использовали такую модель в § 3.3.2, а после публикации работы Чоу и др. [52] сети такого вида стали называть *асинхронными сетями с ограниченной задержкой* или просто *ABD-сетями*.

Определение 12.3. Асинхронной сетью с ограниченной задержкой называется распределенная система, по отношению которой верны следующие допущения.

1. Осуществление события в процессе занимает нуль единиц времени.
2. Каждый процесс имеет часы для измерения физического времени; при этом если запись $CLOCK_p^{(t)}$ обозначает показание часов, принадлежащих процессу p в момент времени t , и процесс p не изменял показания своих часов в промежутке времени между моментами t_1 и t_2 , то

$$CLOCK_p^{(t_2)} - CLOCK_p^{(t_1)} = t_2 - t_1.$$

3. Физическое время между отправлением и приемом сообщения ограничено величиной μ , т. е. если сообщение было отправлено в момент времени σ и получено в момент времени τ , то

$$\sigma < \tau < \sigma + \mu.$$

Синхронную сеть совсем нетрудно реализовать на основе ABD-сети, и при этом издержки по времени и обменам сообщениями будут невелики. Тот механизм, при помощи которого реализуются синхронные сети, называется *синхронизатором* (см. также § 12.6.3). Главная трудность в проектировании синхронизатора состоит в том, чтобы обеспечить получение всех сообщений, отправленных в i -м такте, до того как произойдет изменение состояния и начнется следующий $(i + 1)$ -й такт.

ABD-синхронизаторы. Принцип работы синхронизатора для ABD-сетей состоит в том, что показания часов, принадлежащих процессам, согласуются настолько точно, насколько это возможно, скажем с точностью δ , и каждый процесс переходит к выполнению следующего такта работы спустя $\delta + \mu$ единиц времени по показаниям его часов. Алгоритмы сверки часов обсуждаются в § 15.4.3. Здесь мы приведем один простой результат, полученный в работе Теля и др. [191].

Работа ABD-синхронизатора Теля и др. складывается из двух этапов, а именно, из *этапа сверки часов* и *этапа моделирования*, которые в целом дают алгоритм 12.1. На этапе сверки часов каждый процесс устанавливает нулевое показание на своих часах и отправляет сообщение $\langle \text{start} \rangle$ каждому соседу. Согласно основному допущению, принятому для ABD-сетей, на сброс показаний и отправку сообщений $\langle \text{start} \rangle$ время не затрачивается. Всякий процесс выполняет эти действия либо при запуске алгоритма синхронизации (самопроизвольно), либо при получении первого сообщения $\langle \text{start} \rangle$. В результате выполнения этапа сверки часов часы соседних процессов будут согласованы с точностью μ .

Теорема 12.4. В каждый момент времени t после выполнения процедуры инициализации $init$ соседними процессами p и q показания часов, принадлежащих процессам p и q , будут удовлетворять неравенству

$$|CLOCK_p^{(t)} - CLOCK_q^{(t)}| < \mu.$$

Доказательство. Обозначим символом w_p (или w_q) момент времени, в который процессор p (или соответственно q) выполнил процедуру $init$. Если

```

var  $CLOCK_p$  : clock ;
     $state_p$  :  $Z_p$  ;
     $pulse_p$  : integer;

procedure  $init$ : (* Может быть выполнена самопроизвольно *)
begin if not  $started_p$  then
  begin  $CLOCK_p := 0$  ;
        forall  $q \in Neigh_p$  do send  $\langle \text{start} \rangle$  to  $q$ 
      end
  end
end

 $I_p$ : { Сообщение  $\langle \text{start} \rangle$  достигло процесса  $p$  }
begin receive  $\langle \text{start} \rangle$  ;  $init$  end

 $P_p^{(i)}$ : { Начало  $i$ -го такта работы процесса  $p$  }
when  $CLOCK_p = 2i\mu$  do
  begin (* Завершить предыдущий такт, вычислить  $state_p^{(i-1)}$  *)
    if  $i = 1$ 
    then  $state_p := \text{initial state}$ 
    else (*  $M$  — это набор  $(i - 1)$ -сообщений *)
         $state_p := d_p$  satisfying  $(state_p, M) \vdash d_p$  ;
         $pulse_p := i$  ;
        (* отправить сообщения, относящиеся к  $i$ -му такту *)
         $M' = MG(state_p)$  ; send all messages of  $M'$ 
      end
end

 $M_p$ : { доставлено основное  $i$ -сообщение }
begin receive and store the message end

```

Алгоритм 12.1. ABD-синхронизатор

процесс p отправил процессу q сообщение $\langle \text{start} \rangle$ в момент времени w_p и q выполнил $init$ позднее, когда это сообщение было ему доставлено, то справедливо неравенство $w_q < w_p + \mu$. Поменяв ролями процессы p и q в этом рассуждении, получаем неравенство $|w_p - w_q| < \mu$. Так как p и q не вносят изменений в показания своих часов после выполнения процедуры $init$, для всякого момента времени $t \geq \max(w_p, w_q)$ справедлива следующая цепочка равенств:

$$\begin{aligned}
|CLOCK_p^{(t)} - CLOCK_q^{(t)}| &= \\
&= |[CLOCK_p^{(w_p)} + (t - w_p)] - [CLOCK_q^{(w_q)} + (t - w_q)]| = \\
&= |[0 + (t - w_p)] - [0 + (t - w_q)]| = \\
&= |w_q - w_p| < \mu.
\end{aligned}$$

□

На этапе моделирования процессы выполняют те действия (изменения состояний), которые предписаны на каждом такте синхронным алгоритмом. Сообщения, которыми обмениваются процессы синхронного алгоритма на i -м такте,

будем называть i -сообщениями. Отметим, что i -й такт начинается, когда локальные часы покажут время $2i\mu$, после изменения состояния на основе полученных ранее $(i - 1)$ -сообщений. Приближенная сверка часов, а также верхняя оценка задержки передачи сообщения позволяют заключить, что все сообщения на каждом такте будут доставлены процессу, до того как этот процесс приступит к выполнению следующего такта, и это значит, что наша система ведет себя так, как должна вести себя синхронная система. Воспользуемся записью $state_p^{(i)}$ для обозначения состояния процесса p после $i + 1$ изменения состояния, т.е. после начала $(i + 1)$ -го такта, и пусть γ_i обозначает набор $(state_{p_1}^{(i)}, \dots, state_{p_N}^{(i)})$. Читатель должен иметь в виду, что этот набор не обязательно должен быть конфигурацией ABD-сети, так как процессы могут никогда не оказаться в указанных состояниях одновременно.

Теорема 12.5. *Последовательность $C = (\gamma_0, \gamma_1, \dots)$ является вычислением заданного синхронного алгоритма.*

Доказательство. Покажем, что последовательность $(\gamma_0, \gamma_1, \dots, \gamma_i)$ является префиксом вычисления, проведя индукцию по i . Для $i = 0$ достаточно заметить, что γ_0 является начальной конфигурацией синхронного алгоритма, потому что для каждого процесса p состояние $state_p^{(0)}$ является начальным состоянием процесса p в синхронном алгоритме.

Предположим, что последовательность $(\gamma_0, \gamma_1, \dots, \gamma_i)$ является префиксом некоторого вычисления синхронного алгоритма. Процесс p завершает $(i + 1)$ -й такт и переходит в состояние $state_p^{(i+1)}$, когда показания его часов равны $2(i + 2)\mu$, используя при этом множество $M_p^{(i+1)}$, состоящее из $(i + 1)$ -сообщений, полученных к этому времени. Чтобы убедиться в том, что конфигурация $(state_{p_1}^{(i+1)}, \dots, state_{p_N}^{(i+1)})$ продолжает это вычисление, достаточно показать, что $M_p^{(i+1)}$ совпадает с множеством сообщений, отправленных процессу p в $(i + 1)$ -м такте.

Допустим, что процесс q отправляет сообщение процессу p в $(i + 1)$ -м такте, т.е. когда часы процесса q показывают время $2(i + 1)\mu$. Обозначим символом σ (глобальный) момент времени отправления этого сообщения, а символом τ — (глобальный) момент времени доставки этого сообщения процессу p . Так как $CLOCK_q^{(\sigma)} = 2(i + 1)\mu$ и процесс q является соседом процесса p , выполняется неравенство $CLOCK_p^{(\sigma)} < 2(i + 1)\mu + \mu$ (по теореме 12.4). А коль скоро $\tau - \sigma < \mu$, то выполняется неравенство $CLOCK_p^{(\tau)} < 2(i + 1)\mu + \mu + \mu = 2(i + 2)\mu$, т.е. процесс p получает указанное сообщение, до того как он завершит $(i + 1)$ -й такт. $\square$

Может случиться так, что процесс получит сообщение, относящееся к i -му такту, еще до того как начнет выполнение этого такта. Это сообщение тогда нужно сохранить в памяти и обработать его только на следующем такте. Поэтому процесс должен иметь возможность разбираться в том, на каком такте нужно обрабатывать каждое полученное сообщение. Этого можно добиться, снабдив каждое отправленное сообщение индикатором, равным номеру такта по модулю 2 (см. упражнение 12.3). Дополнительной информации не понадобится, если

каждый процесс будет сохранять в памяти показания своих часов в момент получения сообщения $\langle \text{start} \rangle$ от каждого из своих соседей (см. упражнение 12.4 или работу [191]).

Сложность синхронизации. Измерим сложность механизма синхронизации по времени и по числу обменов сообщениями, принимая во внимание как инициализацию синхронизатора, так и издержки моделирования одного такта синхронного вычисления (см. также § 12.6.3).

Для проведения этапа инициализации требуется совершить $2|E|$ обменов сообщениями, а сам этап занимает $O(D)$ единиц времени. Для проведения этапа моделирования не требуется дополнительных сообщений (отправляются только сообщения, относящиеся к самому синхронному алгоритму), и каждый процесс завершает один такт спустя 2μ единиц времени. Длительность одного такта будем называть *длиной такта*.

В работе Теля и др. [191] был исследован класс всех синхронизаторов для ABD-сетей, в которых используется $2|E|$ обменов сообщениями на этапе инициализации и не требуется никаких дополнительных сообщений на этапе моделирования. Было установлено, что длина такта, равная 2μ , является оптимальной для произвольных сетей, для колец и для гиперкубов. Для звездчатых сетей и клик можно провести синхронизацию так, что длина такта будет составлять соответственно $(3/2)\mu$ и $(2 - 1/N)\mu$ единиц времени, и обе эти длины являются оптимальными.

Дрейф часов и учет времени обработки данных. Чтобы использовать результаты, полученные для ABD-сетей на практике, допущения, приведенные в определении 12.3, должны быть ослаблены. На практике часы процессоров подвержены дрейфу, а время обработки сообщений отлично от нуля.

С дрейфом часов можно справиться путем постепенного увеличения длины такта на этапе моделирования, чтобы компенсировать возможное возрастающее расхождение в показаниях часов соседних процессов. Если длина такта в конце концов станет слишком велика, то после определенного числа тактов можно выполнить повторно этап сверки часов (см. [187, гл. 2]).

Если нельзя пренебречь временем, необходимым для обработки сообщений в одном такте, но можно ограничить это время некоторой константой π , то синхронизатор можно легко приспособить к этому путем увеличения длины такта на величину π .

12.2. Выборы в синхронных сетях

В этом параграфе мы задаемся целью продемонстрировать, что решение определенных задач осуществляется синхронными сетями с меньшей сложностью по числу обменов сообщениями по сравнению с асинхронными сетями. В гл. 7 были доказаны нижние оценки сложности по числу обменов сообщениями для задачи о выборе лидера в асинхронной сети. Сложность того синхронного алгоритма, который будет представлен в этом параграфе, меньше этих оценок. Сложность

по времени (т. е. число тактов, на протяжении которых избирается лидер), существенно зависит от того, известен или нет размер сети.

В этом алгоритме подразумевается, что отличительные признаки процессов представлены попарно различными целыми числами; если отличительные признаки выбраны из какого-нибудь счетного множества, то их можно перевести в целые числа, воспользовавшись какой-нибудь фиксированной нумерацией этого множества. Предполагается также, что все инициаторы алгоритма приступают к выборам в одном и том же такте (в 0-м такте). В § 12.2.3 мы вкратце обсудим тот случай, когда инициаторы не обязательно запускают алгоритм в одном и том же такте.

12.2.1. Размер сети известен

В алгоритмах, которые мы будем рассматривать в этом параграфе, существенно используется осведомленность о верхней оценке D диаметра сети. Если известно число N процессоров в сети, то величину $N - 1$ можно также использовать в качестве верхней оценки диаметра.

Алгоритм. Процесс p может инициировать поток сообщений, отправив некоторое сообщение всем своим соседям. После этого каждый отличный от p процесс, получивший впервые это сообщение, отправляет его всем своим соседям в следующем такте. Такой способ наводнения сети некоторым сообщением обойдется в $2|E|$ обменов сообщениями, и если процесс инициирует поток сообщений в i_0 -м такте, то каждый процесс получит это сообщение самое позднее в $(i_0 + (D - 1))$ -м такте. Отсюда следует, что если процесс не получил наводняющее сеть сообщение в $(i + (D - 1))$ -м такте или ранее, то никакой процесс не инициировал поток сообщений в i -м такте или ранее.

В рассматриваемом алгоритме избрания лидера сеть наводняется только одним сообщением, а именно тем, которое было отправлено процессом p_0 , имеющим наименьший отличительный признак; само наводнение начинается в $p_0 D$ -м такте. Процесс p_0 обнаружит, что в сети нет процессов с меньшими отличительными признаками, если он не получит наводняющее сеть сообщения в $(p_0 D - 1)$ -м такте или ранее. Не начиная наводнять сеть сообщениями, все процессы тем самым неявно отправляют поток сообщений вида «мой отличительный признак превосходит $p_0 - 1$ ».

В итоге процесс p ожидает, до тех пор пока не будет доставлено какое-нибудь сообщение или не наступит pD -такт. В последнем случае p инициирует поток сообщений и становится лидером. Этот алгоритм был предложен в работе Сан-торо и Ротема [170]. Его сложность по числу обменов сообщениями составляет величину $O(|E|)$, а сложность по времени равна $p_0 D + (D - 1)$, где p_0 — это наименьший отличительный признак. Обратите внимание на то, что сложность по времени ограничена функцией, не зависящей от N .

Расширения и варианты. Алгоритм можно легко расширить так, чтобы вычислять остовное дерево, корнем которого являлся бы лидер. Каждый процесс

выбирает в качестве родительской вершины один из тех процессов, от которых ему довелось получить хоть какое-то сообщение в самом раннем такте.

Коль скоро сообщения не предполагают какой-либо обмен информацией, битовая сложность таких алгоритмов равна их сложности по числу обменов сообщениями.

Если есть сведения о топологии сети, то наводнение сети сообщениями можно осуществить более эффективно. Само собой разумеется, что в случае произвольной, заранее неизвестной топологии сети хотя бы одно сообщение должно быть отправлено по каждому каналу; в необходимости этого можно убедиться, повторив рассуждения, подобные тем, которые были использованы при доказательстве теоремы 7.15. Если известно, что сеть представляет собой клику, тор или гиперкуб, то число сообщений можно сократить, воспользовавшись более эффективными потоковыми алгоритмами (см. упражнение 12.6).

Ван Ливен и др. в работе [129] показали, что сложность по времени можно уменьшить ценой увеличения числа обменов сообщениями. Если позволить наводнению осуществляться k раз (доведя тем самым сложность по числу обменов сообщениями до величины $O(k|E|)$), наименьший отличительный признак можно будет определить спустя $O(p_0^{1/k} D)$ тактов после начала работы.

12.2.2. Размер сети неизвестен

В случае, когда размер и диаметр сети неизвестны, эффективный алгоритм избрания лидера получается за счет применения принципа угасания волны (§ 7.3.1) к алгоритму *обхода*, с использованием выборочной задержки.

Если процесс p является инициатором выборов лидера, то p запускает алгоритм обхода и помечает маркер своим отличительным признаком p . Маркер передается в соответствии с алгоритмом обхода, до тех пор пока либо он не будет поглощен процессом, которому уже довелось видеть маркер с меньшим отличительным признаком, либо он не завершит свой обход в процессе p . В последнем случае p обретает статус *leader*. Маркер инициатора p перед каждым переходом задерживается на 2^p тактов, т. е. если какой-либо процесс получает в i -м такте маркер, выпущенный процессом p , то передача маркера произойдет в $(i + 2^p)$ -м такте.

Корректность предложенного алгоритма доказывается по той же схеме, что и теорема 7.17. Обозначим символом p_0 инициатор выборов, имеющий наименьший отличительный признак. Процесс p_0 получает свой маркер обратно и избирается лидером; при этом ни один маркер другого инициатора не завершает обход, поскольку все они будут поглощены процессом p_0 или каким-нибудь другим инициатором.

Остается показать, как выборочная задержка ограничивает сложность по числу обменов сообщений величиной $O(W)$, где W — сложность алгоритма обхода. Процесс p_0 избирается на $(2^{p_0} W)$ -м такте, потому что маркер совершает W переходов, для каждого из которых требуется 2^{p_0} тактов. К началу $(2^{p_0} W)$ -го такта каждый процесс уже смог ознакомиться с отличительным признаком p_0 , и это

означает, что никакие маркеры после этого такта уже не отправляются. Маркер инициатора с отличительным признаком $p \neq p_0$ передается самое большее $(2^{p_0}W)/2^p$ раз, поскольку к началу каждого P -го такта этот маркер передавался не более $P/2^p$ раз, а после $(2^{p_0}W)$ -го такта он не передается вообще. Рассмотрим множество инициаторов S . Общее число передач сообщений ограничено величиной, равной W (число переходов маркера процесса p_0) плюс число переходов, совершенных маркерами процессов, отличных от p_0 :

$$\begin{aligned} \text{сложность} &= W + \sum_{p \in S \setminus \{p_0\}} (\text{переходы } p\text{-го маркера}) \leq \\ &\leq W + \sum_{p \in S \setminus \{p_0\}} \frac{2^{p_0}W}{2^p} < \\ &< W + \sum_{i>0} \frac{W}{2^i} = 2W. \end{aligned}$$

Содержимое сообщений можно кодировать с использованием времени, и поэтому каждый маркер можно передавать при помощи двух сообщений по одному биту каждое. Отсюда следует, что битовая сложность задачи о выборах лидера ограничена сверху величиной $O(W)$. В нашем случае кодирование при помощи времени не очень сильно увеличивает сложность по времени, так как длительность задержки маркера в каждой вершине сети превосходит время, необходимое для его передачи. Сложность по времени этого алгоритма также не может быть ограничена функцией зависящей от N , причем эта функция экспоненциально растет относительно p_0 , наименьшего отличительного признака процесса.

12.2.3. Дополнительные результаты

Алгоритмы сравнения. Алгоритмы, описанные в этом параграфе, не относятся к числу алгоритмов сравнения. Поскольку в алгоритмах сравнения к отличительным признакам процессов можно применять только операцию сравнения, отсчет тактов в зависимости от значения отличительного признака в этих алгоритмах недопустим.

Фридриксон и Линч в работе [91] показали, что в алгоритмах избрания лидера с использованием только одной операции сравнения на синхронных кольцах приходится совершать не менее $\Omega(N \log N)$ обменов сообщениями, и это означает, что нижняя оценка для асинхронных алгоритмов сохраняется и для синхронных алгоритмов сравнения. Кроме того, Фридриксон и Линч установили, что если ограничить сложность по времени функцией, зависящей от N и предполагать, что мощность множества возможных отличительных признаков не ограничена, то каждый алгоритм в худшем случае будет вести себя, как алгоритм сравнения. А это означает, что потребуется совершить не менее $\Omega(N \log N)$ обменов сообщениями.

Эти результаты показывают, что уменьшение сложности по числу обменов сообщениями можно достичь только за счет увеличения сложности по времени

до величины, которую нельзя ограничить функцией, зависящей от N . Открытым остается вопрос о том, существуют ли такие синхронные алгоритмы избрания лидера для сетей заранее неизвестного размера, сложность которых по числу сообщений оставалась бы такой же, как и у алгоритмов, рассмотренных нами здесь, а сложность по времени была бы ограничена *полиномом*, зависящим от наименьшего отличительного признака.

Неодновременное начало. Задача о выборах становится труднее, если не предполагать (как это мы сделали неявно), что каждый процесс приступает к выборам в одном и том же такте (т.е. все процессы начинают отсчет с 0-го такта). Наводняя сеть сообщениями-побудками, которые рассылаются процессами, приступившими к выборам, можно добиться того, что все инициаторы запустят алгоритм спустя D тактов. Этого вполне достаточно для (адаптированных вариантов) алгоритмов, рассмотренных в этом параграфе. При этом, однако, для такой побудки потребуется совершить $O(|E|)$ обменов сообщениями.

Выборы в клике. Алгоритмы, представленные в этом параграфе, позволяют провести выборы в клике с использованием $O(N)$ сообщений. Однако для этого требуется, чтобы все процессы приступили к выборам в одном и том же такте. Добавление этапа инициализации, на котором сообщения-побудки наводят сеть, увеличивает сложность по числу обменов сообщениями на $2|E|$ сообщений, и мы получаем величину $\Omega(N^2)$. Сложность проведения выборов лидера в кликах изучалась в работе Афека и Гафни [3], которым удалось доказать, что если не предусматривается одновременное начало выборов, то для их проведения необходимо и достаточно совершить $\Theta(N \log N)$ обменов сообщениями. В их работе представлен алгоритм, имеющий именно такую сложность по числу обменов сообщениями, и при этом его сложность по времени составляет величину $O(\log N)$.

12.3. Алгоритмы-синхронизаторы

В этом параграфе мы рассмотрим несколько синхронизаторов для вполне асинхронных сетей. Для измерения сложности таких синхронизаторов мы будем учитывать следующие четыре параметра:

- M_{init} , — сложность по числу обменов сообщениями этапа инициализации;
- T_{init} , — сложность по времени этапа инициализации;
- M_{pulse} , — сложность по числу обменов сообщениями моделирования одного такта;
- T_{pulse} , — сложность по времени моделирования одного такта.

Будем говорить, что процесс моделирует тот или иной такт, если он отправляет все сообщения, относящиеся к этому такту. Сложность этапа инициализации по времени считается равной наибольшему промежутку времени между первым шагом, совершенным каким-либо процессом, и последним моделированием первого такта. Сложность по времени моделирования одного такта считается равной максимальному промежутку времени между последним моделированием одного такта и последним моделированием следующего за ним такта. В обоих случаях

сложность по времени измеряется в рамках идеализированных допущений о согласованности времени, представленных в определении 6.31. Сложность по числу обменов сообщениями моделирования одного такта учитывает только обмены дополнительными сообщениями, а не обмены сообщениями моделируемого алгоритма.

Синхронный алгоритм, имеющий сложность по числу обменов сообщениями, равную M , и сложность по времени, равную T , может быть промоделирован с использованием $M_{\text{init}} + M + M_{\text{pulse}}T$ обменов сообщениями за $T_{\text{init}} + T_{\text{pulse}}T$ единиц времени. В таблице 12.3 приведена сводка алгоритмов-синхронизаторов, рассматриваемых в этой главе.

12.3.1. Простой синхронизатор

Во всех синхронизаторах, рассматриваемых в этой главе, всякий процесс удостоверяется в получении всех сообщений, относящихся к текущему такту; как только это происходит, все сообщения обрабатываются, и наступает следующий такт. В ABD-синхронизаторах для удостоверения в получении применяются часы. Но синхронизаторы для вполне асинхронных систем такой возможностью не обладают. Так как всякий асинхронный алгоритм можно представить в такой форме, когда управление вычислением осуществляют сообщения, каждый процесс обязан получить хоть какое-то сообщение, чтобы перейти к следующему такту. Отсюда следует, что для моделирования каждого такта нужно совершить не менее N обменов сообщениями.

В простейшем синхронизаторе каждый процесс отправляет в точности по одному сообщению каждому своему соседу в каждом такте. Если в моделируемом алгоритме отправление сообщения в каком-либо такте не предусматривается, то синхронизатор добавляет «пустое» сообщение. Если в моделируемом алгоритме отправляется более одного сообщения, эти сообщения упаковываются синхронизатором в одно сообщение. Каждый процесс обязан получить в точности по одному сообщению от каждого своего соседа в каждом такте. Когда от каждого соседа будет получено сообщение, относящееся к i -му такту, проводится обработка всех сообщений и переход к моделированию $(i + 1)$ -го такта (см. алгоритм 12.2). Процессы-инициаторы проводят инициализацию самопроизвольно, а остальные процессы присоединяются к этому после получения первого сообщения.

Процесс p , тактовый счетчик которого показывает $pulse_p = i$, может принимать только такие сообщения, которые относятся к i -му и $(i + 1)$ -му тактам. Все сообщения, относящиеся к более ранним тактам, уже были получены, и ни один сосед процесса p не может отправить сообщение, относящееся к более позднему такту, чем $i + 1$, до того как p отправит ему свое сообщение, относящееся к $(i + 1)$ -му такту. Поэтому процессу p приходится вести учет только числа сообщений, прием которых ожидается в текущем ($wait_p$) и следующем ($nwait_p$) тактах, а для этого достаточно передавать номер такта по модулю 2.

Чтобы приступить к моделированию, не нужно никаких дополнительных сообщений: каждый процесс отправляет сообщения, относящиеся к начальному

```

var pulsep : integer;
statep : Zp;
waitp, nwaitp : integer;
init |Neighp|;

```

Чтобы запустить алгоритм:

```

begin (* начать самопроизвольно первый такт *)
  statep := initial state; pulsep := 1;
  forall q ∈ Neighp do
    begin Mp[q] := {m ∈ MGp(statep) : m has destination q};
    send ⟨pack, Mp[q], pulsep⟩ to q
  end
end

{ Доставлен пакет сообщений ⟨pack, (m1, ..), i⟩ }
begin receive and store message;
  if i = pulsep then waitp := waitp - 1 else nwaitp := nwaitp - 1;
  while waitp = 0 do
    begin (* M — это множество сообщений, полученных в этом такте *)
      statep := dp with (statep, M) ⊢ dp;
      pulsep := pulsep + 1; waitp := nwaitp; nwaitp := |Neighp|;
      forall q ∈ Neighp do
        begin Mp[q] := {m ∈ MGp(statep) : m has destination q};
        send ⟨pack, Mp[q], pulsep⟩ to q
      end
    end
  end
end

```

Алгоритм 12.2. Простой синхронизатор

такту, либо самопроизвольно, либо по получении какого-нибудь сообщения. Как только хоть один процесс приступит к моделированию, все остальные процессы присоединятся к нему спустя самое большее, D единиц времени. Следовательно, $M_{\text{init}} = 0$ и $T_{\text{init}} = D$.

Для моделирования каждого такта требуется совершить в точности $2|E|$ обменов сообщениями, и в худшем случае все эти сообщения могут оказаться дополнительными сообщениями. Поэтому $M_{\text{pulse}} = 2|E|$. Если последнее моделирование i -го такта случилось в момент времени t , все сообщения, относящиеся к такту i , уже будут к моменту времени $t + 1$, а это означает, что последнее моделирование $(i + 1)$ -го такта случится самое позднее в момент времени $t + 1$. Следовательно, $T_{\text{pulse}} = 1$.

Для использования простого синхронизатора в приложениях не нужны ни лидер, ни сведения о размере сети, ни какая-либо информация о топологии сети. Также не требуется, чтобы процессы сети носили имена. Такая всеобщность простого синхронизатора свидетельствует о том, что все синхронные алгоритмы можно моделировать при помощи асинхронных сетей. Поэтому любую задачу, ко-

торию можно решить при помощи синхронных алгоритмов, можно решить также и при помощи асинхронного алгоритма.

12.3.2. α , β , и γ -синхронизаторы

В простом синхронизаторе каждый процесс в любом такте взаимосвязан с каждым из своих соседей, и это приводит к тому, что сложность по числу обменов сообщениями при моделировании одного такта составляет величину $\Theta(|E|)$. Сложность по числу обменов сообщениями можно понизить (за счет увеличения сложности по времени), если проводить взаимосвязь, используя некоторую выделенную подсеть. Коммуникация, которая навязывается синхронизатором, предназначена для того, чтобы каждый процесс удостоверился в том, что все сообщения, отправленные в его адрес соседними процессами, уже получены.

Процесс p может *обезопаситься* в своем текущем такте некоторое время спустя после получения всех сообщений, отправленных процессом p в этом такте. Подтверждения отправляются для всех сообщений моделируемого алгоритма, и когда p получит подтверждения для всех отправленных им сообщений, он будет в безопасности. Теперь задачей синхронизатора становится организация такой межпроцессной связи, благодаря которой всякий процесс сможет убедиться в том, что все его соседи находятся в безопасности. Как только процесс p проверит, что все его соседи находятся в безопасности в такте i , он сможет обработать сообщения, относящиеся к i -му такту, перейти в другое состояние и приступить к моделированию $(i + 1)$ -го такта.

Описанные далее α и β -синхронизаторы представляют собой частные случаи γ -синхронизатора, который будет описан последним. Эти три разновидности синхронизаторов были предложены в работе Авербаха [14].

α -синхронизатор. Первый из этих трех синхронизаторов — α -синхронизатор — очень похож на простой синхронизатор. Обмен сообщениями в нем проводится по каждому каналу: каждый процесс, оказавшийся в безопасности, отправляет сообщение $\langle \text{iamsafe} \rangle$ всем своим соседям. Процесс узнает о том, что все соседи находятся в безопасности, как только получит сообщение $\langle \text{iamsafe} \rangle$ от каждого соседа.

Никакой процедуры инициализации не предусматривается, и в каждом такте, как легко видеть, синхронизатор отправляет $O(|E|)$ сообщений. Если последнее моделирование i -го такта происходит в момент времени t , все сообщения, относящиеся к этому такту, будут получены к моменту $t + 1$, а все подтверждения — к моменту времени $t + 2$. Таким образом, каждый процесс оказывается в безопасности к моменту времени $t + 2$, и все сообщения $\langle \text{iamsafe} \rangle$ будут получены к моменту времени $t + 3$, что позволит всем процессам приступить к моделированию $(i + 1)$ -го такта. Значит, $T_{\text{pulse}} = 3$.

β -синхронизатор. В β -синхронизаторе связь осуществляется по остовному дереву. Процесс отправляет сообщение $\langle \text{ts} \rangle$ («дерево безопасно») родительской вершине дерева, когда все процессы в его поддереве оказываются в безопасно-

сти. Это имеет место, когда сам процесс находится в безопасности и сообщение $\langle \text{ts} \rangle$ было получено им от каждой сыновней вершины дерева. Когда корневая вершина будет в безопасности и получит сообщение $\langle \text{ts} \rangle$ от каждой сыновней вершины, все процессы будут в безопасности. Тогда корневая вершина отправляет сообщение $\langle \text{pulse} \rangle$ по остовному дереву, чтобы дать знать всем процессам, что все они оказались в безопасности в текущем такте. Получение сообщения $\langle \text{pulse} \rangle$ свидетельствует о том, что все процессы в безопасности. Это означает, что все соседи процесса, получившего это сообщение, находятся в безопасности и сам процесс может перейти к моделированию следующего такта.

Для этого синхронизатора нужен этап инициализации, на котором строится остовное дерево. В параграфе 7.5.3 рассказывается о том, как это сделать с использованием $\Theta(N \log N + |E|)$ обменов сообщениями при том условии, что процессам доступны уникальные отличительные признаки. Существует решение, сложность которого составляет $O(N)$ единиц времени. Централизованный алгоритм может вычислить остовное дерево с использованием $O(|E|)$ обменов сообщениями за $O(N)$ единиц времени.

Кроме подтверждений, для моделирования каждого такта требуются, чтобы сообщения $\langle \text{ts} \rangle$ и $\langle \text{pulse} \rangle$ были отправлены по каждому ребру дерева. Всего для этого требуется $O(N)$ контрольных сообщений. Время, которое расходуется на их рассылку, пропорционально глубине дерева, которая в худшем случае оценивается величиной $\Omega(N)$.

γ -синхронизатор. Для γ -синхронизатора нужен этап инициализации, в ходе которого сеть разбивается на кластеры. В каждом кластере алгоритм работает как β -синхронизатор, а между кластерами алгоритм работает как α -синхронизатор.

В каждом кластере есть остовное дерево кластера и лидер кластера, являющийся корневой вершиной остовного дерева кластера. Для любых двух соседних кластеров (два кластера считаются соседними, если существует ребро между вершинами этих двух кластеров) выделяется одно *предпочтительное ребро* между этими двумя кластерами. Синхронизация состоит из пяти шагов.

1. Процесс отправляет сообщение $\langle \text{ts} \rangle$ родительской вершине в дереве кластера, если все процессы его поддерева находятся в безопасности, т. е. если сам процесс находится в безопасности и он получил сообщение $\langle \text{ts} \rangle$ от каждой своей сыновней вершины.

2. Если лидер кластера находится в безопасности и получил сообщение $\langle \text{ts} \rangle$ от каждой сыновней вершины, все процессы кластера находятся в безопасности. Тогда по остовному дереву кластера отправляется сообщение $\langle \text{cs} \rangle$ («кластер безопасен»), чтобы дать знать каждому процессу кластера, что все процессы кластера в безопасности.

3. Те процессы кластера, которые инцидентны предпочтительным ребрам, ведущим в другие кластеры, отправляют сообщения другим кластерам, чтобы оповестить их о том, что этот кластер в безопасности. Для этого по предпочтительным ребрам отправляются сообщения $\langle \text{ocs} \rangle$ («наш кластер безопасен»), как только будут получены сообщения $\langle \text{cs} \rangle$.

4. В каждом кластере сообщения отправляются в корневую вершину, чтобы уведомить ее о том, что все соседние кластеры находятся в безопасности. Всякий процесс отправляет для этого своей родительской вершине в дереве кластера сообщение $\langle \text{nсs} \rangle$ («соседний кластер безопасен»), как только все соседние кластеры, соединенные с данным остовным деревом кластера, сообщают о том, что они находятся в безопасности. Это происходит тогда, когда этот процесс получает сообщение $\langle \text{ocs} \rangle$ по всем инцидентным ему предпочтительным ребрам, а также сообщения $\langle \text{nсs} \rangle$ от всех своих сыновних вершин в дереве кластера.

5. Когда лидер кластера получает сообщение $\langle \text{nсs} \rangle$ от каждой сыновней вершины, а также сообщение $\langle \text{ocs} \rangle$ по каждому инцидентному предпочтительному ребру, все соседние кластеры уже в безопасности. Если это так и сам кластер в безопасности, по дереву кластера рассылается сообщение $\langle \text{pulse} \rangle$, чтобы запустить в каждом процессе кластера моделирование очередного такта.

Когда процесс получает сообщение $\langle \text{pulse} \rangle$, его кластер, а также все соседние кластеры находятся в безопасности. Это означает, что все его соседи находятся в безопасности.

Чтобы установить сложность этого синхронизатора, обозначим для кластерного разбиения c суммарное число ребер в деревьях и предпочтительных ребер символом E_c , а максимальную высоту деревьев в разбиении c — символом H_c . Так как синхронизатор в ходе моделирования одного такта отправляет четыре сообщения по каждому ребру дерева ($\langle \text{ts} \rangle$, $\langle \text{cs} \rangle$, $\langle \text{nсs} \rangle$, и $\langle \text{pulse} \rangle$) и два сообщения по каждому предпочтительному ребру (два сообщения $\langle \text{ocs} \rangle$), значение параметра M_{pulse} составляет величину $O(E_c)$. Допустим, что последнее моделирование i -го такта произошло в момент времени t . Тогда к моменту времени $t+2$ все процессы будут в безопасности, к моменту времени $t+2+H_c$ все кластеры (представленные их лидерами) будут осведомлены о том, что они находятся в безопасности, и к моменту времени $t+2+2H_c$ все процессы получают сообщения $\langle \text{cs} \rangle$ и отправят сообщения $\langle \text{ocs} \rangle$ по предпочтительным ребрам. К моменту времени $t+3+2H_c$ все сообщения $\langle \text{ocs} \rangle$ будут доставлены, к моменту времени $t+3+3H_c$ корневые вершины каждого кластера будут осведомлены о том, что все соседние кластеры находятся в безопасности, и к моменту времени $t+3+4H_c$ все процессы получают сообщение $\langle \text{pulse} \rangle$ и приступят к моделированию следующего такта. Поэтому значение параметра T_{pulse} составляет величину $O(H_c)$.

Если в качестве единого кластера взять всю сеть, то $E_c = N - 1$, $H_c = O(N)$, и мы получаем β -синхронизатор. Если сделать кластером каждую отдельную вершину, то $E_c = E$, $H_c = O(1)$, и мы получаем α -синхронизатор.

Вычисление подходящего разбиения на кластеры. Сложность γ -синхронизатора зависит от разбиения сети на кластеры, используемого в алгоритме. Это разбиение (кластеризация) определяется следующими характеристиками:

- 1) разбиением сети на кластеры (связные подграфы);
- 2) выделением центра (лидера) и корневого остовного дерева в каждом кластере;
- 3) выделением одного предпочтительного ребра для каждой пары соседних кластеров.

Теорема 12.6. Для любого k из отрезка $2 \leq k < N$ существует такая кластеризация c , что $E_c \leq kN$ и $H_c \leq \log N / \log k$.

Доказательство. Требуемая кластеризация строится путем последовательного выбора кластеров $C_1, C_2, \dots$. Предположим, что мы уже выбрали кластеры $C_1, \dots, C_r$ и в сети есть вершины, не принадлежащие ни одному из этих кластеров.

Чтобы построить кластер C_{r+1} , нужно выбрать одну из таких оставшихся вершин p в качестве лидера нового кластера и добавлять к этому кластеру вершины слой за слоем. Множество $\{p\}$, состоящее из одной вершины, будет первым слоем нового кластера. Предположим, что после построения i слоев кластера C_{r+1} в нем оказалось n вершин. Рассмотрим множество S всех тех соседей вершин i -го слоя, которые еще не принадлежат кластеру. Если мощность множества S не меньше $(k-1)n$, то вершины множества S добавляются в кластер C_{r+1} в качестве следующего слоя. В противном случае, т. е. если S содержит менее $(k-1)n$ вершин, построение кластера C_{r+1} считается завершенным.

При построении каждого кластера предполагается, что в кластере, состоящем из i уровней (т. е. имеющем остовное дерево глубины $i-1$), содержится не менее k^{i-1} вершин и, следовательно, параметр H_c ограничен величиной $\log_k N = \log N / \log k$.

Число древесных ребер при проведении кластеризации ограничено числом N , поскольку из этих ребер составлен остовный лес сети. Для всякой пары соседних кластеров выбирается одно предпочтительное ребро, которое приписывается к тому из этих двух кластеров, который был построен первым. Построение этого кластера считается завершенным, когда число оставшихся соседних вершин менее чем в $k-1$ раз меньше размера кластера. Отсюда следует, что для каждого кластера размера n имеется не более $(k-1)n$ соседних кластеров, которые будут построены позднее. Следуя такой стратегии приписывания предпочтительных ребер, мы всякий раз будем получать кластеры, которым приписано не более $(k-1)n$ предпочтительных ребер, где n — это размер самого кластера. Поэтому общее число предпочтительных ребер не будет превосходить $(k-1)N$. Этим подтверждается неравенство $E_c < kN$. $\square$

Распределенный алгоритм, воплощающий эту конструкцию, был описан в работе Авербаха [14]. В этом алгоритме проводится $O(kN^2)$ обменов сообщениями, и его сложность по времени составляет величину $O(N \log N / \log k)$.

12.4. Приложения: поиск в ширину

Чтобы продемонстрировать применение синхронизаторов, мы рассмотрим в этом параграфе несколько алгоритмов вычисления остовного дерева сети, соответствующего поиску в ширину (BFS). Остовное дерево T сети G называется *деревом поиска в ширину*, если для каждой вершины дерева путь из корня дерева в эту вершину является кратчайшим путем в G . Таким образом, остовное дерево поиска в ширину является оптимальным деревом с точки зрения метрики по числу переходов, которое обеспечивает связь корня со всеми вершинами (см. теорему 4.2).

Синхронизатор	M_{init}	T_{init}	M_{pulse}	T_{pulse}	Прим.
ABD	$2 E $	$D\mu$	0	2μ	
Простой	0	D	$2 E $	1	
α	0	D	$2 E $	3	Acks.
β	$\Theta(N \log N + E)$	$O(N)$	$2N - 2$	$2N$	Acks.
γ	$O(kN^2)$	$O\left(N \frac{\log N}{\log k}\right)$	$O(kN)$	$O\left(\frac{\log N}{\log k}\right)$	$2 \leq k < N$ Acks.

Таблица 12.3. Сводная таблица синхронизаторов

Деревья поиска в ширину играют важную роль в проектировании алгоритмов, эффективных по времени и по числу обменов сообщениями, поскольку они позволяют осуществлять оптимальную широкополосную рассылку сообщений за минимальное время. Используя BFS-деревья, можно заставить работать более эффективно схему древесной разметки Санторо и Хатиба (§ 4.4.2). Фридриксон в своей работе [92] использовал BFS-деревья для эффективного решения проблемы поиска кратчайших путей.

Вычисление BFS-дерева проводится централизованным алгоритмом, инициатором которого служит корневая вершина дерева (предполагается, что она выделена заранее). В этом параграфе мы будем анализировать сложность алгоритма поиска в ширину на основе параметров N и $|E|$. Следует обратить внимание на то, что глубина дерева ограничена величиной N . Можно провести и более подробный анализ на основе параметров N , $|E|$ и D , где D обозначает верхнюю оценку глубины дерева.

В § 12.4.1 мы сначала покажем, что вычисление BFS-дерева проводится необычайно просто в синхронных сетях: на самом деле, каждое выполнение алгоритма эха приводит к BFS-дереву. В § 12.4.2 мы совместим этот синхронный алгоритм с синхронизатором и сравним сложность предложенных решений. Асинхронные алгоритмы, приведенные в § 12.4.2 легко построить и понять, но они менее эффективны, нежели некоторые другие алгоритмы решения этой задачи. В § 12.4.3 мы обсудим два асинхронных алгоритма, предложенных Фридриксоном [92].

12.4.1. Синхронный BFS-алгоритм

Естественный способ построения дерева поиска в ширину состоит в последовательном вычислении его слоев. Первоначально корень дерева образует нулевой ярус, а после того как i -й ярус дерева построен, следующий $(i+1)$ -й ярус образуют соседи вершин i -го уровня.

В синхронном алгоритме вычисления дерева поиска в ширину каждый процесс будет отправлять сообщение каждому из своих соседей только в одном такте. Инициатор делает это в первом такте. Всякий другой процесс отправляет сообщения в $(i+1)$ -м такте при условии, что в i -м такте этот процесс впервые получил

одно или несколько сообщений. Процесс, впервые получивший сообщение в i -м такте причисляет себя к i -му ярусу и выбирает один из тех процессов, от которых он получил сообщение в i -м такте в качестве родительской вершины.

Теорема 12.7. *В описанном алгоритме процесс, находящийся на расстоянии i от инициатора, отправляет сообщения в $(i+1)$ -м такте, и если $i > 0$, то он выбирает процесс, находящийся на расстоянии $i-1$ от инициатора, в качестве родительской вершины.*

Исходя из выбора родительских вершин видно, что построенное в результате дерево будет деревом поиска в ширину. А так как каждый процесс отправляет сообщения только один раз, сложность алгоритма по числу обменов сообщениями равна $2|E|$. Сложность алгоритма по времени составляет величину $O(N)$, поскольку после N -го такта каждый процесс будет иметь родительскую вершину.

Алгоритм	Сообщения	Время
Синхронный	$2 E $	$O(N)$
Синх+ABD	$O(E)$	$O(N)$
Синх+ α	$O(N E)$	$O(N)$
Синх+ β	$O(N^2)$	$O(N^2)$
Синх+ γ	$O(kN^2)$	$O\left(N \frac{\log N}{\log k}\right)$
Простой	$O(N^2 + E)$	$O(N^2)$
Улучшенный	$O(N\sqrt{ E })$	$O(N\sqrt{ E })$
[15]	$O(E 2^{\sqrt{\log N \log \log N}})$	$O(N 2^{\sqrt{\log N \log \log N}})$

Таблица 12.4. Сложность алгоритмов поиска в ширину

12.4.2. Соединение с синхронизатором

Алгоритм поиска в ширину для ABD или асинхронных сетей можно легко получить за счет соединения синхронного алгоритма и одного из тех синхронизаторов, которые рассматривались ранее в этой главе. Сложность полученных асинхронных алгоритмов приведена в таблице. Полученные таким путем алгоритмы легко постижимы, но они не являются наилучшими известными асинхронными алгоритмами.

12.4.3. Асинхронные BFS-алгоритмы

В этом параграфе мы обсудим три асинхронных алгоритма поиска в ширину, сложность которых меньше, чем сложность алгоритмов, представленных в предыдущем параграфе. У читателя не должно складываться впечатление о практической бесполезности применения концепции синхронизатора для проектиро-

вания асинхронных алгоритмов — синхронизаторы будут неявно применяться при проектировании этих новых алгоритмов.

Дерево поиска в ширину обычно строят последовательно по ярусам. После того как был сформирован i -й ярус дерева, соседи всех вершин этого яруса (если только они уже не вошли в состав дерева ранее) добавляются к дереву в качестве вершин $(i + 1)$ -го яруса. Посмотрим, что случится, если построение следующего $(i + 2)$ -го яруса начнется прежде, чем завершится построение $(i + 1)$ -го яруса. Соседи некоторых вершин, лежащих на $(i + 1)$ -м ярусе, добавляются к дереву в качестве вершин $(i + 2)$ -го яруса, но позднее может обнаружиться, что некоторые из этих новых вершин соседствуют с вершинами i -го яруса и поэтому на самом деле должны относиться к $(i + 1)$ -му ярусу.

В том простом алгоритме, который будет описан далее, избежать описанной ситуации удастся за счет того, что расширение дерева на каждом ярусе проводится согласованно. Синхронизация проводится согласно правилам работы β -синхронизатора и применяется к той части дерева, которая уже была построена. В более совершенных алгоритмах синхронизация проводится через каждые l ярусов. Затраты на синхронизацию при этом уменьшаются, но зато не исключается возможность того, что какая-нибудь вершина по ходу вычисления будет приписана к нескольким ярусам.

Уровень расположения вершины в построенном BFS-дереве будет равен расстоянию в сети между этой вершиной и корнем. Вершина, находящаяся на уровне f , может иметь соседей только на уровнях $f - 1$, f и $f + 1$ (см. упражнение 12.9).

Простой алгоритм. При запуске алгоритма инициатор приписывает себя к нулевому ярусу, и построение нулевого яруса BFS-дерева считается завершенным. В дереве строится последовательно ярус за ярусом, и после завершения построения f -го яруса каждая вершина на этом уровне будет осведомлена о том, что она находится на f -м ярусе, а также о тех ее соседях, которые лежат на $(f - 1)$ -м ярусе.

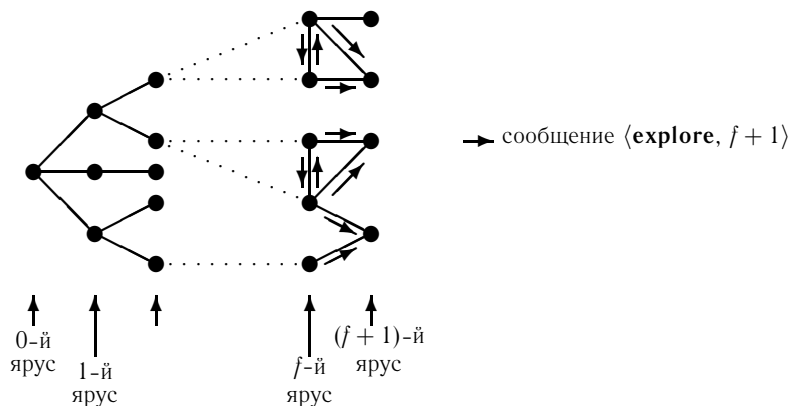


Рис. 12.5. Построение $(f + 1)$ -го уровня

Предположим, что построение f -го яруса завершено, но алгоритм еще не завершил работу. Чтобы начать построение $(f + 1)$ -го яруса, инициатор широко-вещательно рассылает по дереву сообщение $\langle \text{forward}, f \rangle$. Вершина p , лежащая на f -м ярусе, после получения сообщения $\langle \text{forward}, f \rangle$ отправляет сообщение $\langle \text{explore}, f + 1 \rangle$ тем из своих соседей, относительно которых она еще не располагает сведениями об их принадлежности $(f - 1)$ -му ярусу. Ситуация, связанная с отправлением сообщения $\langle \text{explore}, f + 1 \rangle$, изображена на рис. 12.5.

В результате того что сообщения $\langle \text{explore}, f + 1 \rangle$ будут отправлены всеми вершинами f -го яруса, вершина, находящаяся на расстоянии $f + 1$ от инициатора, получит указанное сообщение от каждого своего соседа, лежащего на f -м ярусе. Первое сообщение $\langle \text{explore}, f' \rangle$, полученное вершиной, указывает ее уровень f' и ее родительскую вершину в BFS-дереве. В ответ на это сообщение отправляется отклик $\langle \text{reverse}, \text{true} \rangle$. В ответ на все сообщения $\langle \text{explore}, f' \rangle$, поступившие позднее, следует отклик $\langle \text{reverse}, \text{false} \rangle$, и все отправители этих сообщений будут знать, что они находятся на $(f' - 1)$ -м ярусе. Заметим, что, когда построение $(f + 1)$ -го яруса завершается (а это означает, что на все сообщения $\langle \text{explore}, f + 1 \rangle$ были даны ответы), каждый процесс, лежащий на $(f + 1)$ -м ярусе уже будет осведомлен о том, какие из соседей находятся на f -м ярусе.

Вершина на f -м ярусе может также получать сообщения $\langle \text{explore}, f + 1 \rangle$, а именно от тех ее соседей, которые лежат на f -м ярусе. В ответ на эти сообщения не следует откликов $\langle \text{reverse}, b \rangle$, поскольку сообщение $\langle \text{explore}, f + 1 \rangle$, отправленное такому соседу, истолковывается как сообщение $\langle \text{reverse}, \text{false} \rangle$.

Все вершины f -го яруса ожидают поступления откликов на все отправленные ими сообщения $\langle \text{explore}, f + 1 \rangle$. Откликом является либо сообщение $\langle \text{reverse}, b \rangle$ (отправленное новой вершиной, лежащей на $(f + 1)$ -м ярусе и ставшей сыновней вершиной адресата этого сообщения, если $b = \text{true}$), либо сообщение $\langle \text{explore}, f + 1 \rangle$ (отправленное соседней вершиной, находящейся на том же самом f -м ярусе). Каждая вершина v , находящаяся не выше, чем на f -м ярусе, отправляет сообщение $\langle \text{reverse}, b \rangle$ своей родительской вершине, чтобы уведомить ее о том, что на все сообщения $\langle \text{explore}, f + 1 \rangle$, отправленные вершинами поддерева, растущего из v , был получен отклик. Значением параметра b будет true тогда и только тогда, когда хоть одна новая вершина была добавлена к такому поддереву. Всякая вершина, лежащая на f -м ярусе, отправляет сообщение $\langle \text{reverse}, b \rangle$, если она получила отклик на каждое отправленное ею сообщение $\langle \text{explore}, f + 1 \rangle$, и при этом $b = \text{true}$ лишь в том случае, когда среди этих откликов было хотя бы одно сообщение $\langle \text{reverse}, \text{true} \rangle$. Всякая вершина, лежащая выше f -го яруса, отправляет сообщение $\langle \text{reverse}, b \rangle$ родительской вершине, если ею было получено сообщение $\langle \text{reverse}, b' \rangle$ от каждой сыновней вершины. Точно так же значение параметра b равно true , если среди полученных сообщений было хотя бы одно сообщение $\langle \text{reverse}, \text{true} \rangle$.

Построение $(f + 1)$ -го яруса завершается, когда корень получает сообщения $\langle \text{reverse}, b \rangle$ от всех сыновних вершин. Если к дереву не было добавлено ни одной новой вершины (во всех полученных сообщениях содержалось значение false), то построение дерева завершается. В противном случае алгоритм переходит к построению $(f + 2)$ -го яруса.

Теорема 12.8. *Дерево поиска в ширину может быть построено централизованным асинхронным алгоритмом с использованием $O(N^2)$ обменов сообщениями за $O(N^2)$ единиц времени.*

Доказательство. Построение завершится, после того как будет построено самое большее N ярусов, и при построении любого яруса по каждому ребру дерева проходит не более одного сообщения типа **forward** или **explore** и не более одного отклика типа **reverse**. По всякому стягивающему ребру, соединяющему две вершины одного и того же яруса f , по ходу построения $(f+1)$ -го яруса проходят два сообщения $\langle \text{explore}, f+1 \rangle$. По стягивающему ребру, соединяющему вершину f -го яруса с вершиной $(f+1)$ -го яруса, по ходу построения $f+1$ -го яруса проходит одно сообщение $\langle \text{explore}, f+1 \rangle$ и одно сообщение $\langle \text{reverse}, false \rangle$. В любом случае по всякому стягивающему ребру проходят в точности два сообщения за все время работы алгоритма. Таким образом, сложность по числу обменов сообщениями ограничена величиной $2N(N-1) + 2(|E| - (N-1)) \leq 2N^2 + 2|E| = O(N^2)$. Построение $(f+1)$ -го яруса осуществляется за $2(f+1)$ единиц времени, и это означает, что алгоритм завершит свою работу спустя $O(N^2)$ единиц времени. $\square$

Улучшенный алгоритм. В простом алгоритме разведка каждого отдельного уровня синхронизируется корневой вершиной (посредством сообщений типа **forward** и **reverse**). Как видно из проведенного анализа сложности алгоритма, преобладающий вклад в нее вносят синхронизирующие сообщения. Если сеть является *плотной* (т. е., число ребер $|E|$ составляет величину $\Omega(N^2)$), сложность по числу обменов сообщениями нельзя улучшить, потому что количество ребер $|E|$ дает нижнюю оценку числа обменов сообщениями. Если же сеть *разрежена* (т. е., число ребер $|E|$ составляет величину $o(N^2)$), то издержки на синхронизацию можно сократить за счет проведения разведки сразу на несколько уровней вглубь за один синхронизованный раунд.

Предположим, что построение f -го яруса дерева завершено. Для построения следующих l ярусов инициатор широкоэвентально распространяет по дереву сообщение $\langle \text{forward}, f \rangle$. После получения этого сообщения вершины, лежащие на f -м ярусе, отправляют сообщения $\langle \text{explore}, f+1, l \rangle$ тем из своих соседей, относительно которых они еще не располагают сведениями об их принадлежности $(f-1)$ -му ярусу. Первый параметр этого сообщения указывает новый уровень адресата в том случае, если последний становится сыновней вершиной отправителя этого сообщения. Второй параметр задает количество уровней, которые должны быть разведаны, отсчитывая от того уровня, на котором находится отправитель этого сообщения.

Первоначально ярус, на котором лежит вершина, полагается равным ∞ . Когда в вершину p поступает сообщение $\langle \text{explore}, f+1, l \rangle$, параметр f сравнивается с тем ярусом $level_p$, к которому в текущий момент времени приписана вершина p . Если $f \geq level_p$, то в ответ на это сообщение отправляется отклик $\langle \text{reverse}, false \rangle$. Если $f < level_p$, то вершина p воспринимает отправителя этого сообщения как родительскую вершину в дереве, приписывает себя к вершинам

яруса f и в случае $l > 1$ отправляет сообщения $\langle \text{explore}, f+1, l-1 \rangle$ тем из своих соседей, относительно которых она еще не располагает сведениями об их принадлежности $(f+1)$ -му ярусу. (Последнее имеет место тогда, когда от этих соседей ранее было доставлено сообщение $\langle \text{explore}, f+2, l \rangle$.) Так же, как и в простом алгоритме, завершение построения фрагмента дерева начиная с $(f+1)$ -го яруса и оканчивая $(f+l)$ -м ярусом определяется потоком откликов типа **reverse** в ответ на сообщения типа **forward** и **explore**.

Теорема 12.9. *Если известно (приблизленно) отношение величин $|E|$ и N^2 , то дерево поиска в ширину может быть построено централизованным асинхронным алгоритмом с использованием $O(N\sqrt{E})$ обменов сообщениями за время $O(N\sqrt{E})$.*

Доказательство. Построение завершается приблизительно за N/l синхронизованных раундов, и поэтому по каждому ребру дерева проходит не более $2N/l$ сообщений типа **forward** и откликов на эти сообщения, и для всего дерева это составляет $O(N^2/l)$ обменов сообщениями. По каждому ребру дерева проходит не более l сообщений типа **explore** и l откликов на эти сообщения, и для всего дерева это составляет $O(l|E|)$ обменов сообщениями. В итоге сложность по числу сообщений составляет величину $O(N^2/l + l|E|)$. Так как для построения l ярусов дерева начиная с $(f+1)$ -го яруса и оканчивая $(f+l)$ -м ярусом требуется $O(f+l)$ единиц времени, сложность данного алгоритма по времени составляет величину $O(N^2/l)$.

Те оценки сложности, которые указаны в теореме, получаются при $l = \sqrt{N^2/E}$. $\square$

Алгоритм Авербаха—Галладжера. Введя дополнительные средства синхронизации, Авербах и Галладжер в работе [15] сумели еще более уменьшить сложность поиска в ширину. Те l ярусов, которые просматриваются между раундами синхронизации, будем называть *полосой*. В алгоритме Авербаха—Галладжера проводится не только глобальная синхронизация между просмотрами любых двух полос, но также и синхронизация просмотра каждой полосы; для этого используется остовный лес, построенный по ходу просмотра. Их алгоритм слишком сложен, чтобы объяснять его здесь более подробно, но для полноты картины его сложность также приведена в таблице 12.4.

12.5. Допущение Архимеда

Способность процессов следить за течением времени зависит от того, имеют или нет процессы доступ к реальным, физическим часам. Но даже в том случае, когда за ходом времени следить нельзя, вполне оправданно можно воспользоваться так называемым допущением Архимеда. Согласно этому допущению отношение между максимально возможной и минимально возможной длительностью одного шага вычисления, совершаемого процессом, или одного акта пересылки сообщения ограничено сверху.

Даже если система не снабжена настоящим устройством для измерения времени, некоторое грубое подобие часов можно получить, проводя, например, отсчет команд, выполненных каждым процессом, и объявляя начало очередного такта через каждые 1000 или 10000 команд. Тогда всякий процесс оценивает «истекшее время» по числу тактов, отсчитанных этим процессом. В дальнейшем для обозначения моментов физического времени мы будем использовать символы t_1 и t_2 , полагая, что их смысл будет ясен из контекста. Обозначим символом m наименьшую длительность одного такого такта какого-нибудь процесса в промежутке времени между моментами t_1 и t_2 . Обозначим символом u сумму, которую образуют максимальная задержка передачи сообщения в промежутке времени между моментами t_1 и t_2 и наибольшая длительность одного такта одного из процессов в промежутке времени между моментами t_1 и t_2 . Отношение u/m называется архимедовым отношением интервала времени (t_1, t_2) .

Определение 12.10. Распределенная система называется *архимедовой* (с *архимедовым отношением* s), если отношение u/m ограничено (некоторой величиной s).

Если заданы моменты времени t_1 и t_2 , то в интервале времени между моментами t_1 и t_2 случится лишь конечное число тактов или отправлений сообщений. Отсюда следует, что существует такое число s , при котором система является архимедовой с коэффициентом s . Для многих систем можно выдвинуть вполне обоснованное допущение, что существует такое число s_0 , что вся система является архимедовой с коэффициентом s_0 на всяком интервале времени между началом и завершением работы любого алгоритма, выполняемого этой системой. Отсюда следует, что система удовлетворяет (очень слабому) допущению синхронизации. В связи с этим возникает вопрос, можно ли использовать такую слабую форму синхронизации при проектировании эффективных распределенных алгоритмов.

Мы уже приводили аргументы в пользу того, что лучше применять вполне асинхронные алгоритмы и не полагаться на допущения о согласовании времени.

1. Самый медленный компонент системы определяет (наряду с самым быстрым компонентом) значение архимедова коэффициента. Если сложность алгоритма зависит от этого коэффициента (как это обычно бывает), то быстродействие всей системы будет определяться быстродействием ее самого медленного компонента.

2. Анализ допущения о согласованности времени может оказаться чрезвычайно трудным, и это повлечет за собой неправильные алгоритмы.

3. Введение произвольных задержек при выполнении алгоритмов (как это имеет место, например, для кольцевых алгоритмов, используемых в алгоритме Дейкстры—Фейджена—ван Гастерена) может сделать несостоятельным допущение Архимеда. В результате эти алгоритмические методы нельзя будет применять в алгоритмах, опирающихся на допущение Архимеда.

4. Замена одного компонента системы на другой, более быстрый или медленный, может привести к изменению архимедова коэффициента, а это повлияет на алгоритмы и может привести к вычислению ошибочных результатов. Ошибки

такого рода чрезвычайно трудно отыскать и исправить, поскольку они возникают в результате замены одного компонента на другой, функционально эквивалентный исходному.

В свете этих доводов можно выделить два способа применения допущения Архимеда. Алгоритм называется *устойчивым* или *робастным*, если его правильность не зависит от этого допущения, т.е. алгоритм остается корректным и во вполне асинхронных системах. Допущение Архимеда используется при анализе сложности этого алгоритма, и оно позволяет обнаружить, что производительность алгоритма становится лучше, когда коэффициент близок к единице. Алгоритм называется *критическим*, если его правильность зависит от допущения Архимеда, т.е. этот алгоритм может вычислить неверный результат, если архимедов коэффициент превысит в ходе выполнения предусмотренное проектом значение s_0 . Методы «кодирования с использованием времени» и отправления «неявных сообщений» относятся к числу критичных по отношению к допущению Архимеда, тогда как метод «выборочной задержки» устойчив по отношению к нему.

В работе Витаньи [198] было показано, что если воспользоваться выборочной задержкой, то для архимедовых систем возможно проведение устойчивых выборов (см. упражнение 12.11). В работе Спиракиса и Тампакаса [178] представлен ряд устойчивых и критических распределенных алгоритмов решения различных задач.

12.6. Упражнения к главе 12

12.6.1

Упражнение 12.1. Усовершенствуйте протокол передачи информации посредством «кодирования с использованием времени» так, чтобы можно было передавать сообщение m при помощи фиксированного числа битов за $O(\sqrt{m})$ единиц времени.

Упражнение 12.2. В теореме 9.8 утверждается, что

если размер кольца не известен, то не существует детерминированных алгоритмов, завершающих вычисление функции f , отличных от константы, по окончании работы процессов.

Остается ли справедливой эта теорема и для синхронных сетей?

Упражнение 12.3. Докажите, что в алгоритме 12.1 всякий процесс может получить по ходу работы в i -м такте (в интервале $(2i\mu, 2(i+1)\mu)$ согласно показанию его локальных часов) только сообщения, относящиеся к i -му и $(i+1)$ -му тактам.

(Вследствие этого номер того такта, к которому относится сообщение, можно определить по времени его доставки адресату и индикатору *четности* номера того такта, к которому оно относится.)

Упражнение 12.4. Предположим, что этап синхронизации в алгоритме 12.1 расширен, за счет того что процесс p сохраняет в памяти показание часов $\delta_{pq} = CLOCK_p^{(\tau)}$ в момент времени τ получения процессом p сообщения $\langle \text{start} \rangle$ от

процесса q . При этом $\delta_{pq} = 0$, если получение этого сообщения повлекло за собой выполнение процессом p процедуры *init*. Докажите, что в момент получения процессом p от процесса q сообщения, относящегося к i -му такту, показания часов процесса p будут лежать в интервале между $\delta_{pq} + 2i\mu - \mu$ и $\delta_{pq} + 2i\mu + \mu$. (Вследствие этого номер такта, к которому относится сообщение, полученное от q в тот момент, когда часы показывают время s , может быть вычислен по формуле $\lfloor (s - \delta_{pq} + \mu)/2 \rfloor$, и поэтому отпадает необходимость включать в состав сообщения номер того такта, к которому оно относится.)

Упражнение 12.5. Внесите необходимые изменения в алгоритм 12.1, чтобы его можно было применять и в тех случаях, когда для выполнения внутренних действий процессов требуется отличное от нуля время. При этом предполагается, что время, необходимое для выполнения блока $\mathbf{P}_p^{(i)}$, ограничено сверху величиной λ .

12.6.2

Упражнение 12.6. Разработайте алгоритм распространения потока сообщений в кликах, торах, и гиперкубах с использованием $N - 1$ обменов сообщениями за $O(D)$ единиц времени. (Предполагается, что торы и гиперкубы размечены.)

Упражнение 12.7. Разработайте синхронный алгоритм избрания лидера в сетях, размер которых заранее неизвестен, при условии, что выборы не обязательно начинаются всеми участвующими процессами в одном и том же такте, а могут быть начаты разными процессами в разное время.

12.6.3

Упражнение 12.8. Оцените сложность по числу обменов сообщениями для асинхронного алгоритма избрания лидера, который получается в результате подключения синхронизаторов из § 12.6.3 к синхронным алгоритмам из § 12.6.2.

12.6.4

Упражнение 12.9. Докажите, что всякий сосед вершины, находящейся на f -м ярусе BFS-дерева, может лежать на одном из трех ярусов $f - 1$, f или $f + 1$.

Упражнение 12.10. Оцените сложность алгоритмов поиска в ширину относительно параметров N , $|E|$ и D (диаметр сети).

12.6.5

Упражнение 12.11 (см. [198]). Цель этого упражнения — исследовать тот эффект, который приносит выборочная задержка в архимедовых сетях. Пусть задана архимедова сеть с архимедовым коэффициентом s .

Выборы проводятся за счет применения принципа угасания к алгоритму обхода, сложность которого по числу обменов сообщениями равна W . Каждое сообщение процесса p задерживается каждым процессом на $f(p) - 1$ тактов. Особо выделенный процесс-будильник следит за тем, чтобы каждый процесс приступил к выполнению своей процедуры обхода спустя не более Du единиц времени после запуска алгоритма.

1. Докажите, что этот алгоритм завершит работу, спустя $Du + Wuf(p_0)$ единиц времени, где p_0 — это процесс с наименьшим отличительным признаком.

2. Докажите, что маркер процесса p будет передаваться не более $1 + t/(f(p) - 2)$ раз на протяжении интервала времени длины t .

3. Приведите формулу оценки сложности данного алгоритма по числу обменов сообщениями в наихудшем случае.

4. Покажите, выбрав подходящую функцию f , что для данного алгоритма может быть установлена линейная сложность по числу обменов сообщениями.

Г Л А В А 13

ОТКАЗОУСТОЙЧИВОСТЬ В РАСПРЕДЕЛЕННЫХ СИСТЕМАХ

В предыдущих главах этой книги мы изучали, как достигается согласованность поведения распределенных систем, которые состоят из *надежных* процессов. В силу ряда причин было бы интересно выяснить, насколько согласованным может быть поведение процессов, если предполагать, что некоторые из них могут выходить из строя. Изучению этого вопроса как раз и посвящена последняя часть книги. Оказывается, многие методы повышения отказоустойчивости пригодны только для частных случаев этой задачи, и совсем непросто подвести единую теоретическую базу под многочисленные отрицательные результаты. С другой стороны, для некоторых задач создана изящная и хорошо разработанная теория и предложены простые решения, которые вполне заслуживают того, чтобы представить их в этом учебном пособии.

Эта глава является общим введением ко всем последующим главам. В § 13.1 мы объясним, зачем нужно использовать отказоустойчивые алгоритмы, а затем введем два типа отказоустойчивых алгоритмов: устойчивые, или робастные, алгоритмы (§ 13.2) и стабилизирующие алгоритмы (§ 13.3).

13.1. Зачем нужны отказоустойчивые алгоритмы

С увеличением числа компонентов распределенной системы увеличивается и вероятность того, что некоторые из этих компонентов будут выходить из строя при выполнении распределенного алгоритма. В компьютерах сети могут происходить сбои, процессы системы могут быть уничтожены по ошибке при выключении рабочей станции, вычислительная машина может выдать неверный результат в связи с неполадками в работе запоминающего устройства. Современные компьютеры становятся все более и более надежными, и вероятность возникновения ошибок такого рода в каждой отдельной вычислительной машине непрерывно снижается. Вместе с тем вероятность неисправности, которая может произойти в какой-нибудь части распределенной системы, может стать сколь угодно большой с увеличением числа компонентов системы. Во избежание необходимости перезапускать алгоритм всякий раз, когда случается сбой в работе одного из компонентов системы, алгоритмы должны быть устроены так, чтобы они могли правильным образом справляться с такими неполадками.

Безусловно, возможность возникновения неисправности принимается в расчет и в последовательных вычислениях, особенно в тех программных продуктах, которые являются критическими с точки зрения безопасности, или при проведе-

нии продолжительных вычислений, результаты которых нельзя проверить. В центральном процессоре есть средства проверки, обеспечивающие защиту от ошибок некоторого вида (например, контрольные суммы позволяют обнаружить арифметические ошибки), но никакие средства защиты не помогут, конечно, в тех случаях, когда будет уничтожен или поврежден код программы. Поэтому возможности отказоустойчивых вычислений на основе последовательных алгоритмов в однопроцессорных вычислительных системах ограничены.

Ввиду того что вычислительные ресурсы всякой распределенной системы расщеплены, эти системы обладают свойством *частичной неисправности*: какова бы ни была неполадка, она обычно выводит из строя только некоторую часть всей системы. И хотя с ростом числа компонентов системы стремительно возрастает и вероятность отказа одного из этих компонентов, столь же стремительно уменьшается вероятность выхода из строя всех компонентов системы. Поэтому можно рассчитывать на то, что все те функции, которые были возложены на вышедшие из строя процессы, возьмут на себя уцелевшие компоненты системы, и вместо полной потери функциональных возможностей системы мы получим лишь постепенное их сокращение. И действительно, как будет показано в последующих главах, распределенные алгоритмы можно проектировать и для систем, содержащих дефектные процессы, добиваясь при этом их согласованного поведения.

Благодаря свойству частичного отказа распределенные архитектуры с дублированием могут быть использованы для разработки таких компьютерных приложений, от которых требуется высокая надежность работы, хотя по своей сути они не являются распределенными системами. Примером тому может служить главная вычислительная система космического корабля Space Shuttle; о ее разработке Спектор и Джиффорд рассказывают в книге [177]. Для решения задачи управления космическим кораблем вполне достаточно одного-единственного стандартного микропроцессора, но при проектировании этого процессора серьезные опасения вызвала возможность его выхода из строя во время полета. В окончательном виде система управления была составлена из четырех совершенно идентичных процессоров, и каждый из них должен был выполнять одно и то же вычисление¹⁾. Исполнительные блоки, получая выходные данные, проводили их сравнение (простым голосованием), и это позволяло сохранять возможности системы управления во всей полноте даже после выхода из строя одного процессора. (Физическая реализация исполнительных блоков позволяла системе сохранять работоспособность и в случае последующего выхода из строя второго процессора.) Хотя дублирование функций представляется весьма привлекательным способом повышения надежности, устройство алгоритмов, предназначенных для согласования работы целого кластера (вообще говоря, ненадежных) процессоров, не такое уж и простое. Об этом как раз свидетельствует задержка первого запуска космического корабля Shuttle (перенесенная с 8 апреля на 10

¹⁾В книге [177] рассказывается также о назначении пятого («горячий резерв») и шестого («холодный резерв») процессоров. — *Прим. автора*

апреля 1981 г.) в связи с ошибкой в программном обеспечении, которое как раз было предназначено для этой цели.

К счастью, с 1981 г. исследования в области отказоустойчивых алгоритмов значительно продвинулись, и в настоящее время надежные прикладные программы, основанные на принципе дублирования, вполне доступны.

В научной литературе выделяют два подхода к построению отказоустойчивых систем, и мы рассмотрим в этой книге оба подхода. В *робастных алгоритмах* любой процесс совершает каждый шаг вычисления достаточно осторожно, и поэтому, несмотря на сбои в системе, исправные процессы совершают только правильные действия. В *стабилизирующихся алгоритмах* исправные процессы могут выходить из строя, но при этом гарантируется, что правильное выполнение алгоритма будет восстановлено из любой конфигурации, как только правильная работа процессов возобновится. Мы немного расскажем о робастных алгоритмах в § 13.2, познакомимся со стабилизирующимися алгоритмами в § 13.3, и проведем их короткий сравнительный анализ в конце этой главы.

13.2. Робастные алгоритмы

Робастные алгоритмы проектируются так, чтобы непрерывно обеспечивать правильное функционирование исправно работающих процессов, невзирая на то, что в ходе выполнения другие процессы могут выходить из строя. В этих алгоритмах применяются различные стратегии наподобие голосования, руководствуясь которыми процесс принимает определенную информацию только тогда, когда достаточно много других процессов уже объявили о том, что они приняли эту информацию. Однако процесс никогда не должен дожидаться получения информации от всех остальных процессов, потому что в таком случае обязательно возникнет блокировка, как только в одном из процессов возникнет неполадка.

13.2.1. Модели неисправностей

Для того чтобы выяснить, каким образом исправно работающие процессы могут обезопасить себя от неисправных процессов, необходимо сделать определенные допущения о том, *как именно* процессы могут выходить из строя. В последующих главах мы будем всегда исходить из предположения о том, что только *процессы* могут выходить из строя, а каналы остаются исправными. Таким образом, если один исправный процесс отправляет сообщение другому исправному процессу, то это сообщение обязательно будет доставлено адресату спустя конечный промежуток времени. Неисправность канала связи можно промоделировать, если объявить неисправным один из процессов, инцидентных этому каналу (так называемая ошибка пропуска). Кроме того, мы будем всегда полагать, что каждый процесс может отправить сообщение любому другому процессу.

В этой книге мы будем иметь дело со следующими *моделями неисправностей*.

1. *Изначально бездействующие процессы.* Процесс называется изначально бездействующим, если его локальный алгоритм не совершает ни одного действия.

2. *Модель выхода процесса из строя.* Мы будем говорить, что процесс выходит из строя, если до некоторого момента времени его локальный алгоритм работает правильно, а затем не выполняет ни одного действия.

3. *Византийское (предательское) поведение.* Процесс называется византийским (или предательским), если он выполняет сколь угодно много действий, которые не предписаны его локальным алгоритмом. В частности, византийский процесс может отправлять сообщения с произвольным содержанием. Требования корректности для робастных алгоритмов выдвигаются только в отношении локальных состояний (или выходных данных) *исправных* процессов. Изначально бездействующий процесс никогда не вычисляет выходных данных и всегда пребывает в начальном состоянии. Если процесс вышел из строя, то его выходные данные (если таковые были получены) будут вычислены правильно, потому что до того, как процесс вышел из строя, он функционировал правильно. Само собой разумеется, что как локальное состояние, так и выходные данные византийского процесса могут быть произвольными, и в таком случае никакое нетривиальное требование не может быть выполнено.

Иерархия моделей неисправности. Эти три модели неисправности образуют иерархию. Во-первых, изначально бездействующий процесс можно рассматривать как частный случай процесса, вышедшего из строя, — поломка произошла еще до того, как в этом процессе произошло первое событие. Во-вторых, процесс, вышедший из строя, можно рассматривать как частный случай византийского процесса, поскольку произвольное поведение византийского процесса подразумевает и возможность не совершать ни одного шага выполнения. Поэтому приспособить систему к выходу процессов из строя труднее, чем справиться с изначально бездействующими процессами, но гораздо легче, чем сделать ее устойчивой по отношению к византийским процессам. Иначе говоря, всякий алгоритм, который устойчив по отношению к византийским процессам, является устойчивым и по отношению к выходу процессов из строя, и любой алгоритм, который устойчив по отношению к выходу процессов из строя, является устойчивым по отношению к изначально бездействующим процессам. С другой стороны, невозможность построения алгоритма, устойчивого по отношению к изначально бездействующим процессам, означает, что нельзя построить алгоритм, который был бы устойчив по отношению к выходу процессов из строя, и невозможность построения алгоритма, устойчивого по отношению к выходу процессов из строя, делает неосуществимым построение алгоритма, который был бы устойчив по отношению к византийским процессам.

Неисправность пропущенного действия заключается в том, что алгоритм пропускает какой-то шаг своего выполнения (например, отправку или прием сообщения) и после этого продолжает вычисление. Неисправность пропущенного действия является частным случаем византийского поведения, а выход процесса из строя является частным случаем неисправности пропущенного действия,

при которой начиная с некоторого момента пропускаются *все* шаги алгоритма. Поэтому в рассмотренной иерархии неисправность пропущенного действия располагается между выходом процесса из строя и византийской неисправностью.

Смешанные неисправности и ошибки отсчета времени. Изначально бездействующие и вышедшие из строя процессы относятся к так называемой «доброкачественной» форме неисправностей, тогда как византийские процессы называют «злокачественной» формой неисправности. Оказывается, для некоторых задач, связанных с распределенными вычислениями, система, состоящая из N процессов, может быть стойкой по отношению к t «доброкачественным» неисправностям, если $2t < N$, в то время как для устойчивости по отношению к t «злокачественным» неисправностям требуется, чтобы выполнялось неравенство $3t < N$ (в *асинхронной* модели вычисления). Поскольку «злокачественные» неисправности возникают очень редко по сравнению с «доброкачественными» неисправностями, Гейрай и Перри [97] распространили некоторые результаты на модели со смешанными неисправностями, в которых допускается поломка t процессов, причем b из них имеют «злокачественную» форму неисправности. В этой модели оставшиеся процессы могут сохранить правильное поведение, если выполняется неравенство $2b + t < N$ (в *синхронной* модели вычисления).

В синхронных распределенных системах возможна еще одна разновидность неисправности, которая проявляется в том, что процесс выполняет правильное действие не вовремя из-за сбоя в работе часов этого процесса. Этот тип неправильного поведения процессов получил название *ошибки отсчета времени*.

13.2.2. Проблемы принятия решения

При изучении робастных алгоритмов основное внимание уделяется исследованию так называемая *проблем принятия решения*, в которых требуется, чтобы каждый (исправный) процесс в качестве результата своей работы осуществлял необратимую запись некоторого значения, определяющего решение, принятое процессом. Требуемое значение решения обычно находится в очень простой зависимости от исходных данных, которые получают процессы, и поэтому эту задачу можно весьма легко решить в надежной вычислительной среде при отсутствии неисправностей.

К решениям обычно предъявляются требования следующих трех типов: *завершаемость*, *непротиворечивость* и *невыврожденность*.

Требование завершаемости гласит, что все исправно работающие процессы должны завершить свою работу, т.е. рано или поздно осуществить запись принятого решения в качестве выходных данных. В детерминированных алгоритмах требуется, чтобы все вычисления завершались; в вероятностных алгоритмах требуется, чтобы вычисления завершались с вероятностью 1 (см. гл. 9). Требуя, чтобы все исправно работающие процессы осуществили запись вычисленных значений в качестве выходных данных, мы тем самым исключаем всякое решение, в котором процесс должен ожидать поступления какой-либо информации от более чем $N - t$ процессов. В таком случае выход из строя t процессов приводил бы

к тому, что исправный процесс пребывал бы в неопределенно долгом ожидании, вопреки требованию завершения.

Требование непротиворечивости устанавливает некоторые отношения между решениями, принятыми разными процессами. В простейшем случае требуется, чтобы все решения были одинаковыми; в таком случае говорят о *проблеме консенсуса* (при этом требуемое значение решения может зависеть от входных данных). В более сложных задачах определяется класс выходных векторов и решения, принятые исправными процессами, должны образовывать вектор из этого класса. Например, в задаче избрания лидера (гл. 7) требуется, чтобы один из процессов принял решение о том, что он *избран лидером*, и записал «1», а все остальные процессы приняли решение о том, что они проиграли выборы, и записали «0».

Требований завершаемости и непротиворечивости обычно бывает достаточно, чтобы признать распределенный алгоритм полезным. В тех случаях, когда устанавливается, что задачу нельзя решить, нужно ввести еще одно дополнительное техническое требование. Требование невырожденности призвано исключить из рассмотрения те алгоритмы, в которых процессы принимают решения без взаимодействия друг с другом и которые выдают одни и те же фиксированные выходные данные в качестве решения задачи. Например проблему консенсуса можно было бы решить при помощи алгоритма, в котором каждый процесс выдает немедленно в качестве ответа «0». Требование невырожденности гласит, что различные выполнения алгоритма могут приводить по крайней мере к двум существенно отличным друг от друга ответам (например, в случае консенсуса алгоритм должен иметь выполнения, которые приводят к записи «0», и выполнения, которые приводят к записи «1»; при этом, разумеется, в одном и том же выполнении решения, принятые всеми процессами, должны быть одинаковыми).

Как мы увидим, проблемы принятия решения можно рассматривать как абстракции широкого класса задач, возникающих в распределенных вычислениях.

1. *Совершить—отказаться*. В распределенных базах данных транзакция, в которой участвуют несколько сайтов, может быть выполнена либо *во всех* участвующих в ней сайтах, либо *ни в одном* из них. Поэтому после объявления о преобразовании данных в этих сайтах каждый сайт должен определить, может ли это преобразование быть осуществлено локально, и принять решение либо об осуществлении преобразования, либо об отказе от него. Таким образом, все исправно работающие сайты должны решить, следует ли им всем *совершить* преобразование или *отказаться* от него. Если все процессы за выполнение преобразования, то оно должно быть совершено, а если хоть один процесс выскажет против его выполнения, то все процессы должны отказаться от этого преобразования. Непротиворечивость в данном случае означает, что окончательные решения всех процессов должны быть одинаковыми. Проблема является невырожденной, поскольку в зависимости от входных данных система может совершить преобразование или отказаться от него.

2. *Распределенные вычисления: консенсус по входным данным*. В некоторых системах для повышения надежности вычисления дублируются; в таком

случае результаты вычисления на разных процессорах будут одинаковыми только в том случае, когда были одинаковыми исходные данные. Влияние дефектного сенсора, отправляющего разные значения разным процессорам, должно быть устранено за счет выполнения алгоритма достижения консенсуса между процессорами. Входными данными для каждого процессора служат значения, поступающие от сенсора, и каждый исправный процессор принимает решение на основе одних и тех же данных, которые будут использованы в последующем вычислении. Обычно требуется, чтобы в качестве выходного результата выступало то значение, которое поступает на большее число процессоров, или, в крайнем случае, которое возникает хотя бы на одном входе. В обоих случаях задача совсем не проста.

3. *Избрание лидера.* В задаче избрания лидера (см. также гл. 7) требуется, чтобы один из процессов принял решение стать лидером, а остальные (исправные) процессы пришли к решению отказаться от лидерства. Эта задача становится нетривиальной, если несколько разных процессов имеют потенциальную возможность стать лидером.

4. *Приближенное соглашение.* В тех случаях, когда нельзя достичь консенсуса по входным данным, для некоторых прикладных задач достаточно обеспечить более слабую форму согласованности. В задаче приближенного соглашения каждый процесс имеет на входе целое число из некоторого конечного множества, например из отрезка $1, \dots, k$. Требуется, чтобы решения, принятые процессами, отличались не более чем на 1 и находились в тех пределах, в которых находятся те числа, которые подаются на вход. Непрерывный вариант этой задачи (ϵ -приближенное соглашение) рассматривается в § 14.6.3.

13.2.3. Краткое содержание глав 14–16

Было получено очень много результатов, касающихся робастных алгоритмов, и в этой книге просто невозможно полностью осветить современное положение дел в этой области. При отборе материала для глав 14–16 автор руководствовался следующими соображениями.

1. Необходимостью включить в эти главы ряд основополагающих результатов об отсутствии детерминированного консенсуса для асинхронных систем, а также о том, что вероятностные алгоритмы устойчивы по отношению к «доброкачественным» формам неисправности, охватывающим не более половины процессов, и к «злонакачественным» формам неисправности, охватывающим не более одной трети процессов.

2. Желанием продемонстрировать те методы, которые применяются как для достижения устойчивости, так и для обоснования невозможности ее достичь.

3. Стремлением показать, что синхронные системы являются более стойкими, чем асинхронные. Это заключение может быть противопоставлено результатам, представленным в главе 12, которые свидетельствуют о том, что вычислительные возможности надежных синхронных систем не превосходят возможности надежных асинхронных систем.

4. Пониманием того, что детекторы неисправностей представляют собой новую парадигму, которая открывает возможности внедрения новых прикладных технологий.

В гл. 14 изучаются вопросы о том, какого рода стойкости можно достичь для *асинхронных* систем. В работе Фишера, Линча и Патерсона [81] был получен основополагающий результат, гласящий, что не существует детерминированного алгоритма достижения консенсуса, который был бы устойчив даже по отношению к выходу из строя одного процесса. Моран и Вольфштал в своей работе [151] обобщили этот результат для более широкого класса задач принятия решения, который включает в себя задачу избрания лидера, но не включает задачу приближенного соглашения. Из этих работ можно прийти к выводу о том, что устойчивыми по отношению к поломкам могут быть только вероятностные алгоритмы (или синхронные системы). Эти результаты нельзя распространить на более слабые модели неисправности с изначально бездействующими процессами: как будет показано, в тех случаях, когда неисправность проявляет себя только в виде изначально бездействующих процессов, существуют детерминированные алгоритмы избрания лидера и достижения консенсуса. Рандомизация может оказаться очень полезной. Брача и Туэг в работе [35] построили рандомизованный алгоритм достижения консенсуса для асинхронных систем, который устойчив по отношению к выходу из строя $t < N/2$ процессов и к византийскому поведению $t < N/3$ процессов.

В гл. 15 изучаются вопросы о том, какого рода стойкости можно достичь для *синхронных* систем. В отличие от асинхронных систем, выход процесса из строя может быть обнаружен оставшимися процессами, поскольку для исправного процесса имеется ограниченное время отклика. В связи с этим удается достичь гораздо большей степени устойчивости, что показано на примере идеализированной модели системы, работающей в потактовом (импульсном) режиме (см. гл. 12). Детерминированные алгоритмы могут быть устойчивы по отношению к византийскому поведению $t < N/3$ процессов, к выходу из строя $t < N$ процессов, а в случае использования аутентификационных сообщений и по отношению к византийскому поведению $t < N$ процессов. Отсюда, а также из результатов гл. 14 следует вывод о невозможности построения стойкого синхронизатора для вполне асинхронных сетей. Для реализации потактового (импульсного) режима работы в системах с ограниченной асинхронностью (обладающий физическими часами и ограниченным временем задержки при доставке сообщений) требуется, чтобы часы были тщательно синхронизованы. Эта задача рассматривается в последнем параграфе гл. 15.

В гл. 16 обсуждается метод обнаружения неисправностей, который, наряду с синхронизацией, также позволяет усилить вычислительную модель; эта возможность в той или иной форме осуществима в распределенных системах, и связана с обнаружением неисправностей. Подобно тому как на основе сравнительного анализа различных моделей синхронизации можно понять, какие средства нужно ввести, чтобы решить задачу, мы можем изучить модели обнаружения неисправностей, чтобы понять, насколько точно их результат отражает характер неисправности, и воспользоваться этим для решения задачи консенсуса.

13.2.4. Темы, которые не были включены в эту книгу

Результаты, представленные в гл. 14 и 15, показывают, до какой степени надежными могут быть распределенные системы. Многие задачи и результаты оставлены читателям для дальнейшего самостоятельного изучения; мы обозначим лишь некоторые из них, указав основные публикации на эти темы.

1. *Уточнение допущений о синхронизации.* В этой книге рассматриваются только вполне синхронные и вполне асинхронные системы. Эти два типа систем значительно отличаются друг от друга по той степени устойчивости, которая может быть для них достигнута. Долев, Дворк, Линч и Стокмейер исследовали в работах [74] и [75] вопросы отказоустойчивости для систем промежуточного типа, в которых действуют ослабленные допущения о синхронизации.

2. *Характеризация разрешимых задач.* В этой книге была установлена разрешимость одних задач и неразрешимость других. Уточнению характеристических свойств разрешимых и неразрешимых задач посвящена работа Байрана, Морана и Закса [29].

3. *Сложность отказоустойчивости.* Помимо вопроса о том, какие задачи имеют решения, заслуживает внимания также и вопрос о том, какой объем вычислительных ресурсов используется протоколом для разрешимых задач. Можно рассматривать различные меры сложности: сложность по числу обменов сообщениями, битовую сложность, сложность по времени (которую в этом контексте часто называют раундовой сложностью). Обзор некоторых результатов в этом направлении, а также описание протоколов, в которых одновременно достигаются различные нижние оценки сложности, представлен в работе [28].

4. *Динамические системы и групповая принадлежность.* В этой книге предполагается, что вся совокупность процессов, образующих систему, заранее задана и известна (хотя при этом некоторые процессы могут выходить из строя). Такой статический способ формирования системы удобен в тех случаях, когда от нее требуется надежная работа в течение заданного ограниченного промежутка времени, как это имеет место для систем управления воздушными судами, когда ремонт и изменение конфигурации производится во внерабочее время. Отказоустойчивая работа систем в течение неопределенно долгого времени (как это имеет место, например, для операционной системы управления воздушным движением) предполагает проведение ремонта и изменение ее конфигурации во время ее работы без прерывания вычислений. Реконфигурация семейства процессов осуществляется путем выполнения протокола *групповой принадлежности*, в ходе которого между действующими и введенными процессами системы достигается соглашение. Некоторые результаты, относящиеся к этим протоколам, а также необходимые ссылки можно отыскать в работе [162].

5. *Коммуникация при помощи совместно используемых переменных.* В этой книге рассматриваются системы, в которых связь осуществляется посредством обменов сообщениями. В работе Таубенфельда и Морана [185] было проведено изучение отказоустойчивости распределенных систем, в которых коммуникация осуществляется при помощи совместно используемых переменных.

6. *Синхронизация без простоя.* Еще более замысловатой выглядит коммуникационная модель, в которой процессы совместно используют произвольные объекты, а не только отдельные регистры памяти. В этой модели желаемое взаимодействие между процессами можно описать в объектных терминах, а сами алгоритмические проблемы формулируются как задачи реализации нужных объектов. В работе [12] представлена хорошо разработанная теория о том, какие объекты могут или не могут быть реализованы.

13.3. Стабилизирующиеся алгоритмы

Робастные алгоритмы постоянно придерживаются правильного согласованного поведения даже в случае возникновения неисправностей, но для их реализации требуется, чтобы количество ошибок было ограничено, и точная модель неисправности обычно должна быть известна заранее. Совсем иным является другой подход к решению проблемы отказоустойчивости, связанный со стабилизирующимися алгоритмами. Здесь допускается сколь угодно большое количество ошибок произвольного типа, и правильная работа алгоритма восстанавливается спустя некоторое время, после того как эти неисправности будут устранены. *Стабилизирующийся* алгоритм (иногда его называют *самостабилизирующимся*) может быть запущен из любой системной конфигурации, рано или поздно достигает требуемого состояния и далее ведет себя в соответствии со своей спецификацией. Таким образом, влияние временных неисправностей непродолжительно, и необходимости проводить инициализацию не возникает.

Стабилизирующиеся алгоритмы изучаются в гл. 17. Сама идея стабилизации была высказана в 1974 г. работе Дейкстры [61] и применена к алгоритмам, обеспечивающим взаимно исключенный доступ для систем процессоров, образующих кольцо. Эти алгоритмы заслуживают внимания из-за их непреходящего исторического значения, а также потому, что они очень изящны и требуют определенной проницательности. Стабилизирующиеся решения существуют и для ряда других задач, которые рассматриваются в этой книге; к их числу относятся задачи передачи данных, избрания лидера, вычисления таблиц маршрутизации и построения деревьев поиска в глубину.

Сравнительный анализ робастных и стабилизирующихся алгоритмов.

Стабилизирующиеся алгоритмы служат для защиты от так называемых *кратковременных неисправностей*, вызванных сбоями в работе отдельных компонентов системы. Такие сбои могут случаться в больших разделах распределенной системы, когда физические условия ее функционирования достигают предельного уровня и вследствие этого процессоры и устройства памяти могут срабатывать ошибочно. В качестве примеров можно взять систему управления космическим кораблем, которая подверглась воздействию интенсивного потока космических лучей, или систему, многие компоненты которой одновременно подверглись воздействию какого-нибудь природного катаклизма. Как только нормальные условия восстанавливаются, процессы возобновляют свою работу в соответствии с их программами, но из-за кратковременного сбоя в работе глобальное состояние

системы (конфигурация) может быть произвольным. Стабилизирующие свойства призваны гарантировать постепенное восстановление требуемого поведения.

Робастные алгоритмы предназначены для защиты от *устойчивых неисправностей*, возникающих в ограниченном числе компонентов. Процессы, оставшиеся исправными, поддерживают правильное (хотя, возможно, и менее эффективное) поведение системы, пока проводится ее ремонт и реконфигурация. Таким образом, робастные алгоритмы нужно использовать тогда, когда недопустимо даже кратковременное прерывание обслуживания.

В работах Гопала и Перри [100], а также Анагностоу и Хадзилакоса [7] изучаются алгоритмы, которые являются одновременно и робастными, и стабилизирующимися. Гопал и Перри показали, как можно автоматически превратить (транслировать) всякий робастный протокол в такой протокол, который будет и робастным, и стабилизирующимся. Анагностоу и Хадзилакос установили, что для задач избрания лидера и вычисления размера кольца нельзя построить протокол, который был бы одновременно и робастным, и стабилизирующимся, но предложили рандомизированный протокол наделения процессов разными именами.

ГЛАВА 14

ОТКАЗОУСТОЙЧИВОСТЬ В АСИНХРОННЫХ СИСТЕМАХ

В этой главе изучается разрешимость проблем принятия решения в асинхронных распределенных системах. В центре нашего внимания будет результат, полученный в работе Фишера, Линч и Патерсона [81] и представленный в § 14.6.1. В нем дается доказательство невозможности построения целого класса алгоритмов принятия решения. В то же время его можно расценивать как список тех допущений об устройстве системы, которые в совокупности исключают получение положительного решения рассматриваемой проблемы. Как будет показано в последующих параграфах, при ослаблении этих допущений можно получать практические решения. Более подробное обсуждение этого вопроса проводится в § 14.1.3.

14.1. Невозможность консенсуса

В этом параграфе будет доказана основополагающая теорема Фишера, Линч и Патерсона [81], гласящая, что не существует асинхронного детерминированного протокола, который был бы устойчив к выходу из строя одного процесса. Для получения этого результата приходится проводить анализ справедливых последовательностей выполнения алгоритмов. Вначале мы введем специальную систему обозначений (в дополнение к той, которая была введена в § 2.5.1) и приведем элементарные результаты, которые будут использоваться в последующих параграфах.

14.1.1. Обозначения, определения и простейшие результаты

Последовательность событий $\sigma = (e_1, \dots, e_k)$ называется *применимой* к конфигурации γ , если событие e_1 допустимо в конфигурации γ , событие e_2 допустимо в конфигурации $e_1(\gamma)$ и т. д. Если в результате этого применения будет достигнута конфигурация δ , то для обозначения этого мы будем использовать запись $\gamma \rightsquigarrow^\sigma \delta$ или $\sigma(\gamma) = \delta$, чтобы явным образом обозначить те события, которые происходят при переходе от γ к δ . Если $S \subseteq \mathbb{P}$ и последовательность σ состоит только из тех событий, которые происходят в процессах множества S , то мы будем также использовать запись $\gamma \rightsquigarrow_S \delta$.

Предложение 14.1. Пусть последовательности σ_1 и σ_2 применимы к конфигурации γ и при этом ни один процесс не участвует одновременно в обеих последовательностях σ_1 и σ_2 . Тогда последовательность σ_2 при-

менима к конфигурации $\sigma_1(\gamma)$, а последовательность σ_1 применима к конфигурации $\sigma_2(\gamma)$ и при этом $\sigma_2(\sigma_1(\gamma)) = \sigma_1(\sigma_2(\gamma))$.

Доказательство. Достаточно последовательно применять теорему 2.19. $\square$

Процесс p имеет входную переменную x_p , которая доступна только для считывания, и выходной регистр y_p , который доступен только для однократной записи и в котором вначале содержится значение b . Для каждого процесса исходная конфигурация полностью определяется значением переменной x_p . Процесс p может *принять решение*, выработав одно из двух значений (обычно 0 или 1) и записав его в регистр y_p ; начальное значение b не считается показателем принятия решения. Предполагается, что в исправно работающем процессе происходит бесконечно много событий на протяжении справедливого выполнения; чтобы соблюсти это допущение, мы будем считать, что в процессе всегда может происходить (возможно, незначущее) внутреннее событие.

Определение 14.2. *Справедливое выполнение с t вышедшими из строя процессами* — это выполнение, в котором бесконечно много событий происходит в каждом из не менее чем $N - t$ процессов и каждое сообщение, отправленное исправному процессу, поступает по назначению. (Процесс работает исправно, если в нем происходит бесконечно много событий.)

Наибольшее число вышедших из строя процессов, поломка которых не повлекла за собой неправильной работы алгоритма, называется *стойкостью* алгоритма; в дальнейшем стойкость алгоритма мы будем обычно обозначать буквой t . В этом параграфе будет показана невозможность построения асинхронного детерминированного алгоритма со стойкостью, равной единице.

Определение 14.3. *Алгоритмом консенсуса, робастным относительно одного отказа*, называется алгоритм, который удовлетворяет следующим трем требованиям.

1. **Завершаемость.** В ходе каждого справедливого выполнения с одним вышедшим из строя процессом все исправно работающие процессы принимают решение.
2. **Договоренность.** Если в достижимой конфигурации для исправных процессов p и q выполняются неравенства $y_p \neq b$ и $y_q \neq b$, то $y_p = y_q$.
3. **Нетривиальность.** Для $v = 0$ и для $v = 1$ существуют достижимые конфигурации, в которых для одного из процессов p выполняется равенство $y_p = v$.

В случае $v = 0, 1$ конфигурация называется *конфигурацией принятия v -решения*, если для какого-то процесса p верно равенство $y_p = v$; всякая конфигурация 0-решения или 1-решения называется *конфигурацией принятия решения*. В конфигурации принятия v -решения некоторые процессы уже приняли решение v . Конфигурация γ называется *v -валентной*, если во всех конфигурациях принятия решения, которые достижимы из γ , принимается решение v . Конфигурация называется *бивалентной*, если из нее достижимы как конфигурации принятия 0-решения, так и конфигурации принятия 1-решения,

и *унивалентной*, если она является либо 1-валентной, либо 0-валентной. Как следует из определения, в унивалентной конфигурации решение уже предопределено, несмотря на то что ни один процесс еще, быть может, не принял решения.

Конфигурация γ , возникающая при выполнении t -робастного протокола, называется *разветвлением*, если существуют такое множество T , состоящее не более чем из t процессов, а также такие конфигурации γ_0 и γ_1 , что $\gamma \rightsquigarrow_T \gamma_0$, $\gamma \rightsquigarrow_T \gamma_1$ и при этом конфигурация γ_v является v -валентной. Содержательный смысл введенного понятия таков: конфигурация γ является разветвлением, если некоторое подмножество, состоящее из t процессов, может привести к выработке как 0-решения, так и 1-решения. В следующем ниже утверждении формально выражается условие сохранения дееспособности системы, в которой вышли из строя не более t процессов.

Предложение 14.4. *Для каждой достижимой конфигурации t -робастного алгоритма для каждого подмножества S , содержащего не менее $N - t$ процессов, существует такая конфигурация принятия решения δ , для которой выполняется соотношение $\gamma \rightsquigarrow_S \delta$.*

Доказательство. Рассмотрим такую конфигурацию γ и такое множество процессов S , которые были указаны в формулировке леммы. Рассмотрим также какое-нибудь выполнение, в котором достигается конфигурация γ и после этого происходит бесконечно много событий в каждом процессе семейства S (но при этом ни один процесс, не принадлежащий множеству S , не совершает ни одного шага). Это выполнение является справедливым выполнением с t вышедшими из строя процессами, и все процессы из семейства S работают исправно. Следовательно, будет достигнута конфигурация принятия решения. $\square$

Лемма 14.5. *Не существует достижимого разветвления.*

Доказательство. Рассмотрим достижимую конфигурацию γ и множество T , состоящее не более чем из t процессов.

Множество всех остальных процессов системы обозначим символом S , т. е. $S = \mathbb{P} \setminus T$. Так как в множестве S содержится не менее $N - t$ процессов, существует такая конфигурация принятия решения δ , что $\gamma \rightsquigarrow_S \delta$ (утверждение 14.4). Конфигурация δ является либо конфигурацией принятия 0-решения, либо конфигурацией принятия 1-решения. Не ограничивая общности, будем считать, что она является конфигурацией принятия 0-решения.

Теперь убедимся в том, что ни для одной 1-валентной конфигурации γ' не выполняется соотношение $\gamma \rightsquigarrow_T \gamma'$. Допустим, что для некоторой конфигурации γ' имеет место соотношение $\gamma \rightsquigarrow_T \gamma'$. Поскольку шаги выполнения процессов из множества T и из множества S можно переставлять (утверждение 14.1), найдется такая конфигурация δ' , которая достижима как из конфигурации δ , так и из конфигурации γ' . Поскольку δ — это конфигурация принятия 0-решения, такой же должна быть и конфигурация δ' , и это означает, что конфигурация γ' не является 1-валентной. $\square$

14.1.2. Доказательство невозможности построения робастных асинхронных систем

Вначале мы воспользуемся нетривиальностью задачи, для того чтобы убедиться в существовании бивалентной начальной конфигурации (лемма 14.6). Вслед за этим мы покажем, что в выполнении, начинающемся в этой бивалентной конфигурации, каждый допустимый шаг можно совершить так, что при этом не происходит переход в унивалентную конфигурацию (лемма 14.7). Этого достаточно, чтобы прийти к выводу о невозможности построения алгоритмов консенсуса (теорема 14.8). Далее символ A мы будем использовать для обозначения произвольного алгоритма консенсуса, робастного по отношению к выходу из строя одного процесса.

Лемма 14.6. *Для алгоритма A существует бивалентная начальная конфигурация.*

Доказательство. Так как алгоритм A является нетривиальным (см. определение 14.3), для него существуют как конфигурации принятия 0-решения, так и конфигурации принятия 1-решения. Рассмотрим две такие начальные конфигурации δ_0 и δ_1 , что из δ_v , $v = 0, 1$, достижима некоторая конфигурация принятия v -решения.

Если $\delta_0 = \delta_1$, то эта начальная конфигурация как раз и является искомой бивалентной конфигурацией. В противном случае найдутся две такие начальные конфигурации γ_0 и γ_1 , что из δ_v , $v = 0, 1$, достижима некоторая конфигурация принятия v -решения и при этом γ_0 и γ_1 отличаются входными данными только одного процесса. Действительно, рассмотрим последовательность начальных конфигураций, начинающуюся конфигурацией δ_0 и оканчивающуюся конфигурацией δ_1 , в которой каждая следующая начальная конфигурация отличается от предыдущей начальными данными только одного процесса. Эта последовательность получается путем инвертирования входных битов одного за другим. Из первой конфигурации этой последовательности δ_0 достижима конфигурация принятия 0-решения, а из последней конфигурации этой последовательности, δ_1 , достижима конфигурация принятия 1-решения. Так как конфигурация принятия решения достижима из каждой начальной конфигурации, те две конфигурации γ_0 и γ_1 , которые были описаны выше, будут соседними членами этой последовательности. Обозначим буквой p тот процесс, который имеет разные входные данные в конфигурациях γ_0 и γ_1 .

Рассмотрим справедливое выполнение, начинающееся конфигурацией γ_0 , в котором процесс p не выполняет ни одного шага; оно будет справедливым выполнением с одним вышедшим из строя процессом, и поэтому в ходе этого выполнения будет достигнута конфигурация принятия решения β . Если β является конфигурацией принятия 1-решения, то γ_0 — бивалентная конфигурация. Если же β является конфигурацией принятия 0-решения, то заметим, что γ_1 отличается от γ_0 только данными процесса p , а процесс p не принимает участия в выполнении. Следовательно, конфигурация β достижима из γ_1 , и это означает, что γ_1 — бивалентная конфигурация. (Точнее говоря, из конфигурации γ_1 можно достичь

такой конфигурации β' , которая отличается от β только состоянием процесса p ; следовательно, β' является конфигурацией принятия 0-решения.) $\square$

Чтобы построить справедливое выполнение, в котором не будет принято ни-какого решения, мы должны показать, что каждый процесс может совершить шаг и каждое сообщение может быть получено, не требуя при этом, чтобы было принято какое-нибудь решение. Будем считать, что шаг s соответствует приему и обработке какого-либо сообщения или спонтанному событию (внутреннему или отправлению) в одном из процессов. В зависимости от того, в каком состоянии пребывает тот процесс, который совершает шаг, могут происходить разные события. Прием сообщения допустим только тогда, когда это сообщение пребывает на этапе пересылки, а спонтанный шаг может быть совершен всегда.

Лемма 14.7. *Пусть γ — достижимая бивалентная конфигурация и в этой конфигурации процесс p может совершить шаг s . Тогда существует такая последовательность событий σ , что шаг s может быть применен к конфигурации $\sigma(\gamma)$ и при этом конфигурация $s(\sigma(\gamma))$ будет бивалентной.*

Доказательство. Рассмотрим множество C , состоящее из всех конфигураций, которых можно достичь из конфигурации γ без применения шага s , т. е. $C = \{\sigma(\gamma) : s \text{ не встречается в } \sigma\}$. Заметим, что шаг s применим к любой конфигурации из C , поскольку s является именно шагом, а не каким-то особым событием.

В множестве C найдутся такие конфигурации α_0 и α_1 , что какая-нибудь конфигурация принятия v -решения достижима из конфигурации $s(\alpha_v)$. Чтобы убедиться в этом, заметим, что в силу бивалентности конфигурации γ некоторые конфигурации β_v принятия v -решения достижимы из γ для обоих значений $v = 0, 1$. Если $\beta_v \in C$ (т. е. шаг s не применялся для достижения конфигурации принятия решения), то остается только заметить, что конфигурация $s(\beta_v)$ все еще будет являться конфигурацией принятия v -решения. Поэтому в качестве α_v можно взять конфигурацию β_v . Если же $\beta_v \notin C$ (т. е. шаг s применялся для достижения конфигурации принятия решения), то в качестве α_v можно взять ту конфигурацию, к которой применялся шаг s .

Если $\alpha_0 = \alpha_1$, то $s(\alpha_0)$ — это и есть требуемая бивалентная конфигурация. Допустим, что $\alpha_0 \neq \alpha_1$. Тогда рассмотрим конфигурации, через которые проходят пути, ведущие из γ в α_0 и α_1 . Две конфигурации, лежащие на этих путях, назовем соседними, если одна из них достигается из другой за один шаг. Ввиду того что некоторая конфигурация принятия 0-решения достижима из конфигурации $s(\alpha_0)$ и некоторая конфигурация принятия 1-решения достижима из $s(\alpha_1)$, мы приходим к выводу о том, что

- 1) либо на этих путях лежит такая конфигурация γ' , что $s(\gamma')$ является бивалентной конфигурацией,
- 2) либо существуют такие соседние конфигурации γ_0 и γ_1 , что $s(\gamma_0)$ является конфигурацией принятия 0-решения $s(\gamma_1)$ является конфигурацией принятия 1-решения.

В первом случае $s(\gamma')$ как раз и является требуемой бивалентной конфигурацией. А во втором случае одна из конфигураций γ_0 или γ_1 будет являться разветвлением, и это приводит нас к противоречию. Действительно, допустим, не ограничивая общности, что γ_1 достигается за один шаг из конфигурации γ_0 , т. е. $\gamma_1 = e(\gamma_0)$ для некоторого события e в процессе q . Тогда $s(e(\gamma_0)) = s(\gamma_1)$, и поэтому $s(e(\gamma_0))$ является 1-валентной конфигурацией. Но конфигурация $e(s(\gamma_0))$ не может быть 1-валентной, поскольку $s(\gamma_0)$ уже является 0-валентной конфигурацией. Таким образом, события e и s не являются перестановочными, и отсюда согласно теореме 2.19 следует, что $p = q$. Но в таком случае достижимая конфигурация γ_0 удовлетворяет соотношениям $\gamma_0 \rightsquigarrow_p s(\gamma_0)$ и $\gamma_0 \rightsquigarrow_p s(e(\gamma_0))$. Поскольку конфигурация $s(\gamma_0)$ является 0-валентной, а конфигурация $s(e(\gamma_0))$ является 1-валентной, конфигурация γ_0 будет разветвлением, и это приводит к противоречию. $\square$

Теорема 14.8. *Не существует алгоритмов достижения консенсуса, которые были бы одновременно асинхронными, детерминированными и робастными относительно выхода из строя одного процесса.*

Доказательство. Допустим, что такой алгоритм существует. Тогда построим справедливое выполнение, которое начинается из бивалентной начальной конфигурации γ_0 и не приводит к принятию решения.

Когда в ходе этого построения будет достигнута конфигурация γ_i , выберем в качестве очередного шага s_i такой применимый шаг, вслед за которым выполняется самая длинная цепочка шага. Согласно предыдущей лемме это выполнение можно продолжить таким образом, что будет выполнен шаг s_i и достигнута бивалентная конфигурация γ_{i+1} .

Это построение приводит к бесконечному справедливому выполнению, в ходе которого все процессы остаются исправными, но решение никогда не будет принято. $\square$

14.1.3. Обсуждение

Полученный нами результат гласит, что не существует алгоритмов достижения консенсуса, которые были бы одновременно асинхронными, детерминированными и робастными относительно выхода из строя одного процесса. Тем самым показано, что для некоторых нетривиальных проблем (см. § 13.2.2) нет алгоритмов решения.

К счастью, некоторые допущения, положенные в основу результата Фишера, Линч и Патерсона, можно выделить в явном виде, и оказывается, что этот результат очень чувствителен к малейшему ослаблению любого из этих допущений. Несмотря на полученный результат о невозможности построения некоторых алгоритмов, у многих нетривиальных проблем есть алгоритмические решения, даже в классе асинхронных систем, в которых некоторые процессы могут выходить из строя.

1. *Ослабленная модель отказа.* В § 14.6.2 будет рассмотрена модель неисправностей изначально бездействующих процессов, которая является более сла-

бым типом неисправности, чем модель выхода процесса из строя. В рамках этой модели детерминированные алгоритмы могут решать задачи достижения консенсуса и избрания лидера.

2. *Ослабленная координация.* В § 14.6.3 мы рассмотрим задачи, в которых требуется не столь тесная согласованность действий процессов системы, как это требуется для решения проблемы консенсуса. Как будет показано, для некоторых из этих задач, к числу которых относится задача переименования, существуют решения даже в модели выходящих из строя процессов.

3. *Рандомизация.* В § 14.6.4 будут рассмотрены рандомизованные протоколы, в которых ослабленное условие завершаемости позволяет получать решения даже в случае византийского поведения неисправных процессов.

4. *Ослабленное условие завершаемости.* В § 14.6.5 рассматривается другой вариант ослабленного условия завершаемости; в этом случае требуется, чтобы вычисления обязательно завершали только исправные процессы. Здесь также будет показано, что решение возможно для византийского поведения неисправных процессов.

5. *Синхронность.* В гл. 15 мы оценим, насколько существенно условие синхронности.

Совсем тривиальные решения можно получать, если просто исключить хотя бы одно из трех требований, фигурирующих в определении 14.3 (см. упражнение 14.1). В упражнении 14.2 исследуется вопрос о том, что произойдет, если проигнорировать допущение о том, что вычисления должны проводиться для всех возможных комбинаций входных данных (это допущение явно используется в доказательстве леммы 14.6).

14.2. Изначально бездействующие процессы

В модели изначально бездействующих процессов ни один процесс не выходит из строя, после того как в нем произойдет хотя бы одно событие. Поэтому в справедливом выполнении в каждом процессе происходит либо 0, либо бесконечно много событий.

Определение 14.9. *Справедливым выполнением с t -изначально бездействующими процессами* называется такое выполнение, в котором активны не менее $N - t$ процессов, в каждом активном процессе происходит бесконечно много событий и каждое сообщение, отправленное исправному процессу, доходит до адресата.

В каждом алгоритме консенсуса, который является устойчивым относительно t -изначально бездействующих процессов, каждый исправный процесс принимает решение в любом справедливом выполнении с t -изначально бездействующими процессами. Понятия договоренности и нетривиальности определяются так же, как и в модели вышедших из строя процессов.

Поскольку процессы не выходят из строя после отправления сообщения, любой процесс q может безбоязненно ожидать получения сообщения от процесса p , если q осведомлен о том, что p уже хотя бы один раз отправлял сообще-

ние. Как будет показано, задачи консенсуса и избрания лидера имеют решение в модели изначально бездействующих процессов, если неисправно меньшинство процессов ($t < N/2$). В случае возникновения большего числа неисправностей построить устойчивый алгоритма уже не удастся (см. упражнение 14.3).

Договоренность в подмножестве процессов. Вначале был предложен алгоритм Фишера, Линч и Патерсона [81], в котором каждый исправный процесс вычисляет *одну и ту же* совокупность исправных процессов. Устойчивость этого алгоритма составляет величину $\lfloor (N - 1)/2 \rfloor$. Условимся обозначать буквой L величину $\lceil (N + 1)/2 \rceil$ и будем иметь в виду, что в системе есть не менее L исправных процессов. Вычисления этого алгоритма разбиваются на два этапа (см. алгоритм 14.1).

```

var  $Succ_p, Alive_p, Rcvd_p$  : sets of processes    init  $\emptyset$  ;

begin shout  $\langle name, p \rangle$  ;
  (* то есть: forall  $q \in \mathbb{P}$  do send  $\langle name, p \rangle$  to  $q$  *)
  while  $\#Succ_p < L$ 
    do begin receive  $\langle name, q \rangle$  ;  $Succ_p := Succ_p \cup \{q\}$  end;
  shout  $\langle pre, p, Succ_p \rangle$ ;
   $Alive_p := Succ_p$  ;
  while  $Alive_p \not\subseteq Rcvd_p$ 
    do begin receive  $\langle pre, q, Succ \rangle$  ;
       $Alive_p := Alive_p \cup Succ \cup \{q\}$  ;
       $Rcvd_p := Rcvd_p \cup \{q\}$ 
    end;
end

```

Алгоритм 14.1. Вычисление клубка в графе G

Следует обратить внимание на то обстоятельство, что процессы отправляют сообщения самим себе. Это принято делать во многих робастных алгоритмах, чтобы облегчить их анализ. Здесь и далее запись «shout $\langle mes \rangle$ » будет служить для обозначения операции

forall $q \in \mathbb{P}$ do send $\langle mes \rangle$ to q .

Процессы строят ориентированный граф G , оповещая о своих отличительных признаках (в сообщениях $\langle name, p \rangle$) и ожидая до тех пор, пока они не получат в ответ L сообщений. Так как в системе имеется не менее L исправных процессов, каждый исправный процесс получит достаточно много сообщений, чтобы успешно завершить этот этап алгоритма. Преемниками процесса p в графе G становятся все те узлы q , от которых процесс p получил сообщения $\langle name, q \rangle$.

Изначально бездействующий процесс не отправляет и не получает сообщений, и поэтому в графе G он будет образовывать изолированный узел. У исправного процесса есть L преемников, и, значит, он не будет изолированным. Клубком называется компонента сильной связности, не имеющая исходящих из нее дуг

и содержащая не менее двух узлов. В графе G есть клубок, в состав которого входят исправные процессы. Так как у каждого исправного процесса полустепень исхода равна L , размер клубка будет не меньше L . Коль скоро $2L > N$, в графе может быть только один клубок; обозначим его символом K . Принимая во внимание, что исправный процесс p имеет L преемников, мы приходим к выводу о том, что хотя бы один из его преемников будет входить в состав клубка K . Отсюда следует, что все процессы клубка K являются наследниками процесса p .

Поэтому на втором этапе работы алгоритма процессы занимаются построением индуцированного подграфа графа G , в котором содержатся все их наследники. Для этого каждый исправный процесс оповещает всех о множестве своих преемников. Поскольку процессы никогда не выходят из строя после отправления хотя бы одного сообщения, блокировки на этом этапе не возникает. Действительно, процесс p ожидает сообщения от процесса q только в том случае, если на первом этапе какой-то процесс получил сообщение $\langle name, q \rangle$, свидетельствующее о том, что процесс q исправен.

После завершения работы алгоритма 14.1 каждый исправный процесс получит от каждого из своих последователей имена всех его преемников и сумеет таким образом вычислить тот самый единственный клубок в графе G .

Консенсус и избрание лидера. Как только все исправные процессы придут к договоренности в вопросе о клубке исправных процессов, провести выборы лидера можно будет совсем просто: лидером становится процесс с наибольшим отличительным признаком в клубке K . Столь же просто можно прийти к консенсусу. Каждый процесс отправляет широковещательное сообщение, в котором наряду со списком его преемников указываются и его входные данные (x). После того как будет вычислен клубок K , процессы принимают в качестве решения значение некоторой функции, зависящей от совокупности всех входных данных клубка K (например, ее значением будут те входные данные, которые встречаются наиболее часто, а если таких имеется несколько, то значение полагается равным нулю).

В алгоритме достижения согласия в клубке, а также в алгоритмах консенсуса и избрания лидера происходит обмен $O(N^2)$ сообщениями, и при этом в некоторых сообщениях может содержаться список L имен процессов. Были предложены и более эффективные алгоритмы избрания лидера. Айтаи и др. в работе [112] разработали алгоритм, в котором происходит обмен $O(N(t + \log N))$ сообщениями, и доказали, что эта величина является нижней оценкой сложности задачи. Масузава и др. в работе [139] исследовали эту задачу для клика с восприятием направления, предложили алгоритм, в котором происходит $O(Nt)$ обменов сообщениями, и также установили, что этот алгоритм является оптимальным.

Всякий алгоритм, избирающий в качестве лидера исправный процесс, одновременно с этим дает решение задачи консенсуса, поскольку лидер может оповестить все процессы о своих входных данных и все исправные процессы придут к одному и тому же решению на основе этих данных. Следовательно, упомянутые верхние оценки относятся также и к проблеме консенсуса в случае, когда неисправность ограничивается изначально бездействующими процессами. Если

же речь заходит о процессах, способных выходить из строя по ходу вычисления, то наличие процесса-лидера не помогает нам решить проблему консенсуса: сам лидер может выйти из строя, до того как проведет оповещение всех процессов о своих входных данных. Кроме того, как будет показано в следующем параграфе, проблему избрания лидера нельзя решить в том случае, если процессы могут выходить из строя по ходу вычисления.

14.3. На что способны детерминированные системы?

До сих пор мы рассматривали такую постановку задачи консенсуса, в которой требовалось, чтобы все процессы пришли к одному и тому же решению. В этом параграфе мы покажем, как можно построить алгоритмы для задач, в которых не требуется столь тесного взаимодействия между процессами. В § 14.3.1 будет представлено решение одной практической задачи по переименованию семейства процессов с небольшим разнообразием имен. В § 14.3.2 мы распространим полученные ранее результаты о невозможности построения алгоритмов на более обширный класс проблем принятия решения.

Распределенная *задача* определяется множествами X и D возможных входных и выходных значений и (вообще говоря, частичным) отображением

$$T: X^N \longrightarrow \mathcal{P}(D^N).$$

Это отображение T следует истолковывать так: если входные данные всех процессов представлены вектором $\vec{x} = (x_1, \dots, x_N)$, то $T(\vec{x})$ — это множество всех допустимых выходных результатов алгоритма, которые представлены векторами принятых решений $\vec{d} = (d_1, \dots, d_N)$. Если отображение T является частичной функцией, то это означает, что не всякая комбинация входных данных является приемлемой.

Определение 14.10. Будем говорить, что алгоритм решения задачи. Алгоритм T является робастным относительно выхода из строя t процессов, если он удовлетворяет следующим условиям.

1. **Завершаемость.** В каждом справедливом выполнении с не более чем t вышедшими из строя процессами все исправные процессы принимают решение.
2. **Непротиворечивость.** Если все процессы исправны, то вектор принятых решений $\vec{d}$ принадлежит множеству $T(\vec{x})$.

Условие непротиворечивости означает, что в любом выполнении, в котором лишь некоторые из процессов принимают решения, полученный частичный вектор решений можно всегда доопределить так, чтобы можно было получить вектор из множества $T(\vec{x})$. Для обозначения совокупности всех возможных выходных векторов мы будем использовать запись D_T .

1. *Пример: задача консенсуса.* В задаче консенсуса требуется, чтобы все решения были одинаковыми, т. е.

$$D_{\text{cons}} = \{(0, 0, \dots, 0), (1, 1, \dots, 1)\}.$$

2. *Пример: задача избрания лидера.* В задаче выбора требуется, чтобы в точности один процесс принимал решение 1, а все остальные процессы принимали решение 0, т. е.

$$D_{\text{elec}} = \{(1, 0, \dots, 0), (0, 1, \dots, 0), \dots, (0, 0, \dots, 1)\}.$$

3. *Пример: приближенная договоренность.* В задаче ϵ -приближенной договоренности на вход каждого процесса подается вещественное значение и на выходе его также ожидается вещественное значение. При этом требуется, чтобы максимальное различие между выходными значениями двух процессов не превосходило ϵ , и все выходные значения располагались в отрезке между двумя входными значениями:

$$D_{\text{approx}} = \{(d_1, \dots, d_N) : \max(d_i) - \min(d_i) \leq \epsilon\}.$$

4. *Пример: переименование.* В задаче переименования каждый процесс обладает уникальным отличительным признаком; эти признаки могут выбраны из достаточно большого множества элементов. Каждый процесс выдает в качестве решения новое имя, которое выбирается из меньшей области $1, \dots, K$, и при этом все новые имена должны быть попарно различными, т. е.

$$D_{\text{rename}} = \{(d_1, \dots, d_N) : i \neq j \implies d_i \neq d_j\}.$$

В задаче переименования с сохранением порядка, кроме того, требуется, чтобы новые имена располагались в том же порядке, в котором находились прежние имена, т. е. $x_i < x_j \implies d_i < d_j$.

14.3.1. Разрешимая задача: переименование

В этом параграфе будет представлен алгоритм, разработанный Аттеей и др. в работе [11]. Алгоритм способен выдержать выход из строя $t < N/2$ процессов (здесь t является параметром алгоритма) и провести переименование процессов, используя при этом $K = (N - t/2)(t + 1)$ имен.

Верхняя оценка параметра t . Прежде всего мы покажем, что никакой алгоритм переименования не способен выдержать выход из строя $N/2$ или большего числа процессов. На самом деле почти все робастные алгоритмы теряют работоспособность в случае выхода из строя $t \geq N/2$ процессов, и доказательство приведенной ниже теоремы можно адаптировать к другим задачам.

Теорема 14.11. *Не существует алгоритма переименования, способного выдержать выход из строя $t \geq N/2$ процессов.*

Доказательство. Если $t \geq N/2$, то в распределенной системе можно выделить два непересекающихся множества процессов S и T , в каждом из которых содержится $N - t$ процессов. Поскольку из строя могут одновременно выйти t процессов, каждая из этих групп процессов должна уметь принимать решение «самостоятельно», т. е. без взаимодействия с теми процессами, которые не входят в состав этой группы (см. утверждение 14.4). Но в таком случае эти группы могут принять решения *независимо друг от друга* в одном и том же выполнении.

Ключевой момент доказательства — это демонстрация того, что принятые решения могут быть несовместимы. Мы увидим, каким образом это можно обосновать для задачи переименования.

Согласно утверждению 14.4 для каждой начальной конфигурации γ найдется такая конфигурация δ_S , в которой все процессы из множества S уже приняли решения, и при этом $\gamma \rightsquigarrow_S \delta_S$. Аналогичным свойством обладают и процессы множества T . В результате работы алгоритма в группе, состоящей из $N - t$ процессов, будет определено отношение между векторами, образованными $N - t$ исходными отличительными признаками, и векторами, образованными $N - t$ новыми именами. Так как пространство исходных имен не ограничено, а новые имена берутся из ограниченной области, существуют два таких набора входных данных, которые не имеют общих элементов, но при этом отображаются в частично перекрывающиеся векторы выходных данных. Иначе говоря, существуют такие входные векторы $\vec{u}$ и $\vec{v}$ длины $N - t$, что $u_i \neq v_j$ для любой пары чисел i, j , но при этом соответствующие им выходные векторы $\vec{d}$ и $\vec{e}$ таковы, что хотя бы для одной пары чисел i, j имеет место равенство $d_i = e_j$.

Некорректное выполнение алгоритма переименования будет проходить следующим образом. В начальной конфигурации γ исходные данные процессов из группы S образуют вектор $\vec{u}$, а исходные данные процессов из группы T образуют вектор $\vec{v}$. При этом мы не будем забывать, что исходные имена всех процессов попарно различны (исходные имена процессов вне этих двух групп можно выбирать произвольным образом). Рассмотрим последовательность шагов σ_T , посредством которых группа T переходит от конфигурации γ к такой конфигурации δ_T , где все процессы из множества T сформировали решение (вектор выходных данных) $\vec{e}$. Согласно утверждению 14.1 эта последовательность шагов будет также применима и к конфигурации γ_S , в которой все процессы из группы S уже сформировали решение $\vec{d}$. Тогда в конфигурации $\sigma_T(\gamma_S)$ есть два процесса, которые пришли к одному и тому же решению (ввиду того что $d_i = e_j$), и это свидетельствует о том, что алгоритм не обеспечивает непротиворечивости. $\square$

Далее мы будем полагать, что $t < N/2$.

Алгоритм переименования. В алгоритме переименования (алгоритм 14.2) процесс p ведет в множестве V_p учет тех входных данных других процессов, которые процесс p сумел увидеть. Вначале множество V_p содержит только один элемент x_p . Всякий раз, когда p получает множество входных данных, в котором есть новые для процесса p имена, к множеству V_p добавляются новые входные данные. В начале работы алгоритма, а также всякий раз, когда множество V_p пополняется новыми элементами, процесс p оповещает всех об этом множестве. Ясно, что множество V_p может только расширяться по ходу выполнения, т.е. последовательные значения переменной V_p линейно упорядочены по отношению включения, и к тому же множество V_p может содержать не более N имен. Следовательно, процесс p оповещает об изменении этого множества не более N раз. Поэтому алгоритм завершает свою работу и его сложность по числу обменов сообщениями ограничена сверху величиной $O(N^3)$.

```

var  $V_p$       : set of identities ;
     $c_p$        : integer;

begin  $V_p := \{x_p\}$  ;  $c_p := 0$  ; shout  $\langle \text{set}, V_p \rangle$  ;
  while true
  do begin receive  $\langle \text{set}, V \rangle$ ;
    if  $V = V_p$  then
      begin  $c_p := c_p + 1$  ;
        if  $c_p = N - t$  and  $y_p = b$  then
          (*  $V_p$  впервые стабилизировалось: принять решение *)
           $y_p := (\# V_p, \text{rank}(V_p, x_p))$ 
        end
      else if  $V \subseteq V_p$  then
        skip (* Игнорировать «устаревшую» информацию *)
      else (* новые входные данные; изменить  $V_p$  и возобновить отсчет *)
        begin if  $V_p \subset V$  then  $c_p := 1$  else  $c_p := 0$  ;
           $V_p := V_p \cup V$  ; shout  $\langle \text{set}, V_p \rangle$ 
        end
      end
    end
  end
end

```

Алгоритм 14.2. Простой алгоритм переименования

Кроме того, процесс p подсчитывает (используя для этого переменную c_p), сколько раз он получил копию своего текущего множества V_p . Первоначально значение c_p полагается равным 0, и это значение увеличивается на единицу всякий раз, когда процесс получает сообщение, в котором содержится множество V_p . Может случиться так, что после получения сообщения $\langle \text{set}, V \rangle$ множество V_p расширится; тогда подсчет начинается заново. Если новое значение переменной V_p становится равным V (т.е. V строго включает в себя прежнее значение переменной V_p), то переменная c_p полагается равной 1; в противном случае переменной c_p присваивается значение 0.

Будем говорить, что процесс p достиг *стабильного множества* V , если значение c_p оказалось равным $N - t$, и значение переменной V_p при этом равно V . Иначе говоря, процесс p получил $(N - t)$ раз подряд текущее значение V переменной V_p .

Лемма 14.12. *Стабильные множества линейно упорядочены, т.е. если процесс q достигает стабильного множества V_1 и процесс r достигает стабильного множества V_2 , то $V_1 \subseteq V_2$ или $V_2 \subseteq V_1$.*

Доказательство. Предположим, что процесс q достигает стабильного множества V_1 и процесс r достигает стабильного множества V_2 . Это означает, что процесс q получил сообщение $\langle \text{set}, V_1 \rangle$ от $N - t$ процессов и процесс r получил сообщение $\langle \text{set}, V_2 \rangle$ от $N - t$ процессов. Так как $2(N - t) > N$, существует хотя бы один такой процесс (обозначим его буквой p), от которого процесс q получил сообщение $\langle \text{set}, V_1 \rangle$, а процесс r получил сообщение $\langle \text{set}, V_2 \rangle$. Следовательно,

оба эти множества V_1 и V_2 являлись значениями одной и той же переменной V_p . Поэтому одно из этих множеств должно включать другое. $\square$

Лемма 14.13. *Всякий исправный процесс достигает стабильного множества хотя бы один раз по ходу каждого справедливого выполнения, в котором выходят из строя не более t процессов.*

Доказательство. Рассмотрим исправный процесс p . Множество V_p может только расширяться, и в нем содержится не более N входных имен. Поэтому переменная V_p достигает своего максимального значения V_0 . Процесс p оповещает всех об этом значении, и каждый исправный процесс получает сообщение $\langle \text{set}, V_1 \rangle$. Отсюда следует, что каждый исправный процесс в конечном итоге построит такое множество имен, которое будет включать в себя V_0 . Это включение, однако, не может быть строгим, поскольку иначе мы пришли бы к противоречию: исправный процесс оповестил бы о таком множестве все другие процессы, и процесс p получил бы сообщение, в котором содержится множество, строго включающее в себя максимальное известное для процесса p множество V_0 . Таким образом, в каждом исправном процессе q переменная V_q достигает значения V_0 хотя бы один раз по ходу выполнения, и поэтому в этом выполнении каждый исправный процесс отправляет процессу p сообщение $\langle \text{set}, V_1 \rangle$. Все эти сообщения будут в конечном итоге получены и подсчитаны. Коль скоро значение переменной V_p не может превосходить V_0 , после завершения подсчета множество V_0 станет стабильным множеством процесса p . $\square$

Как только впервые будет достигнуто стабильное множество, процесс p принимает решение (s, r) , где s — это размер множества V , а r — ранг элемента x_p в множестве V . Стабильное множество было получено от $N - t$ процессов, и поэтому в нем содержится не менее $N - t$ входных имен. Это свидетельствует о том, что $N - t \leq s \leq N$. Ранг элемента в множестве размера s удовлетворяет неравенству $1 \leq r \leq s$. Поэтому число всевозможных решений составляет величину $K = \sum_{s=N-t}^N s$, которая равна $(N - t/2)(t + 1)$. При желании для удобства можно задать фиксированное отображение из множества пар натуральных чисел в отрезок $1, \dots, K$ (см. упражнение 14.5).

Теорема 14.14. *Алгоритм 14.2 решает задачу переименования, получая на выходе имена, принадлежащие множеству размера $K = (N - t/2)(t + 1)$.*

Доказательство. По ходу любого справедливого выполнения, в котором выходит из строя не более t процессов, каждый исправный процесс достигает стабильного множества, и поэтому каждый исправный процесс вычисляет какое-то новое имя в качестве решения. Чтобы убедиться в том, что все новые имена попарно различны, рассмотрим стабильные множества V_1 и V_2 , достигнутые соответственно процессами q и r . Если эти множества имеют разные размеры, то решения, которые принимаются процессами q и r , тоже будут разными, поскольку размеры стабильных множеств будут включены в эти решения. Если V_1 и V_2 имеют один и тот же размер, то по лемме 14.12 эти стабильные множества будут одинаковыми. В таком случае в одном и том же стабильном множестве процессы

q и r будут иметь разные ранги, и это вновь приводит нас к выводу о том, что эти процессы примут разные решения. $\square$

Обсуждение. Нужно обратить внимание на то, что процесс не завершает выполнение алгоритма, 14.2 после того как сформирует новое имя; он продолжает выполнять алгоритм, чтобы «помочь» другим процессам принять решение. В работе [11] Аття и др. показали, что это необходимо, поскольку может сложиться такая ситуация, при которой некоторые из процессов работают настолько медленно, что они успевают выполнить свой первый шаг уже после того, как другие процессы приняли свое решение.

Тот простой алгоритм, который был нами рассмотрен, не является наилучшим с точки зрения размера пространства имен, используемых для переименования. Аття и др. описали в работе [11] более сложный алгоритм, который выбирает имена из отрезка $1, \dots, N + t$. Как будет видно из результатов следующего параграфа, нижняя оценка размера пространства новых имен составляет величину $N + 1$.

Аття и др. также предложили алгоритм переименования с сохранением порядка. В этом алгоритме в качестве новых имен выбираются натуральные числа из отрезка от 1 до $K = 2^t \cdot (N - t + 1) - 1$. Было установлено также, что это наименьший размер пространств имен, который позволяет провести переименование, робастное относительно выхода из строя t процессов.

14.3.2. Более общие результаты о невозможности построения алгоритмов

Результат, свидетельствующий о невозможности построения алгоритма консенсуса (теорема 14.8), был распространен Мораном и Вольфшталем в работе [151] на проблемы принятия решения более общего вида. *Графом решений* задачи T называется граф $G_T = (V, E)$, в котором $V = D_T$ и

$$E = \{(\vec{d}_1, \vec{d}_2) : \vec{d}_1 \text{ и } \vec{d}_2 \text{ отличаются значениями только одной компоненты}\}.$$

Задача T называется *связной*, если G_T является связным графом, и *несвязной* в противном случае. Моран и Вольфшталь решили ввести дополнительное предположение о том, что граф входных данных задачи T (который определяется так же, как и граф решений) является связным, т. е. мы можем, как это было сделано в доказательстве леммы 14.6, переходить от одной начальной конфигурации к другой, поочередно изменяя входные данные только в одном процессе. После этого удалось получить новые результаты о невозможности построения нетривиальных алгоритмов, которые помимо свойств завершаемости (1) и непротиворечивости (2) должны также обладать следующим свойством.

3. Нетривиальность. Для любого вектора $\vec{d} \in D_T$ существует достижимая конфигурация, в которой решения, принятые процессами, образуют вектор $\vec{d}$.

Теорема 14.15. *Не существует нетривиальных алгоритмов принятия решений, которые были бы робастными относительно выхода из строя*

хотя бы одного процесса, если граф входных данных задачи T является несвязным.

Доказательство. Допустим, что вопреки утверждению теоремы такой алгоритм A существует. Покажем, что тогда на его основе можно было бы построить алгоритм консенсуса A' , и это противоречило бы теореме 14.8. Для упрощения последующих рассуждений мы будем полагать, что граф G_T содержит две компоненты связности, которые обозначим «0» и «1».

Алгоритм A' вначале симулирует поведение алгоритма A , но вместо того, чтобы принимать в качестве решения значение d , процесс широковежательно рассылает сообщение $\langle \text{vote}, d \rangle$ и ожидает поступления $N - 1$ сообщений, содержащих мнения других процессов. Блокировки здесь не возникает, поскольку все исправные процессы, выполняющие алгоритм A , принимают решения, и не менее $N - 1$ процессов оповестят всех о своем решении.

После получения этих сообщений процесс p будет располагать значениями $N - 1$ компонент вектора решений из множества D^N . К ним можно добавить недостающую компоненту, соответствующую процессу, от которого не было получено сообщений, таким образом, чтобы весь вектор целиком принадлежал бы множеству D_T . (На самом деле эта недостающая компонента представляет собой непротиворечивое решение, которое было или могло быть принято тем самым процессом, мнение которого было не учтено.)

А теперь обратим внимание на то, что разные процессы могут вычислять разные расширения, но все эти расширения будут принадлежать одной и той же компоненте связности графа G_T . Каждый процесс, который узнал мнения $N - 1$ процессов, может выбрать в качестве принятого им решения имя той компоненты связности, которой принадлежит расширенный вектор. Остается показать, что алгоритм A' является алгоритмом консенсуса.

Завершаемость. Как уже было показано ранее, каждый исправный процесс получает не менее $N - 1$ сообщений, содержащих компоненты вектора, вычисленные им самим и другими процессами.

Договоренность. Прежде всего мы покажем, что существует такой вектор $\vec{d}_0 \in D_T$, что каждый исправный процесс получает $N - 1$ компонент вектора $\vec{d}_0$.

Случай 1. Все процессы вычислили решения, выполняя алгоритм A . Пусть $\vec{d}_0$ — вектор достигнутых решений. Тогда каждый процесс получает в сообщениях $N - 1$ компонент вектора $\vec{d}_0$, хотя при этом разные процессы могут «упускать из вида» разные компоненты этого вектора.

Случай 2. Все процессы, кроме одного процесса r , вычислили решения, выполняя алгоритм A . Все исправные процессы получают в сообщениях один и тот же набор из $N - 1$ решений, принятых всеми процессами, за исключением процесса r . Возможно, что процесс r вышел из строя, но с равным успехом можно предполагать, что он просто медленно работает, но тем не менее должен будет в конечном итоге принять решение, т. е. существует такой вектор $\vec{d}_0 \in D_T$, который является расширением набора уже достигнутых решений.

Коль скоро такой вектор $\vec{d}_0$ существует, каждый процесс принимает в качестве решения имя той компоненты связности, которой принадлежит этот вектор $\vec{d}_0$.

Нетривиальность. Учитывая нетривиальность алгоритма A , можно прийти к заключению о том, что достигнуты могут быть как векторы решений, принадлежащие компоненте 0, так и векторы решений, принадлежащие компоненте 1. Из описания алгоритма A' видно, что в таком случае оба типа решений могут быть достигнуты.

Таким образом, мы построили асинхронный детерминированный алгоритм консенсуса A' , который является робастным относительно выхода из строя не более одного процесса. На основании теоремы 14.8 отсюда следует, что указанного алгоритма A не существует. $\square$

Обсуждение. Требование нетривиальности, гласящее, что каждый вектор решений из множества D_T достижим, представляется довольно жестким. Читатель может задаться вопросом: «А существуют ли сколь-нибудь интересные алгоритмы, которые являются тривиальными в указанном выше смысле?» Например, если мы обратимся к алгоритму 14.2, предназначенному для решения задачи переименования, то с первого взгляда не просто заметить, что он является нетривиальным, т. е. *каждый* вектор, состоящий из попарно различных имен, может быть вычислен процессами системы (хотя на самом деле это так). Еще менее ясно, какую роль здесь играет нетривиальность.

Более внимательное изучение доказательства теоремы 14.15 позволяет обнаружить, что в нем можно было воспользоваться ослабленным вариантом условия нетривиальности, а именно требованием, чтобы достижимые векторы решений принадлежали по крайней мере двум различным компонентам связности графа G_T . Ослабленное условие нетривиальности иногда можно вывести из постановки задачи.

Наиболее значительной работой, посвященной разрешимым и неразрешимым задачам принятия решения при наличии одного неисправного процессора, является статья Байрана, Морана и Закса [29]. Они сумели дать полное комбинаторное описание разрешимых задач принятия решения.

14.4. Вероятностные алгоритмы консенсуса

В доказательстве теоремы 14.8 было показано, что каждый асинхронный алгоритм консенсуса допускает бесконечное выполнение, в ходе которого никакого решения не принимается. К счастью, для тщательно выверенных алгоритмов выполнения такого рода возникают достаточно редко; можно сказать, что они встречаются с нулевой вероятностью, и эти алгоритмы являются весьма практичными с вероятностной точки зрения (см. главу 9). В этом разделе мы предложим вниманию читателей два вероятностных алгоритма консенсуса; один из них имеет дело с моделью выхода процесса из строя, а другой — с византийской моделью неисправности. Оба алгоритма были представлены в работе Брачи и Тоуега [35].

В обоих случаях была впервые установлена верхняя оценка стойкости ($t < N/2$ и $t < N/3$ соответственно), и эта оценка достижима для обоих алгоритмов.

В требовании корректности этих вероятностных алгоритмов консенсуса условие завершаемости становится вероятностным, т. е. оно замещается следующим ослабленным условием сходимости.

1. **Сходимость.** Для каждой начальной конфигурации выполняется соотношение

$$\lim_{k \rightarrow \infty} \Pr[\text{исправный процесс не принял решение после выполнения } k \text{ шагов}] = 0.$$

Частичная корректность (условие договоренности) должна иметь место для каждого выполнения. Вероятностные алгоритмы такого рода относятся к классу алгоритмов Лас-Вегаса (см. § 9.1.2).

Вероятность вычисляется по всем выполнениям, начинающимся из заданной начальной конфигурации. Для того чтобы можно было вычислять эти вероятности, необходимо задать распределение вероятностей на множестве всех выполнений. Это можно сделать, позволив процессам использовать случайные величины (как это было предложено в гл. 9), но мы здесь поступим иначе и определим распределение вероятностей на множестве событий получения сообщений.

Распределение вероятностей на множестве выполнений, начинающихся из заданной конфигурации, определяется на основании допущения о *справедливом расписании*. Работа обоих алгоритмов разбивается на этапы (туры); на протяжении одного тура процесс широкоэвентально рассылает какое-то сообщение и ожидает получения $N - t$ сообщений. Мы условимся использовать запись $R(q, p, k)$ для обозначения события получения процессом p в k -м туре сообщения от процесса q среди первых $N - t$ сообщений, относящихся к этому туру. Справедливое составление расписания означает, что

- 1) $\exists \epsilon > 0 \forall p, q, k: \Pr[R(p, q, k)] \geq \epsilon$;
- 2) каков бы ни был тур k и каковы бы ни были три разных процесса p, q, r , события $R(q, p, k)$ и $R(q, r, k)$ являются независимыми.

Заметим, что утверждение 14.4 будет также выполняться и для вероятностных алгоритмов, которые удовлетворяют условию сходимости (завершаемости выполнения с вероятностью 1). Действительно, поскольку всякая достижимая конфигурация достигается с положительной вероятностью, конфигурация принятия решения должна быть достижима из каждой достижимой конфигурации (хотя это может происходить и не в каждом выполнении).

14.4.1. Протоколы консенсуса, робастные относительно выхода процессов из строя

В этом параграфе изучается задача консенсуса в модели неисправностей, допускающей выход процессов из строя. Вначале мы установим верхнюю оценку $t < N/2$ стойкости алгоритмов, а затем опишем алгоритм, имеющий стойкость $t < N/2$.

Теорема 14.16. *Не существует протоколов консенсуса, которые были бы робастными относительно выхода из строя t процессов, где $t \geq N/2$.*

Доказательство. Если такой протокол P существует, то отсюда следует, что верны следующие три утверждения.

Утверждение 14.17. *Протокол P имеет хотя бы одну бивалентную начальную конфигурацию.*

Доказательство. Обоснование этого утверждения аналогично доказательству леммы 14.6. Читателям предлагается провести его самостоятельно. $\square$

Для заданного подмножества процессов S конфигурацию γ будем называть S -бивалентной, если из конфигурации γ , выполняя только действия процессов семейства S , можно достичь как конфигурации принятия 0-решения, так и конфигурации принятия 1-решения. Конфигурация γ называется S -0-валентной, если, выполняя только действия процессов семейства S , можно достичь конфигурации принятия 0-решения, но нельзя достичь конфигурации принятия 1-решения. Аналогичным образом определяется S -1-валентная конфигурация.

Разделим все процессы на две группы S и T размера $\lfloor N/2 \rfloor$ и $\lceil N/2 \rceil$ соответственно.

Утверждение 14.18. *Какова бы ни была достижимая конфигурация γ , она либо одновременно является S -0-валентной и T -0-валентной, либо одновременно S -1-валентной и T -1-валентной.*

Доказательство. Действительно, высокая стойкость нашего протокола приводит к тому, что оба семейства процессов S и T могут принимать решение независимо друг от друга. Если бы эти решения были *разными*, то взяв комбинацию используемых расписаний, можно было бы достичь противоречивой конфигурации. $\square$

Утверждение 14.19. *В протоколе P нельзя достичь бивалентной конфигурации.*

Доказательство. Рассмотрим произвольную бивалентную конфигурацию γ . Предположим, не ограничивая общности, что γ является S -1-валентной и T -1-валентной конфигурацией (согласно утверждению 14.18). Однако, коль скоро конфигурация γ является бивалентной, конфигурация δ_0 принятия 0-решения может быть достигнута из конфигурации γ (разумеется, при взаимодействии процессов, относящихся к разным группам). В последовательности конфигураций выполнения, состоящей из всех конфигураций, следующих за γ и предшествующих δ_0 , можно выделить две такие последовательно идущие конфигурации γ_1 и γ_0 , что $\gamma_v, v \in \{0, 1\}$, является одновременно S - v -валентной и T - v -валентной конфигурацией. Рассмотрим тот процесс p , переход в котором преобразует конфигурацию γ_1 в конфигурацию γ_0 . Тогда этот процесс p не может принадлежать группе S , поскольку конфигурация γ_1 является S -1-валентной, а конфигурация

γ_0 является S -0-валентной. Подобным же образом можно убедиться в том, что $p \notin T$. В результате мы приходим к противоречию. $\square$

Утверждения 14.17 и 14.19 противоречат друг другу, и это свидетельствует о том, что такого протокола P не существует. Таким образом, теорема 14.16 доказана. $\square$

Робастный относительно выхода процессов из строя алгоритм консенсуса Брачи—Туэга. Робастный относительно выхода процессов из строя алгоритм консенсуса, предложенный в работе Брачи и Туэга [35], выполняет операции, которые разделены на этапы, или *туры*. В k -м туре процесс отправляет сообщение всем процессам (и в том числе самому себе) и ожидает поступления $N - t$ сообщений, относящихся к k -му туру. Ожидание такого числа сообщений не является предпосылкой для блокировки (см. упражнение 14.10).

В каждом туре процесс p провозглашает свой выбор 0 или 1 вместе с определенным весовым коэффициентом. Этот весовой коэффициент равен количеству тех сообщений, которые были получены процессом в предшествующем туре и в которых предпочтение отдавалось именно этому выбранному значению. В первом туре весовой коэффициент полагается равным 1. Если весовой коэффициент превосходит $N/2$, то соответствующий сделанный выбор называется *свидетельством*. Как мы увидим далее, в одном и том же туре разные процессы могут голосовать по-разному, но они никогда не будут свидетельствовать в пользу различных значений. Если в k -м туре процесс p получает свидетельство в пользу какого-то значения k , то он голосует за это значение в следующем $(k + 1)$ -ом туре; В противном случае процесс p голосует за то значение, за которое было подано большинство голосов в сообщениях, полученных процессом p . Решение принимается в том случае, если в некотором туре было получено более t свидетельств. Процесс принявший решение, прекращает выполнение основного цикла и оповещает всех о своем свидетельстве на протяжении следующих двух туров, чтобы побудить другие процессы принять решение. Описание этого протокола представлено в алгоритме 14.3.

Сообщения с поданными голосами, относящимися к более поздним турам, должны быть отложены и обработаны в соответствующих турах. В алгоритме это моделируется следующим образом: процесс отправляет соответствующее сообщение самому себе и приступает к его обработке в нужное время. Следует отметить, что в каждом туре процесс получает самое большее один голос от всякого другого процесса. Процесс принимает во внимание только $N - t$ голосов, и коль скоро подать свои голоса могут более $N - t$ процессов, разные процессы могут принимать во внимание разные подмножества поданных голосов. Мы установим ряд полезных свойств, которыми обладает рассматриваемый алгоритм; из этих свойств будет следовать, что данный алгоритм является робастным относительно выхода процессов из строя протоколом консенсуса (теорема 14.24).

Лемма 14.20. *Ни в каком туре не найдется двух процессов, свидетельствующих в пользу разных значений.*

```

var valuep      : (0, 1)   init xp   (* за что голосует p *)
roundp         : integer  init 0     (* номер тура *)
weightp        : integer  init 1     (* вес голоса, который подает p *)
msgsp[0..1]     : integer  init 0     (* счетчик полученных сообщений *)
witnessp[0..1]  : integer  init 0     (* счетчик полученных свидетельств *)
begin
  while yp = b do
    begin witnessp[0], witnessp[1], msgsp[0], msgsp[1] :=
      0, 0, 0, 0 ; (* Сбросить счетчики *)
    shout ⟨vote, roundp, valuep, weightp⟩ ;
    while msgsp[0] + msgsp[1] < N - t do
      begin receive ⟨vote, r, v, w⟩ ;
        if r > roundp then (* Следующие туры ... *)
          send ⟨vote, r, v, w⟩ to p (* ... обработать позже *)
        else if r = roundp then
          begin msgsp[v] := msgsp[v] + 1 ;
            if w > N/2 then (* Свидетельство *)
              witnessp[v] := witnessp[v] + 1
            end
          end (* r < roundp, проигнорировать *) skip
        end; (* Выбрать новое значение: голос и вес в следующем туре *)
      if witnessp[0] > 0 then valuep := 0
      else if witnessp[1] > 0 then valuep := 1
      else if msgsp[0] > msgsp[1] then valuep := 0 else valuep := 1 ;
      weightp := msgsp[valuep] ; (* Принять решение, если ≥ t свидетельств *)
      if witnessp[valuep] > t then yp := valuep ;
      roundp := roundp + 1
    end; (* Помочь другим процессам принять решение *)
    shout ⟨vote, roundp, valuep, N - t⟩ ; shout ⟨vote, roundp + 1, valuep, N - t⟩
  end

```

Алгоритм 14.3. Алгоритм консенсуса Брачи—Туэга

Доказательство. Предположим, что в k -м туре процесс p свидетельствует в пользу значения v , а процесс q свидетельствует в пользу значения w . Так как в первом туре ни один процесс не может свидетельствовать, имеет место неравенство $k > 1$. Из сделанного предположения следует, что в $(k - 1)$ -м туре процесс p получил более $N/2$ голосов, отданных в пользу значения v , и процесс q получил более $N/2$ голосов, отданных в пользу значения w . Таким образом, суммарно процессы p и q приняли во внимание более N голосов. Значит, два множества процессов, голоса которых были приняты во внимание процессами p и q , имеют непустое пересечение, т.е. существует такой процесс r , который отправил голос в пользу решения v процессу p и голос в пользу решения w процессу q . Ввиду того, что в каждом туре процесс не может отдавать голос в пользу двух разных решений, мы приходим к выводу о том, что $v = w$. $\square$

Лемма 14.21. *Если какой-то процесс принимает решение, то все исправные процессы приходят к такому же решению не позднее, чем через два тура.*

Доказательство. Пусть k — это первый тур, в котором было принято какое-то решение. Предположим, что в этом туре решение v принял процесс p . Согласно условию принятия решения это означает, что в k -м туре имелись v -свидетельства. Из леммы 14.20 следует, что свидетельств в пользу других решений нет, и поэтому никакого другого решения в k -м туре принято быть не может.

В k -м туре имелось более t свидетельств в пользу решения v (это следует из условия принятия решения процессом p), и поэтому в этом туре все исправные процессы получили хотя бы одно v -свидетельство. Следовательно, все процессы, которые принимают участие в голосовании в $(k+1)$ -м туре, отдадут свои голоса в пользу решения v (нужно иметь в виду, что процесс p в $(k+1)$ -м туре будет по-прежнему оповещать всех о том, что он голосует за свое решение). Это означает, что если в $(k+1)$ -м туре и принимаются какие-то решения, то эти решения выбирают значение v .

В $(k+1)$ -м туре подаются только голоса, отдающие предпочтение значению v , и поэтому все процессы, которые голосуют в $(k+2)$ -м туре, свидетельствуют в этом туре в пользу значения v (к их числу относится и процесс p). Таким образом, в $(k+2)$ -м туре все исправные процессы, которые еще не приняли решения в предыдущих турах, получают $N - t$ свидетельств в пользу значения v и примут в качестве решения значение v . $\square$

Лемма 14.22.

$$\lim_{k \rightarrow \infty} \Pr[\text{Во всех турах } 0, 1, \dots, k \text{ не было принято ни одного решения}] = 0.$$

Доказательство. Обозначим символом S множество, состоящее из $N - t$ исправных процессов (такое множество существует), и предположим, что ни одного решения не было принято вплоть до k_0 -го тура. Как следует из допущения о справедливом расписании, существует такое число $\rho > 0$, что в любом туре вероятность того, что каждый процесс из множества S получит $N - t$ сообщений с голосами всех процессов из множества S , не меньше этой величины ρ . С вероятностью, не меньшей $\psi = \rho^3$, это может случиться в трех турах с номерами k_0 , $k_0 + 1$ и $k_0 + 2$.

Если это случится, то все процессы из множества S получают один и тот же набор голосов в туре k_0 и, следовательно, все процессы из S выберут в k_0 -м туре одно и то же значение, которое мы обозначим v_0 . Тогда в $(k_0 + 1)$ -м туре все процессы из множества S проголосуют за одно и то же значение v_0 , и поэтому в $(k_0 + 1)$ -м туре каждый процесс из S получит $N - t$ сообщений с голосами в пользу значения v_0 . Вследствие этого в туре $(k_0 + 2)$ -м туре все процессы из множества S будут свидетельствовать в пользу значения v_0 . Значит, в $(k_0 + 2)$ -м туре каждый процесс из S получит $N - t > t$ свидетельств в пользу значения v_0 , и это приведет к тому, что все процессы из множества S примут значение v_0 в качестве решения.

Таким образом, должно выполняться неравенство

$$\Pr[\text{Процессы из } S \text{ не приняли решения в } (k+2)\text{-ом туре}] \leq (1 - \psi) \times \\ \times \Pr[\text{Процессы из } S \text{ не приняли решения в турах с номерами } 0, 1, \dots, k-1],$$

из которого следует утверждение леммы. $\square$

Лемма 14.23. *Если все процессы начнут выполнять свой алгоритм с входным значением v , то все процессы примут решение v во втором туре.*

Доказательство. В первом туре все процессы получают сообщения, в которых содержатся только голоса в пользу значения v , и поэтому все процессы во втором туре будут свидетельствовать в пользу этого значения. Отсюда следует, что все процессы в этом же туре примут решение v . $\square$

Теорема 14.24. *Если $t < N/2$, то алгоритм 14.3 является вероятностным протоколом консенсуса, робастным относительно выхода из строя t процессов.*

Доказательство. Условие сходимости алгоритма было доказано в лемме 14.22, а условие договоренности — в лемме 14.21. Нетривиальность протокола следует из леммы 14.23. $\square$

Зависимость решения от входных данных более подробно исследуется в упражнении 14.11.

14.4.2. Робастные в византийской модели неисправностей протоколы консенсуса

Византийская модель неисправностей доставляет больше неприятностей, чем модель выходящих из строя процессов, поскольку византийские процессы могут выполнять произвольные переходы и отправлять сообщения, которые никак не обусловлены устройством алгоритма. Далее мы будем использовать запись $\gamma \rightsquigarrow \delta$ (или $\gamma \rightsquigarrow_S \delta$) для обозначения последовательности *корректных шагов*, т. е. переходов протокола (соответственно происходящих только в процессах множества S), в результате которых рассматриваемая система переводится из конфигурации γ в конфигурацию δ . Конфигурация γ считается достижимой, если существует последовательность корректных шагов, при выполнении которых одна из начальных конфигураций переходит в конфигурацию γ . Способность к злонамеренным действиям, присущая византийской модели неисправностей, приводит к понижению порога устойчивости системы по сравнению с моделью выхода процессов из строя.

Теорема 14.25. *Не существует робастного протокола консенсуса в византийской модели с t неисправными процессами в случае, когда $t \geq N/3$.*

Доказательство. Предположим, что такой протокол существует вопреки утверждению теоремы. Как и прежде, читателю предоставляется возможность самостоятельно убедиться в том, что у любого такого протокола есть бива-

лентная начальная конфигурация (как обычно, для этого нужно воспользоваться условием нетривиальности).

Высокая степень устойчивости этого протокола позволяет выбрать два таких множества процессов S и T , что $|S| \geq N - t$, $|T| \geq N - t$ и $|S \cap T| \leq t$. Иными словами, оба множества процессов S и T достаточно велики, чтобы все исправные процессы могли быть сосредоточены в любом из этих множеств, но пересечение этих множеств может целиком состоять из предательских процессов. Воспользуемся этими множествами, для того чтобы показать, что ни одна бивалентная конфигурация недостижима в этом протоколе.

Утверждение 14.26. *Всякая достижимая конфигурация γ либо является одновременно S -0-валентной и T -0-валентной, либо является одновременно S -1-валентной и T -1-валентной.*

Доказательство. Так как конфигурация γ достигается путем выполнения последовательности корректных шагов, у нас остается полная свобода выбора t неисправных процессов. Допустим противное: множества процессов S и T могут прийти к *разным* решениям, т.е. $\gamma \rightsquigarrow_S \delta_v$ и $\gamma \rightsquigarrow_T \delta_{\bar{v}}$, где δ_v ($\delta_{\bar{v}}$) — это конфигурация, в которой все процессы множества S (множества T) выбрали в качестве решения значение v (противоположное значение $\bar{v}$). Но тогда может быть достигнуто противоречивое состояние, если предположить, что процессы из множества $S \cap T$ ведут себя злонамеренно, и составить следующий распорядок действий. Начиная с конфигурации γ процессы из множества $S \cap T$ взаимодействуют с остальными процессами из множества S так, как того требует последовательность действий, приводящая к формированию v -решения процессами из множества S . Как только все процессы из S примут это решение, предательские процессы из множества $S \cap T$ восстанавливают свои локальные состояния в том виде, в каком они были в конфигурации γ , и начинают взаимодействовать с остальными процессами из множества T так, как того требует последовательность действий, приводящая к формированию $\bar{v}$ -решения процессами из множества T . В результате будет достигнута конфигурация, в которой исправные процессы принимают разные решения, что противоречит условию договоренности. $\square$

Утверждение 14.27. *Бивалентные конфигурации недостижимы.*

Доказательство. Допустим, что есть достижимая бивалентная конфигурация γ . Не ограничивая общности, будем считать, что γ является одновременно S -1-валентной и T -1-валентной (утверждение 14.26). Поскольку γ является бивалентной конфигурацией, то из нее достижима какая-то конфигурация принятия 0-решения δ_0 (ясно, что это происходит при взаимодействии процессов из множеств S и T). В соответствующем выполнении между конфигурациями γ и δ_0 найдутся две такие идущие друг за другом конфигурации γ_1 и γ_0 , что γ_v является одновременно S - v -валентной и T - v -валентной. Рассмотрим процесс p , действие которого привело к переходу из γ_1 в γ_0 . Процесс p не может принадлежать множеству S , т. к. конфигурация γ_1 является S -1-валентной, а конфигура-

ция γ_0 является S -0-валентной. По той же самой причине невозможно включение $p \in T$. Таким образом, мы приходим к противоречию. $\square$

Последнее из доказанных утверждений противоречит предположению о существовании бивалентных начальных конфигураций. Этим завершается доказательство теоремы 14.25. $\square$

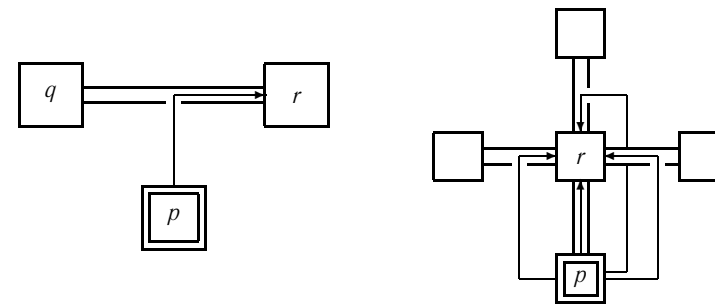


Рис. 14.4. Византийский процесс, симулирующий другие процессы

Алгоритм консенсуса Брачи и Туэга, робастный в византийской модели неисправностей. Если $t < N/3$, то протокол консенсуса, робастный относительно t византийских процессов, действительно существует. Для этого необходимо, чтобы коммуникационная система позволяла процессам-адресатам при получении сообщений определять, какими процессами эти сообщения были отправлены. Если византийский процесс p может отправить исправному процессу r некоторое сообщение так, чтобы процесс r при этом считал, что это сообщение *получено от* q (см. рис.14.4), то проблема становится неразрешимой. Действительно, процесс p сможет в этом случае моделировать достаточно много исправных процессов, для того чтобы вынудить процесс r принять неверное решение.

Точно так же, как и в случае с алгоритмом, робастным относительно выхода процессов из строя, работа рассматриваемого нами алгоритма 14.5, робастного относительно неисправностей византийского типа, разделена на этапы (туры). В каждом туре любой процесс может принять участие в голосовании, и решение будет принято, как только достаточно много голосов будет подано за одно и то же значение. Нижняя граница устойчивости алгоритма ($t < N/3$) позволяет избежать необходимости проводить различия между процессами, выступающими в роли свидетелей, и процессами, не являющимися свидетелями: процесс принимает решение, как только получит более $(N + t)/2$ бюллетеней, поданных за одно и то же значение.

Способность византийских процессов совершать злонамеренные действия требует, однако, введения в протокол механизма верификации голосования, который является ключевым элементом алгоритма 14.5. Без этого механизма византийский процесс может распространять бюллетени голосования среди исправных

```

var  $round_p, msgs_p[0..1], echos_p[\mathbb{P}, 0..1] : \text{integer}$  init 0 ;
     $value_p : (0, 1)$  init  $x_p$  ;
while true do
    begin forall  $v, q$  do begin  $msgs_p[v] := 0 ; echos_p[q, v] := 0$  end;
    shout  $\langle \text{vote}, in, p, value_p, round_p \rangle$  ;
    (* Теперь принять  $N - t$  голосов для текущего тура *)
    while  $msgs_p[0] + msgs_p[1] < N - t$  do
        begin receive  $\langle \text{vote}, t, r, v, rn \rangle$  from  $q$  ;
        if  $\langle \text{vote}, t, r, *, rn \rangle$  уже было получено от  $q$ 
            then skip (*  $q$  повторяется, должно быть, это «византиец» *)
        else if  $t = in$  and  $q \neq r$ 
            then skip (*  $q$  лжет, должно быть, это «византиец» *)
        else if  $rn > round_p$ 
            then (* обработать сообщение в более поздних турах *)
                send  $\langle \text{vote}, t, r, v, rn \rangle$  to  $p$ 
        else (* обработать или ретранслировать сообщение с бюллетенем голосования *)
            case  $t$  of
                in : shout  $\langle \text{vote}, ec, r, v, rn \rangle$ 
                ec : if  $rn = round_p$  then
                    begin  $echos_p[r, v] := echos_p[r, v] + 1$  ;
                        if  $echos_p[r, v] = \lfloor (N + t)/2 \rfloor + 1$ 
                            then  $msgs_p[v] := msgs_p[v] + 1$ 
                        end
                    else skip (* Старое сообщение *)
                esac
            end; (* выбрать значение для следующего тура *)
        if  $msgs_p[0] > msgs_p[1]$  then  $value_p := 0$  else  $value_p := 1$  ;
        if  $msgs_p[value_p] > (N + t)/2$  then  $y_p := value_p$  ;
         $round_p := round_p + 1$ 
    end

```

Алгоритм 14.5. Алгоритм консенсуса, робастный в византийской модели неисправностей

процессов, отправляя разным исправным процессам бюллетени с *разными* значениями. В модели неисправности процессов, выходящих из строя, подобное злоумышленное поведение невозможно. Хотя византийский процесс r может отправить бюллетени голосования с разными значениями исправным процессам p и q , верификация устроена так, что он не может ввести в заблуждение процессы p и q в некоторых турах и вынудить их *учесть при подведении итогов голосования* разные значения, полученные от r .

В основу механизма верификации положена ретрансляция сообщений. Процесс оглашает то значение, за которое он голосует, как *исходное*, сопровождая сообщение пометкой **in**. Каждый процесс после получения бюллетеня голосования с пометкой **in** ретранслирует полученное сообщение, сопровождая его пометкой **out**. При подведении итогов голосования процесс принимает в качестве решения то значение, за которое были отданы голоса более чем в $(N + t)/2$

ретранслированных сообщениях. Механизм верификации поддерживает (частичную) корректность коммуникации между исправными процессами (лемма 14.28), и исправные процессы никогда не учитывают бюллетени, полученные от тех процессов, которые проголосовали за разные значения (лемма 14.29). Блокировка также не возникает (лемма 14.30).

Мы будем говорить, что процесс p *учитывает голос в пользу значения* v , за которое проголосовал процесс r в туре k , если p увеличивает значение $msgs_p[v]$ после получения бюллетеня голосования в сообщении $\langle \text{vote}, ec, r, v, k \rangle$. Алгоритм позволяет процессу p пройти k -й тур только после того, как были учтены $N - t$ голосов, и при этом в каждом туре процесс p учитывает не более одного голоса от каждого процесса.

Лемма 14.28. Если исправный процесс p учитывает в k -м туре голос исправного процесса r в пользу значения v , то процесс p действительно проголосовал в k -м туре за выбор в качестве решения значения v .

Доказательство. Процесс p учитывает бюллетень голосования после получения сообщения $\langle \text{vote}, ec, r, v, k \rangle$ от более чем $(N + t)/2$ (различных) процессов. Хотя бы один исправный процесс s должен был отправить такое сообщение процессу p . Процесс s отправляет ретранслированное сообщение процессу p только после того, как им было получено сообщение $\langle \text{vote}, t, r, v, k \rangle$ от процесса r . Поскольку процесс r исправен, отсюда следует, что в k -м туре он отдал свой голос в пользу значения v . $\square$

Лемма 14.29. Если исправные процессы p и q учитывают в k -м туре голос процесса r , то они относят это голос в пользу одного и того же значения.

Доказательство. Предположим, что в k -ом туре процесс p учитывает голос процесса r , отданный в пользу значения v , а процесс q учитывает голос процесса r , отданный в пользу значения w . Таким образом, процесс p получил сообщение $\langle \text{vote}, ec, r, v, k \rangle$ от более чем $(N + t)/2$ процессов, и процесс q получил сообщение $\langle \text{vote}, ec, r, w, k \rangle$ от более чем $(N + t)/2$ процессов. Так как у нас есть всего лишь N процессов, более чем t процессов отправили сообщение $\langle \text{vote}, ec, r, v, k \rangle$ процессу p и вместе с этим отправили сообщение $\langle \text{vote}, ec, r, w, k \rangle$ процессу q . Значит, это совершил хотя бы один исправный процесс, и поэтому $v = w$. $\square$

Лемма 14.30. Если все исправные процессы участвуют в k -м туре, то все исправные процессы учитывают достаточно много голосов, чтобы завершить этот тур.

Доказательство. Исправный процесс r , начиная k -й тур со значением $r = v$, оглашает исходное значение, за которое он отдает свой голос в этом раунде, и это значение ретранслируется всеми исправными процессами. Таким образом, если процессы p и r исправны, то сообщение $\langle \text{vote}, ec, r, v, k \rangle$ поступит процессу p от не менее чем $N - t$ процессов, и это позволит процессу p в k -м

туре учесть голос процесса r , отданный в пользу значения v , если только процесс p не получит в этом раунде еще раньше $N - t$ других голосов. $\square$

Доказательство корректности рассматриваемого протокола проводится по той же самой схеме, что и доказательство корректности исследованного ранее протокола, робастного относительно выхода процессов из строя.

Лемма 14.31. *Если исправный процесс принимает решение v в k -м туре, то все исправные процессы выбирают значение v в k -м и во всех последующих турах.*

Доказательство. Рассмотрим множество S , состоящее не менее чем из $(N + t)/2$ процессов, голоса которых были отданы за значение v и учтены процессом p в k -м туре. Исправный процесс q в k -м туре учитывает $N - t$ голосов, и среди них не менее $|S| - t > (N - t)/2$ голосов, которые были поданы процессами из множества S . По лемме 14.29 процесс q учитывает более чем $(N - t)/2$ голосов, поданных за значение v , и отсюда следует, что процесс q в k -м туре выбирает значение v .

Чтобы убедиться в том, что все исправные процессы выбирают значение v и в последующих турах, предположим, что все исправные процессы выбирают значение v в некотором туре l . Значит, все исправные процессы в $(l + 1)$ -м раунде отдают свои голоса за значение v . В $(l + 1)$ -м раунде каждый исправный процесс учитывает $N - t$ бюллетеней голосования, к числу которых относятся более чем $(N - t)/2$ голосов, поданных исправными процессами. По лемме 14.28 всякий исправный процесс учитывает не менее $(N - t)/2$ голосов, поданных за значение v , и поэтому в следующем $(l + 1)$ -м раунде он снова выберет значение v . $\square$

Лемма 14.32. *Справедливо равенство*

$$\lim_{k \rightarrow \infty} \Pr[\text{Исправный процесс } p \text{ не принял решения в турах } 0, 1, \dots, k - 1] = 0.$$

Доказательство. Рассмотрим множество S , состоящее не менее чем из $N - t$ исправных процессов, и предположим, что процесс p еще не принял решения перед началом k -го тура. С некоторой вероятностью $\psi > 0$ в k -м туре все процессы из множества S учитывают бюллетени голосования *одного и того же семейства* $N - t$ процессов, а затем в $(k + 1)$ -м туре учитывают только бюллетени голосования процессов из множества S . Если это происходит, то все процессы из множества S голосуют одинаково в $(k + 1)$ -м туре и в том же туре принимают решение. Таким образом,

$$\Pr[\text{Исправный процесс } p \text{ не принял решения в турах } 0, 1, \dots, k + 1] \leq (1 - \psi) \times \Pr[\text{Исправный процесс } p \text{ не принял решения в турах } 0, 1, \dots, k - 1],$$

и отсюда следует утверждение леммы. $\square$

Лемма 14.33. *Если все исправные процессы начинают выполнять алгоритм с входными данными v , то в конечном итоге в качестве решения будет принято значение v .*

Доказательство. Так же как и в доказательстве леммы 14.31, можно показать, что все исправные процессы будут выбирать значение v в каждом туре. $\square$

Теорема 14.34. *Алгоритм 14.5 — это вероятностный протокол консенсуса, который является робастным относительно t византийских процессов при $t < N/3$.*

Доказательство. Сходимость предложенного протокола следует из леммы 14.32. Выполнимость условия договоренности обеспечивает лемма 14.31. Нетривиальность протокола следует из леммы 14.33. $\square$

Зависимость решения от входных значений рассматривается в упражнении 14.12. Чтобы сделать изложение более ясным, мы представили алгоритм 14.5 в виде бесконечного цикла. Теперь расскажем о том, какие изменения нужно внести, чтобы каждый процесс, принявший решение, мог завершить вычисление. После того как в k -м туре процесс p принял решение, он покидает цикл и оглашает «многоразовые» бюллетень голосования $\langle \text{vote}, \text{in}, p, v, k^+ \rangle$ и ретрансляцию $\langle \text{vote}, \text{ec}, *, v, k^+ \rangle$. Эти сообщения истолковываются как исходный и ретранслированный бюллетени голосования, которые будут действительными во всех последующих турах начиная с k -го тура. И в самом деле, процесс p будет голосовать за значение v во всех последующих турах, и так будут поступать все исправные процессы (лемма 14.31). Таким образом, «многоразовые» сообщения — это как раз те сообщения, которые были бы отправлены процессом p , если бы он продолжил выполнять алгоритм. Возможное исключение составляют ретранслированные сообщения бюллетеней голосования, полученных от злонамеренных процессов.

14.5. Слабая завершаемость

В этом параграфе изучается задача широковещательного распространения информации в асинхронных системах, подверженных неисправностям византийского типа. Цель широковещательной рассылки состоит в том, чтобы ознакомить все процессы системы с некоторым значением, которым вначале располагает один из процессов системы (этот процесс в дальнейшем мы будем называть *генералом*). Если говорить более формально, то требование нетривиальности для протокола консенсуса усиливается новым условием, гласящим, что значение, принятое в качестве решения, должно быть входным значением генерала в том случае, когда генерал является исправным процессом:

3. Зависимость. Если генерал исправен, то все исправные процессы принимают в качестве решения значение входных данных генерала.

Однако в таком случае генерал становится единственным узлом, подверженным неисправностям, и поэтому проблема является неразрешимой, о чем свидетельствует следующая теорема.

Теорема 14.35. *Не существует алгоритма, робастного относительно одной неисправности византийского типа, который удовлетворял бы*

словиям сходимости, договоренности и зависимости даже в том случае, когда сходиться должны лишь те выполнения, в которых генерал отправляет хотя бы одно сообщение.

Доказательство. Мы рассмотрим два сценария. В первом из них предполагается, что генерал является византийским процессом. Это сценарий служит нам для того, чтобы определить достижимую конфигурацию γ . Как будет показано во втором сценарии, это приводит к противоречию.

1. Предположим, что генерал является византийским процессом и отправляет сообщение процессу p_0 , призывая его начать широковещательное распространение значения 0, и процессу p_1 , призывая его начать широковещательное распространение значения 1. После этого генерал прекращает работу. Получившуюся в результате этого конфигурацию обозначим символом γ .

Согласно условию сходимости конфигурация принятия решения должна быть достигнута даже в том случае, если генерал выходит из строя. Пусть $S = \mathbb{P} \setminus \{g\}$; тогда, не ограничивая общности, мы будем считать, что $\gamma \rightsquigarrow_S \delta_0$, где δ_0 — это конфигурация принятия 0-решения.

2. Во втором сценарии мы будем полагать, что генерал является исправным процессом, имеющим на входе значение 1, и он отправляет сообщения процессам p_0 и p_1 , призывая их начать широковещательную рассылку значения 1. После этого дальнейшие сообщения, отправленные генералом, задерживаются на очень долгое время. Допустим теперь, что p_0 — это византийский процесс и сразу после получения сообщения от генерала он переходит в то состояние, которое предписано ему в конфигурации γ , т.е. делает вид, что он получил от генерала приказ начать распространение значения 0. Так как $\gamma \rightsquigarrow_S \delta_0$, решение 0 будет достигнуто без всякого взаимодействия с генералом. Но такое решение недопустимо, поскольку генерал исправен и имеет на входе значение 1. $\square$

Невозможность построения алгоритма консенсуса обусловлена тем, что генерал может начать широковещательную рассылку и отключиться (первый сценарий), не предоставив достаточной информации о своих входных данных (и это обстоятельство используется во втором сценарии). Мы попробуем теперь убедиться в том, что детерминированное решение можно получить, если потребовать завершения вычислений лишь в том случае, когда генералом является исправный процесс.

Определение 14.36. Алгоритмом широковещательной рассылки, робастным относительно t неисправностей византийского типа, называется всякий алгоритм, удовлетворяющий следующим трем требованиям.

1. **Слабая завершаемость.** Решение принимают либо все исправные процессы, либо ни один из них. Если генерал исправен, то все исправные процессы принимают решение.
2. **Договоренность.** Если исправные процессы принимают решение, то у всех процессов этим решением является одно и то же значение.
3. **Зависимость.** Если генерал исправен, то все процессы принимают в качестве решения входное значение генерала.

Можно показать, что устойчивость асинхронного алгоритма широковещательной рассылки, подверженного неисправностям византийского типа, ограничена величиной $t < N/3$; доказательство этого утверждения подобно доказательству теоремы 14.25. В алгоритме широковещательной рассылки, предложенном в работе [35] (алгоритм 14.6, задействованы три типа бюллетеней голосования: *исходные* сообщения (тип **in**), *ретранслированные* сообщения (тип **ec**) и сообщения *готовности* (тип **re**). Каждый процесс подсчитывает, сколько сообщений каждого типа с различными значениями было получено, учитывая при этом только по одному сообщению каждого типа, полученному от одного и того же процесса-отправителя.

```
var  $msgs_p[(in, ec, re), 0..1]$  : integer  init0 ;
```

Только для генерала:

```
shout  $\langle vote, in, x_p \rangle$ 
```

Для всех процессов:

```
while  $y_p = b$  do
  begin receive  $\langle vote, t, v \rangle$  from  $q$ ;
    if сообщение  $\langle vote, t, * \rangle$  уже было получено от процесса  $q$ 
      then skip (*  $q$  повторяется, проигнорировать *)
    else if  $t = in$  and  $q \neq g$ 
      then skip (*  $q$  выдает себя за процесс  $g$ , должно быть «византиец» *)
    else begin  $msgs_p[t, v] := msgs_p[t, v] + 1$ ;
      case  $t$  of
        in : if  $msgs_p[in, v] = 1$ 
              then shout  $\langle vote, ec, v \rangle$ 
        ec : if  $msgs_p[ec, v] = \lceil (N + t)/2 \rceil + 1$ 
              then shout  $\langle vote, re, v \rangle$ 
        re : if  $msgs_p[re, v] = t + 1$ 
              then shout  $\langle vote, re, v \rangle$ ;
              if  $msgs_p[re, v] = 2t + 1$ 
                then  $y_p := v$ 
      esac
    end
  end
```

Алгоритм 14.6. Алгоритмом широковещательной рассылки, робастным относительно неисправностей византийского типа

Генерал инициирует широковещательный обмен сообщениями, провозглашая свой исходный бюллетень. После получения исходного бюллетеня от генерала всякий процесс провозглашает ретранслированный бюллетень, содержащий то же самое значение. После того как процесс получит более $(N + t)/2$ ретранслированных сообщений, содержащих значение v , он провозглашает сообщение готовности. Количество ретранслированных сообщений, которое требуется получить, достаточно велико, и это позволяет гарантировать, что ни один исправ-

ный процесс не будет отправлять сообщения готовности, содержащие различные значения (лемма 14.37). Получение более t сообщений готовности с одним и тем же значением (а это означает, что хотя бы один исправный процесс отправил такое сообщение) также заставляет процесс провозглашать сообщения готовности. После получения более $2t$ сообщений готовности с одним и тем же значением (отсюда следует, что более t исправных процессов отправили такое сообщение) процесс принимает это значение в качестве решения. Алгоритм 14.6 не препятствует исправным процессам провозглашать сообщения готовности дважды, но повторные сообщения будут проигнорированы исправными процессами.

Лемма 14.37. *Никакие два исправных процесса не отправляют сообщения готовности с разными значениями.*

Доказательство. Исправный процесс принимает во внимание не более одного исходного сообщения от генерала, и поэтому отправленные им ретранслированные сообщения будут содержать одно и то же значение.

Рассмотрим тот процесс p , который первым из всех процессов отправил сообщение готовности, содержащее значение v , и тот процесс q , который первым из всех процессов отправил сообщение готовности, содержащее значение w . Хотя сообщение готовности можно отправлять после получения достаточно большого числа сообщений готовности, первый процесс, который отправляет сообщение готовности, делает это в силу других причин. Ведь если получение $t + 1$ сообщений готовности вынуждает процесс отправить, в свою очередь, сообщение готовности, то это означает, что среди полученных $t + 1$ сообщений хотя бы одно сообщение готовности поступило от исправного процесса. Таким образом, чтобы отправить сообщение готовности, процесс p должен получить ретранслированные сообщения, содержащие значение v , от более чем $(N + t)/2$ процессов, и процесс q должен получить ретранслированные сообщения от более чем $(N + t)/2$ процессов.

Поскольку всего имеется N процессов и $t < N/3$, найдутся более чем t процессов, среди которых есть хотя бы один исправный процесс r , которые отправили процессу p ретранслированные сообщения, содержащие значение v , а процессу q — ретранслированные сообщения, содержащие значение w . Так как процесс r исправен, $v = w$. $\square$

Лемма 14.38. *Если исправный процесс принимает решение, то и все исправные процессы принимают решение, и принятое решение будет одним и тем же.*

Доказательство. Чтобы принять значение v в качестве решения, процессу нужно получить более $2t$ сообщений готовности, содержащих значение v . Среди этих сообщений более t сообщений готовности были отправлены исправными процессами. Согласно лемме 14.37 все значения в указанных сообщениях согласованы друг с другом.

Предположим, что исправный процесс p принял в качестве решения значение v . Это означает, что процесс p получил более $2t$ сообщений готовности и среди них имелось более t сообщений, отправленных исправными процессами. Исправ-

ный процесс, отправивший сообщение готовности процессу p , отправляет это же сообщение и всем другим процессам. Таким образом, все исправные процессы получили более t сообщений готовности. Отсюда, в свою очередь, следует, что все исправные процессы отправили сообщения готовности, и поэтому каждый исправный процесс в конечном итоге получает $N - t > 2t$ сообщений готовности, и это дает ему возможность принять решение. $\square$

Лемма 14.39. *Если генерал исправен, то все исправные процессы принимают в качестве решения входные данные генерала.*

Доказательство. Если генерал исправен, он не отправляет исходных сообщений со значениями, отличными от его входных данных. Поэтому ни один исправный процесс не отправит ретранслированное сообщение, содержащее значение, отличное от входных данных генерала. Значит, неверные ретранслированные сообщения могут отправить не более t процессов. Этого количества неверных сообщений недостаточно, для того чтобы исправный процесс мог отправить сообщение готовности с неверным значением. Поэтому неверные сообщения готовности могут отправить самое большее t процессов. Этого количества неверных сообщений готовности недостаточно, для того чтобы исправный процесс мог отправить неверное сообщение готовности или принять неверное решение. Таким образом, ни один исправный процесс не отправляет неверных сообщений готовности и не принимает неверные решения.

Если генерал исправен, то он отправляет исходный бюллетень со своими входными данными всем исправным процессам, и все исправные процессы провозглашают в своих ретранслированных сообщениях это значение. Следовательно, все исправные процессы получают не менее $N - t > (N + t)/2$ верных ретранслированных сообщений и провозгласят сообщения готовности, в которых будет содержаться правильное значение. Таким образом, все исправные процессы получают не менее $N - t > 2t$ правильных сообщений готовности и примут верное решение. $\square$

Теорема 14.40. *Алгоритм 14.6 является асинхронным алгоритмом широковещательной рассылки, робастным относительно t неисправностей византийского типа, при $t < N/3$.*

Доказательство. Слабая завершаемость следует из лемм 14.39 и 14.38, условие договоренности выполнено в силу леммы 14.38, а условие зависимости — в силу леммы 14.39. $\square$

14.6. Упражнения к главе 14

14.6.1

Упражнение 14.1. Если упустить хотя бы одно из трех требований, приведенных в определении 14.3 (завершаемость, договоренность, нетривиальность), то задача консенсуса будет иметь очень простое решение. Покажите, что это действительно так, и приведите эти три простых решения.

Упражнение 14.2. В доказательстве леммы 14.6 предполагается, что каждый из 2^N вариантов присваивания одного бита N процессам порождает допустимую начальную конфигурацию.

Постройте детерминированные протоколы консенсуса, робастные относительно выхода из строя одного процесса, для каждого из приведенных ниже ограничений на распределение входных значений.

1. В каждой начальной конфигурации сумма всех входных значений должна быть четным числом (т. е. четное число процессов обладают входным значением, равным 1).

2. Выделены два процесса r_1 и r_2 , и для каждой начальной конфигурации выполняется соотношение $x_{r_1} = x_{r_2}$.

3. В каждой начальной конфигурации имеется не менее $\lceil (N/2) + 1 \rceil$ процессов, имеющих одно и то же значение.

14.6.2

Упражнение 14.3. Покажите, что в случае, когда $t \geq N/2$, не существует алгоритмов избрания лидера, которые были бы робастными относительно неисправности t процессов.

14.6.3

Упражнение 14.4. Покажите, что никакой алгоритм ϵ -приближенной договоренности не может быть стойким относительно выхода из строя $t \geq N/2$ процессов.

Упражнение 14.5. Какова должна быть биекция из множества

$$\{(s, r) : N - t \leq s \leq N \text{ и } 1 \leq r \leq s\}$$

на множество целых чисел из отрезка $[1, \dots, K]$.

Курсовая работа 14.6. Является ли алгоритм 14.2 нетривиальным?

Упражнение 14.7. Как можно приспособить доказательство теоремы 14.15 для случая системы G_T , состоящей из k связанных компонент.

Упражнение 14.8. В этом упражнении мы будем рассматривать обобщенную задачу $[k, l]$ -избрания лидеров, в которой требуется, чтобы каждый процесс принял одно из двух решений 0 («процесс проиграл выборы» или 1 («процесс избран лидером») и при этом число процессов, принявших решение 1 было не меньше k и не больше l .

1. Где могут быть использованы алгоритмы $[k, l]$ -избрания лидеров?

2. Покажите, что не существует детерминированных алгоритмов $[k, k]$ -избрания лидеров (при $0 < k < N$), робастных относительно выхода из строя одного процесса.

3. Постройте детерминированный алгоритм $[k, k + 2t]$ -избрания лидера, робастный относительно выхода из строя t процессов.

14.6.4

Упражнение 14.9. Верно ли, что в случае соблюдения требования сходимости математическое ожидание числа шагов алгоритма будет ограничено? Являет-

ся ли ограниченным математическое ожидание числа шагов для всех алгоритмов, приведенных в этом параграфе?

Упражнение 14.10. Покажите, что если все исправные процессы начинают выполнение k -го тура алгоритма консенсуса, робастного относительно выхода процессов из строя (алгоритм 14.3), то все исправные процессы также и завершат выполнение k -го тура алгоритма.

Упражнение 14.11. 1. Докажите, что если более $(N + t)/2$ процессов начинают выполнение алгоритма консенсуса, робастного относительно выхода процессов из строя (алгоритм 14.3), имея на входе значение v , то решение v будет принято по окончании третьего тура.

2. Докажите, что если более $(N - t)/2$ процессов начинают выполнение алгоритма, имея на входе значение v , то v может быть принято в качестве решения.

3. Может ли быть принято решение v , если в точности $(N - t)/2$ процессов начинают выполнение алгоритма, имея на входе значение v ?

4. Какова бивалентная начальная конфигурация этого алгоритма?

Упражнение 14.12. 1. Докажите, что если более $(N + t)/2$ исправных процессов начинают выполнение алгоритма 14.5 с входным значением v , то значение v будет в конечном итоге принято в качестве решения.

2. Докажите, что если более $(N + t)/2$ исправных процессов начинают выполнение алгоритма 14.5 с входным значением v и $t < N/5$, то решение v будет принято за два тура.

14.6.5

Упражнение 14.13. Докажите, что в случае $t \geq N/3$ невозможно построить асинхронный алгоритм широковещательной рассылки, который был бы робастным относительно t неисправностей византийского типа.

Упражнение 14.14. Докажите, что по ходу выполнения алгоритма 14.6 исправные процессы отправляют не более $N(3N + 1)$ сообщений.

ГЛАВА 15

ОТКАЗОУСТОЙЧИВОСТЬ В СИНХРОННЫХ СИСТЕМАХ

В предыдущей главе мы изучили, в какой мере отказоустойчивыми могут быть вполне асинхронные системы. И хотя приемлемая робастность достижима, на практике надежные системы всегда оказываются синхронными, поскольку в них используются таймеры и верхние оценки времени доставки сообщений. В таких системах достигим более высокий уровень робастности, алгоритмы оказываются проще, и, кроме того, в большинстве случаев эти алгоритмы гарантируют получение верхней оценки времени реакции системы.

Если система является синхронной, то неисправные процессы, не передающие информацию, не могут вводить тем самым в заблуждение правильно работающие процессы; и в самом деле, как только какой-либо процесс не получит ожидаемой информации, он будет использовать вместо нее заранее обусловленное значение и будет считать неисправным тот процесс, который должен был отправить ему эту информацию. Таким образом, вышедшие из строя процессы будут немедленно обнаружены, и они не создадут больших трудностей для работы синхронных систем. В этой главе основное внимание будет приковано к неисправностям византийского типа.

В § 15.4.1 изучается задача широковещательного распространения информации в синхронных сетях; мы установим верхнюю оценку устойчивости ($t < N/3$), а также рассмотрим два алгоритма, имеющих оптимальную устойчивость. Это детерминированные алгоритмы, достигающие консенсуса; в каждом из них предполагается, что все процессы знают, в какой момент времени начинается широковещательная связь. Поскольку в детерминированных асинхронных системах нельзя гарантировать достижения консенсуса (теорема 14.8), это означает, что в случае возникновения неисправностей (даже когда из строя выходит только один процесс) вычислительные способности синхронных систем превосходят вычислительные способности асинхронных систем.

Так как в синхронных системах можно обнаруживать неисправности, обусловленные выходом процессов из строя, эти неисправности можно считать «безвредными». Поэтому нарушать правильность вычислений способны только византийские процессы, которые либо сообщают неверную информацию о своем состоянии, либо искажают ретранслируемые данные. В § 15.4.2 будет показано, что робастность синхронных систем можно улучшить еще более, снабдив эти системы средствами аутентификации информации. Методы аутентификации не позволяют злонамеренным процессам распространять «ложные сведения» о той информации, которую они получили от других процессов, хотя эти методы не исключают возможности отправления неточной информации о внутреннем состоя-

нии самого процесса. Мы покажем также, что на практике для осуществления аутентификации можно воспользоваться методами криптографии.

При рассмотрении алгоритмов в §§ 15.4.1 и 15.4.2 мы придерживаемся некоторой идеальной модели синхронных систем, предполагая, что вычисление системы разбивается на *такты* (их также называют *турами* или *раундами*, см. гл. 12). Существенно более высокая устойчивость синхронных систем по сравнению с асинхронными означает, что средствами асинхронной модели невозможно реализовать детерминированную модель вычислений, которая работает по тактам и при этом является робастной относительно выхода из строя одного процесса. Как было показано в § 12.6.3, в надежных сетях такую реализацию, которая называется *синхронизатором*, можно осуществить.

Однако импульсную модель вычислений можно реализовать в асинхронных сетях с ограниченной задержкой передачи сообщений (§ 12.1.3), в которых процессы снабжены таймерами и осведомлены о верхней оценке времени доставки сообщений. Эта реализация оказывается возможной даже в том случае, когда часы отстают или спешат и до одной трети процессов ведут себя неправильно. Наиболее трудная часть реализации — это надежная синхронизация процессных часов; обсуждению этой задачи посвящен § 15.4.3.

15.1. Синхронные протоколы принятия решения

В этом параграфе мы рассмотрим робастные относительно неисправностей византийского типа алгоритмы для широковещательного распространения информации в синхронных (импульсных) сетях. Вначале мы напомним читателям об устройстве модели импульсных сетей, описанной в § 12.1.1. В синхронных сетях работа процессов происходит по тактам, которые нумеруются натуральными числами 1, 2, 3, и т. д. Каждый процесс может работать на протяжении сколь угодно большого числа тактов, пока не завершится его локальный алгоритм. Начальная конфигурация γ_0 определяется начальными состояниями процессов; конфигурация, которая образуется после i -го такта (условимся обозначать ее символом γ_i), также определяется внутренними состояниями процессов. В i -м такте каждый процесс сначала отправляет конечное множество сообщений, зависящих от того состояния, в котором пребывает этот процесс в конфигурации $conf_{i-1}$. Затем каждый процесс получает все те сообщения, которые были ему отправлены в текущем такте, и определяет то новое внутреннее состояние, в которое он должен перейти в зависимости от прежнего состояния и совокупности сообщений, полученных им в текущем такте вычисления.

Импульсная модель — это идеализированная модель синхронных вычислений. В этой модели синхронность проявляется в том, что

- 1) переходы во всех процессах происходят одновременно;
- 2) в каждом такте сообщения поступают в процесс еще до того, как происходит переход из одного состояния в другое;

Эти идеализированные допущения можно ослабить, ограничившись более реалистичными предположениями о том, что 1) процессы имеют доступ к аппаратным часам и 2) существует верхняя оценка времени доставки сообщений. Полу-

чившаяся таким образом модель *асинхронных сетей с ограниченной задержкой* позволяет очень эффективно реализовать импульсную модель (см. § 12.1.3). Как было показано в гл. 12, одновременное осуществление переходов в процессах — это всего лишь видимость: при реализации этой модели переходы в процессах могут происходить в разное время при том условии, что будет обеспечено своевременное получение всех сообщений. Кроме того, реализация модели должна быть такой, чтобы процесс мог работать на протяжении неограниченно большого числа тактов. В силу этого требования мы не можем воспользоваться для решения задач отказоустойчивости теми реализациями, которые были предложены в гл. 12, поскольку всем им присуща способность к блокировке и большинство этих реализаций могут быть заблокированными даже при потере одного единственного сообщения. Как мы уже сказали, робастные реализации импульсной модели будут изучаться в параграфе 15.4.3.

Поскольку импульсная модель гарантирует доставку отправленных сообщений в том же самом такте, всякий процесс может определить, что его сосед *не отправлял ему сообщений*. Асинхронные системы лишены этого свойства, но именно благодаря ему, как мы вскоре увидим, и задача консенсуса, и даже задача надежного широковещательного распространения информации могут быть решены для синхронных систем.

Постановка задачи широковещательного распространения информации в случае неисправностей византийского типа такова. В системе выделен некоторый процесс g , который называется *генералом*, и этот процесс располагает входными данными x_g из некоторого множества V (обычно $V = \{0, 1\}$). Все остальные процессы называются *лейтенантами*. Необходимо соблюсти следующие три требования.

1. **Завершение.** Каждый исправный процесс p завершает свою работу и принимает в качестве решения значение $y_p \in V$.
2. **Согласованность.** Все исправные процессы принимают в качестве решения одно и то же значение.
3. **Зависимость.** Если сам генерал исправен, то все исправные процессы принимают в качестве решения значение x_g .

Кроме того, можно потребовать, чтобы соблюдалось свойство **одновременности**, т. е. чтобы все исправные процессы принимали решение в одном и том же такте. Все алгоритмы, которые мы будем обсуждать в этом параграфе, обладают свойством одновременности (более подробно об этом говорится в параграфе 15.2.6).

15.1.1. Границы устойчивости

Так же как и для асинхронных сетей (см. теорему 14.25), устойчивость t синхронных сетей в случае возникновения неисправностей византийского типа ограничена величиной $N/3$. Эта оценка была впервые установлена в работе Писа, Шостака и Лампорта [157], в которой было описано несколько сценариев поведения системы, содержащей $N/3$ или более византийских процессов. Здесь, в отличие от тех сценариев, которые рассматриваются в доказательстве теоре-

мы 14.25, исправные процессы получают противоречивую информацию, и это позволяет им прийти к заключению о том, что некоторые процессы неисправны. Тем не менее, нельзя установить, какие из процессов ненадежны, и поэтому неисправные процессы могут заставить принять неправильное решение.

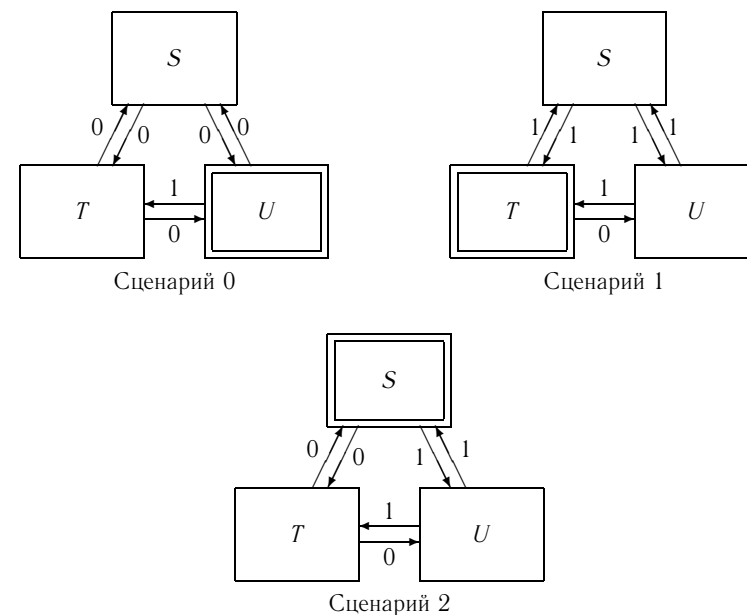


Рис. 15.1. Сценарии доказательства теоремы 15.1.

Теорема 15.1. *Не существует протокола широковещательного распространения информации, который был бы устойчив относительно t неисправностей византийского типа в том случае, когда $t \geq N/3$.*

Доказательство. Как и в предыдущих доказательствах, уровень устойчивости $N/3$ или более высокий позволяет разбить все процессы на три группы (S , T и U), каждая из которых может целиком состоять из неисправных процессов. Группу, в которой содержится генерал, обозначим символом S . Чтобы выявить противоречие, надо рассмотреть три сценария, изображенных на рис. 15.1; группа неисправных процессов выделена двойной рамкой.

В сценарии 0 генерал начинает широковещательное распространение значения 0 и все процессы группы U неисправны; в сценарии 1 генерал начинает широковещательное распространение значения 1 и все процессы группы T неисправны. В i -м такте сценария 0 все процессы группы U отправляют процессам группы T точно такие же сообщения, которые им следовало бы отправлять согласно протоколу в сценарии 1. Иными словами, процессам группы T отправляются в точности те сообщения, которые вырабатываются в ответ на сообщения, полученные в $(i-1)$ -м такте сценария 1. Процессам из группы S отправляются те

сообщения, которые следует отправлять, руководствуясь протоколом. Процессы, относящиеся к группам S и T , конечно же, отправляют правильные сообщения в каждом такте работы алгоритма. Заметим, что в этом сценарии только процессы группы U отправляют неверные сообщения и в спецификациях протокола указано, что все исправные процессы, включая те, которые относятся к группе T , должны принять в качестве решения значение 0.

Сценарий 1 определяется аналогичным образом, но здесь неисправными являются процессы из группы T , и они отправляют те сообщения, которые им следовало бы отправлять согласно сценарию 0. В сценарии 1 все процессы из группы U принимают в качестве решения значение 1.

И, наконец, рассмотрим сценарий 2, в котором все процессы из группы S неисправны и ведут себя следующим образом. Процессам из группы T они отправляют те сообщения, которые нужно отправлять согласно сценарию 0, а процессам из группы U они отправляют те сообщения, которые нужно отправлять согласно сценарию 1. Теперь индукцией по числу тактов можно показать, что все сообщения, которые отправляют процессы группы T процессам группы U (или процессы группы U процессам группы T), — это в точности те сообщения, которые отправляются, когда алгоритм работает по сценарию 0 (или, соответственно, когда он работает по сценарию 1). Следовательно, для процессов из группы T сценарий 2 неотличим от сценария 0, а для процессов из группы U этот сценарий неотличим от сценария 1. Отсюда следует, что все процессы из группы T примут в качестве решения значение 0, а все процессы из группы U примут в качестве решения значение 1 вопреки условию согласованности. $\square$

В доказательстве теоремы мы воспользовались тем, что византийские процессы могут отправлять сообщения, возникающие по ходу сценария 1, даже тогда, когда эти процессы получают только те сообщения, которые возникают по ходу сценария 0. Это означает, что неисправные процессы могут сообщать «ложные» сведения не только о своих внутренних состояниях, но также и о тех сообщениях, которые были ими получены. Именно эту возможность позволяют устранить средства аутентификации, которые мы рассмотрим в § 15.4.2. Эти средства позволяют поднять уровень устойчивости до величины $N - 1$.

15.1.2. Алгоритм широковещательного распространения информации при наличии неисправностей византийского типа

В этом параграфе мы покажем, что верхняя оценка устойчивости, о которой было говорилось ранее, является точной. Более того, в отличие от асинхронных сетей, максимальная устойчивость достигается в том случае, когда используются детерминированные алгоритмы. Мы представим вниманию читателей алгоритм, описанный Писом и др. в работе [157], который способен выдержать t византийских неисправностей для любого значения $t < N/3$. Уровень устойчивости является параметром этого алгоритма.

В качестве алгоритма $Broadcast(N, 0)$ мы будем использовать алгоритм 15.2; он неустойчив ни к каким отказам ($t = 0$), и если никаких отказов не происходит,

Такт

- 1: Генерал отправляет сообщение $\langle \text{value}, x_g \rangle$ всем процессам, лейтенанты не отправляют никаких сообщений.
получение сообщений в первом такте.
 Генерал принимает в качестве решения значение x_g .
 Лейтенанты принимают решение так:
 if сообщение $\langle \text{value}, x \rangle$ было получено от процесса g в первом такте
 then принять в качестве решения значение x
 else принять в качестве решения неопределенное значение $undef$

Алгоритм 15.2. $Broadcast(N, 0)$

то все процессы в первом такте принимают в качестве решения входные данные генерала. Если возникают отказы, то свойство согласованности может быть нарушено, но свойство завершаемости (равно как и одновременности) обязательно будет выполнено.

Такт

- 1: Генерал отправляет сообщение $\langle \text{value}, x_g \rangle$ всем процессам, лейтенанты не отправляют никаких сообщений.
получение сообщений в первом такте.
 Лейтенант p ведет себя так:
 if сообщение $\langle \text{value}, x \rangle$ было получено от процесса g в первом такте
 then $x_p := x$ **else** $x_p := undef$;
 Оповестить в следующем такте о значении x_p остальных лейтенантов, поступая так, как это делает генерал в алгоритме $Broadcast_p(N - 1, t - 1)$.
- $(t + 1)$: **получить сообщения, отправленные в $(t + 1)$ -такте.**
 Генерал принимает в качестве решения значение x_g .
 Лейтенант p :
 вычисляет решение, которое предлагает алгоритм $Broadcast_q(N - 1, t - 1)$ для каждого лейтенанта q .
 $W_p[q] :=$ решение, предложенное $Broadcast_q(N - 1, t - 1)$;
 $y_p := major(W_p)$

Алгоритм 15.3. $Broadcast(N, t)$ (для $t > 0$)

Протокол с уровнем устойчивости $t > 0$ (алгоритм 15.3) осуществляет рекурсивный вызов той же самой процедуры с уровнем стойкости $t - 1$. Генерал отправляет входные данные всем лейтенантам в первом такте, и в следующем такте каждый лейтенант оповещает о полученном значении всех остальных лейтенантов, но для этого используется широковещательная связь с уровнем устойчивости $t - 1$. Именно это уменьшение уровня устойчивости является тонким местом алгоритма, потому что (в том случае, если генерал исправен) все t византийских

процессов могут оказаться среди лейтенантов, и поэтому настоящее число неисправностей может превосходить уровень устойчивости вложенного вызова процедуры *Broadcast*. Чтобы доказать корректность получившегося в результате алгоритма, необходимо принимать во внимание уровень устойчивости *t* а также настоящее число неисправных процессов *f* (см. лемму 15.3). В $(t + 1)$ -м такте вложенные вызовы выдают решение, и поэтому лейтенант *p* принимает решение в $N - 1$ вложенных сеансах широковещательной рассылки сообщений. Эти $N - 1$ решений помещаются в массив W_p , и процесс *p* принимает в качестве решения наиболее распространенное значение (значение, которое было получено непосредственно от генерала, здесь во внимание не принимается!). Для этого вводится детерминированная функция *major*, которая определяется на массивах и обладает тем свойством, что всякий раз, когда большинство элементов массива *W* имеют значение *v*, (т. е. более половины элементов массива имеют значение *v*), выполняется равенство $major(W) = v$.

Лемма 15.2 (завершаемость). *Если выполнение алгоритма $Broadcast(N, t)$ начинается в первом такте, то каждый процесс принимает решение в $(t + 1)$ -м такте.*

Доказательство. Поскольку протокол является рекурсивным, его свойства также доказываются рекурсивно с использованием параметра *t*.

В алгоритме $Broadcast(N, 0)$ (алгоритм 15.2) каждый процесс принимает решение в первом такте.

В алгоритме $Broadcast(N, t)$ лейтенанты осуществляют рекурсивные вызовы алгоритма $Broadcast(N - 1, t - 1)$ во втором такте. При запуске в первом такте алгоритм приводит к решению в *t*-м такте (это индуктивная гипотеза), и это означает, что при запуске во втором такте все вложенные вызовы приведут к принятию решения в $(t + 1)$ -м такте. В этом же такте принимает решение и алгоритм $Broadcast(N, t)$. $\square$

Чтобы обосновать свойство зависимости (также по индукции), условимся считать, что генерал является исправным процессом и, таким образом, все *t* исправных процессов содержатся среди $N - 1$ лейтенантов. Так как неравенство $t < (N - 1)/3$ не обязательно верно, мы не можем воспользоваться простой индукцией, и в своих рассуждениях мы будем учитывать настоящее число неисправностей, которое условимся обозначать символом *f*.

Лемма 15.3 (зависимость). *Если генерал исправен, и при этом в системе имеется *f* неисправных процессов, и $N > 2f + t$, то все исправные процессы принимают в качестве решения входные данные генерала.*

Доказательство. Как видно из описания алгоритма $Broadcast(N, 0)$, если генерал исправен, то все исправные процессы принимают в качестве решения то значение, которое имеется на входе у генерала.

Теперь предположим, что рассматриваемая лемма верна для алгоритма $Broadcast(N - 1, t - 1)$. Так как генерал исправен, он отправляет в первом такте свое входное значение всем лейтенантам, и поэтому каждый исправный лейтенант *q* выбирает $x_q = x_g$. Из неравенства $N > 2f + t$ следует неравенство $(N - 1) > 2f + (t - 1)$,

и поэтому индуктивную гипотезу можно применить к вложенным вызовам, хотя все *f* неисправных процессов находятся среди лейтенантов. Таким образом, для исправных лейтенантов *p* и *q* решение, принятое процессом *p* в алгоритме $Broadcast_q(N - 1, t - 1)$, будет равно x_q , т. е. будет равно x_g . Но с учетом того, что подавляющее большинство лейтенантов ($N > 2f + t$) исправны, процесс завершит работу, построив вектор W_p , в котором подавляющее большинство элементов имеют значение, равное x_g . Поэтому после применения процедуры *major* процесс *p* вычислит искомое значение x_g . $\square$

Лемма 15.4 (согласованность). *Все исправные процессы принимают в качестве решения одно и то же значение.*

Доказательство. Поскольку свойство зависимости влечет за собой и свойство согласованности во всех тех выполнениях, в которых исправен генерал, нам остается обратить внимание только на те случаи, когда генерал неисправен. Но тогда неисправными могут быть не более $t - 1$ лейтенантов, и отсюда следует, что вложенные вызовы выполняются в пределах тех ограничений, которые налагаются на уровень их устойчивости!

Действительно, из неравенства $t < N/3$ следует, что $t - 1 < (N - 1)/3$, и поэтому вложенные вызовы удовлетворяют условию согласованности. Поэтому все исправные лейтенанты примут в качестве решения одно и то же значение x_q для каждого лейтенанта *q* во вложенных вызовах $Broadcast_q(N - 1, t - 1)$. Таким образом, в $(t + 1)$ -м такте каждый исправный лейтенант вычисляет один и тот же вектор *W*, и отсюда следует, что после применения функции *major* все исправные процессы примут одно и то же решение. $\square$

Теорема 15.5. *Протокол $Broadcast(N, t)$ (алгоритм 15.2/15.3) является протоколом широковещательной рассылки сообщений, который устойчив относительно *t* неисправностей византийского типа для $t < N/3$.*

Доказательство. Свойство завершаемости было установлено в лемме 15.2, свойство зависимости — в лемме 15.3, а свойство согласованности — в лемме 15.4. $\square$

В протоколе *Broadcast* решение принимается в $(t + 1)$ -м такте, и это является оптимальным сроком принятия решения (см § 15.2.6). К сожалению, сложность по числу обменов сообщениями растет экспоненциально (см. упражнение 15.1).

15.1.3. Алгоритм широковещательного распространения информации, имеющий полиномиальную сложность

В этом параграфе мы представим византийский алгоритм широковещательного распространения информации, который был предложен Долевым и др. в работе [72]. В этом алгоритме используется только полиномиальное число сообщений и битов, хотя его сложность по времени больше, чем у ранее рассмотренного протокола: для принятия решения новому алгоритму требуется $2t + 3$ тактов. В том описании, которое мы представим, предполагается, что $N = 3t + 1$, а как поступить в случае, когда $N > 3t + 1$, мы обсудим позднее.

В алгоритме фигурируют две пороговые величины $L = t + 1$ и $H = 2t + 1$. Эти числа выбраны исходя из следующих соображений:

- в каждом множестве, состоящем из L процессов, есть хотя бы один исправный процесс,
- в каждом множестве, состоящем из H процессов, есть хотя бы L исправных процессов,
- в системе есть по крайней мере H исправных процессов.

Заметим, что неравенство $N \geq 3t + 1$ задает необходимое и достаточное условие выбора значений L и H , удовлетворяющих перечисленным трем свойствам.

В рассматриваемом алгоритме процессы обмениваются сообщениями вида $\langle \mathbf{bm}, v \rangle$, где v — это либо число 1, либо имя процесса. Здесь **bm** — аббревиатура словосочетания «broadcast message» (широковещательное сообщение). Каждый процесс p проводит учет поступивших сообщений в двумерной булевой таблице R , в которой элемент $R_p[q, v]$ имеет значение *true* в том и только том случае, когда процесс p уже получил сообщение $\langle \mathbf{bm}, v \rangle$ от процесса q . Первоначально все элементы таблицы имеют значение *false*, и мы будем считать, что изменение значений элементов таблицы в каждом такте происходит на этапе приема сообщений (об этом не говорится в описании алгоритма 15.4). Мы будем иметь в виду, что значения R_p монотонно возрастают по ходу вычисления, т. е. если в некотором такте $R_p[q, v]$ принимает значение *true*, то этот элемент будет сохранять значение *true* во всех последующих тактах. Принимая во внимание то, что исправные процессы занимаются только широковещательным оглашением своих сообщений, мы можем также заключить, что для любых исправных процессов p , q и r в конце каждого такта будет выполняться равенство $R_p[r, v] = R_q[r, v]$.

В отличие от протокола *Broadcast*, описанного в предыдущем параграфе, протокол Долева и др. является асимметричным по отношению к значениям 0 и 1. Решение 0 принимается по умолчанию в том случае, если в системе было передано недостаточно большое количество сообщений. Если генерал имеет 1 в качестве входных данных, то он широковещательно распространяет сообщение $\langle \mathbf{bm}, 1 \rangle$ и всякий процесс, получивший достаточно много ретранслированных сообщений вида $\langle \mathbf{bm}, q \rangle$, принимает решение со значением 1.

В рассматриваемом алгоритме имеются три режима работы процессов: *пусковой режим*, *режим поддержки* и *режим одобрения*.

1. *Режим поддержки*. Процесс p поддерживает процесс q в i -м такте, если p уже получил в предыдущих тактах достаточно убедительное свидетельство того, что процесс q отправил сообщения $\langle \mathbf{bm}, 1 \rangle$. Если это произошло, то процесс p в i -м такте будет отправлять сообщения $\langle \mathbf{bm}, q \rangle$. Говорят, что процесс p непосредственно поддерживает процесс q , если он получил сообщение $\langle \mathbf{bm}, 1 \rangle$ от процесса q . Говорят, что процесс p косвенно поддерживает процесс q , если p получил сообщение $\langle \mathbf{bm}, q \rangle$ по меньшей мере от L процессов. Множество S_p процессов, поддерживаемых процессом p , можно определить неявно из

таблицы R_p , руководствуясь следующими соотношениями:

$$\begin{aligned} DS_p &= \{q : R_p[q, 1]\}, \\ IS_p &= \{q : \#\{r : R_p[r, q]\} \geq L\}, \\ S_p &= DS_p \cup IS_p. \end{aligned}$$

Порог, который выбран для оказания косвенной поддержки, обусловлен тем обстоятельством, что если некоторый исправный процесс оказывает поддержку процессу q , то q уже отправил хотя бы одно сообщение $\langle \mathbf{bm}, 1 \rangle$. Действительно, предположим, что некоторый исправный процесс поддерживает процесс q , и пусть впервые это произошло в i -м такте. Так как для косвенной поддержки процесса q требуется получить хотя бы одно сообщение $\langle \mathbf{bm}, q \rangle$ от исправного процесса в предыдущих тактах, первая поддержка процесса q может быть только прямой. Прямая поддержка, которую оказывает исправный процесс, подразумевает, что этот процесс получил сообщение $\langle \mathbf{bm}, 1 \rangle$ от процесса q .

2. *Режим одобрения*. Процесс p одобряет выбор, сделанный процессом q , после получения сообщения $\langle \mathbf{bm}, q \rangle$ от H процессов, и множество процессов, решение которых одобряется процессом p , определяется так:

$$C_p = \{q : \#\{r : R_p[r, q]\} \geq H\}.$$

Порог для выражения одобрения выбран таким образом, что если исправный процесс p одобряет выбор, сделанный процессом q , то все исправные процессы самое позднее в следующем такте одобряют выбор, сделанный процессом q . Действительно, предположим, что процесс p одобряет выбор процесса q после i -го такта. Процесс p уже получил сообщения $\langle \mathbf{bm}, q \rangle$ от H процессов, среди которых есть (согласно определению порогового значения) не менее L исправных процессов, поддерживающих процесс q . Исправные процессы, оказывающие поддержку процессу q , отправляют сообщение $\langle \mathbf{bm}, q \rangle$ всем процессам, и это означает, что в i -м такте все исправные процессы получают не менее L сообщений $\langle \mathbf{bm}, q \rangle$ и оказывают поддержку процессу q в $(i + 1)$ -м такте. Таким образом, в $(i + 1)$ -м такте все исправные процессы отправляют сообщение $\langle \mathbf{bm}, q \rangle$, и, поскольку число исправных процессов превосходит величину H , каждый исправный процесс получает достаточную поддержку, для того чтобы одобрить выбор q .

3. *Пусковой режим*. Процесс p переходит в пусковой режим, когда он получает достаточно убедительное свидетельство в пользу того, чтобы принять в качестве решения значение 1. После перехода в пусковой режим процесс p отправляет сообщения $\langle \mathbf{bm}, 1 \rangle$. Достаточно убедительными для перехода в этот режим признаются свидетельства следующих трех видов:

- процесс p — генерал, и $x_p = 1$,
- процесс p получает сообщение $\langle \mathbf{bm}, 1 \rangle$ от генерала в первом такте,
- процесс p одобрил выбор достаточно большого числа лейтенантов в конце предыдущего такта.

Последняя возможность заслуживает более пристального внимания, потому что «достаточное» число лейтенантов, выбор которых был одобрен, увеличивается по ходу выполнения протокола и одобренный генерал в этом случае не принимает-

ся в расчет. В первых трех тактах для перехода в пусковой режим необходимо одобрить выбор L лейтенантов, но начиная с четвертого такта этот порог увеличивается на 1 через каждые два такта. Таким образом, для перехода в пусковой режим по правилу 3) требуется, чтобы в конце i -го такта одобрение получили $Th(i) = L + \max(0, \lfloor i/2 \rfloor - 1)$ лейтенантов. В описании алгоритма запись C_p^L служит для обозначения множества лейтенантов, которые получили одобрение, т. е. это множество $C_p \setminus \{g\}$. Индикатором пребывания процесса p в пусковом режиме служит булева переменная ini_p .

Если исправный лейтенант r переходит в пусковой режим в конце i -го такта, то все исправные процессы одобряют выбор, сделанный процессом r , в конце $(i + 2)$ -го такта. Действительно, процесс r ширококестельно распространяет сообщение $\langle \mathbf{bm}, 1 \rangle$ в $(i + 1)$ -м такте, и поэтому все исправные процессы (непосредственно) поддерживают процесс r в $(i + 2)$ -м такте, вследствие чего каждый процесс в этом такте получает не менее H сообщений $\langle \mathbf{bm}, q \rangle$.

Работа алгоритма продолжается на протяжении $2t + 3$ тактов. Если процесс p одобрил по меньшей мере H процессов (не считая генерала) в конце последнего такта, то он выбирает в качестве решения значение 1; в противном случае он принимает в качестве решения значение 0. Таково описание алгоритма 15.4.

```

var  $R_p[\dots, \dots]$  : bool init false ;
     $ini_p$  : bool init if  $p = g \wedge x_p = 1$  then true else false ;

```

i -й такт: (* Этап отправления сообщений *)

```

if  $ini_p$  then ширококестельно отправить  $\langle \mathbf{bm}, 1 \rangle$  ;
forall  $q \in S_p$  do ширококестельно отправить  $\langle \mathbf{bm}, q \rangle$  ;
получить все сообщения, относящиеся к  $i$ -му такту ;
(* Изменение состояния *)
if  $i = 1$  and  $R_p[g, 1]$  then  $ini_p := true$  ;
if  $\#C_p^L \geq Th(i)$  then  $ini_p := true$  ;
if  $i = 2t + 3$  then (* Принять решение *)
    if  $\#C_p \geq H$  then  $y_p := 1$  else  $y_p := 0$ 

```

Алгоритм 15.4. Надежный протокол ширококестельной рассылки сообщений

Генерал, будучи единственным процессом, который способен в одиночку вынудить другие процессы перейти в пусковой режим, занимает ведущее положение в этом алгоритме. Как видно из описания алгоритма, если генерал начинает работу исправно, то возникает целая лавина сообщений, которая приводит к тому, что все исправные процессы одобряют выбор, сделанный H процессами, и принимают в качестве решения значение 1. И точно так же, если генерал не будет приступать к делу, то не образуется той «критической массы» сообщений, которая переведет исправные процессы в пусковой режим.

Лемма 15.6. Алгоритм 15.4 удовлетворяет условиям завершения, одновременности и зависимости.

Доказательство. Из описания алгоритма видно, что все исправные процессы принимают решение в конце $(2t + 3)$ -го такта, и поэтому свойства завершаемости и одновременности соблюдаются. Чтобы убедиться в том, что соблюдается свойство зависимости, мы будем предполагать, что генерал исправен.

Если генерал исправен и имеет на входе 1, то в первом такте он проводит ширококестельную рассылку сообщения $\langle \mathbf{bm}, 1 \rangle$ и тем самым вынуждает каждый исправный процесс q перейти в пусковой режим. Поэтому во втором такте каждый исправный процесс q ширококестельно отправит сообщение $\langle \mathbf{bm}, 1 \rangle$ и по окончании второго такта каждый исправный процесс p поддержит все другие исправные процессы. Отсюда следует, что в третьем такте каждый исправный процесс p ширококестельно отправит сообщение $\langle \mathbf{bm}, q \rangle$ с именем каждого исправного процесса q , и поэтому к концу третьего такта каждый исправный процесс получит сообщение $\langle \mathbf{bm}, q \rangle$ от каждого другого исправного процесса, и это вынудит его одобрить выбор, сделанный процессом q . Таким образом, в конце третьего такта каждый исправный процесс одобрит выбор, сделанный H процессами, и это означает, что окончательный выбор будет сделан в пользу значения 1. (Генерал будет поддержан и одобрен всеми исправными процессами на один такт раньше, чем все другие процессы.)

Если генерал исправен и имеет на входе значение 0, то в первом такте он не проводит ширококестельную рассылку сообщения $\langle \mathbf{bm}, 1 \rangle$, и этого не делает ни один другой исправный процесс. Предположим, что никакой исправный процесс не перешел в пусковой режим ни в одном такте начиная с первого и оканчивая $(i - 1)$ -м. Тогда ни один исправный процесс не отправит в i -м такте сообщение $\langle \mathbf{bm}, 1 \rangle$. В конце i -го такта ни один исправный процесс не будет поддерживать или одобрять какой-либо исправный процесс, поскольку, как мы уже показали ранее, отсюда следовало бы, что еще до начала i -го такта какой-нибудь исправный процесс отправил сообщение $\langle \mathbf{bm}, 1 \rangle$. Следовательно, ни один исправный процесс не перейдет в пусковой режим в конце i -го такта. Таким образом, ни один исправный процесс вообще не перейдет в пусковой режим. Отсюда следует, что ни один исправный процесс не будет поддерживать какой-либо исправный процесс и ни один исправный процесс не одобрит более t процессов, и поэтому в последнем такте будет принято решение 0. $\square$

Последующие леммы помогут доказать соблюдение условия согласованности. Далее мы будем полагать, что генерал является *неисправным* процессом. Достаточно большая «критическая масса» сообщений, которая неизбежно приведет процессы к принятию 1-решения, образуется в том случае, если не менее L исправных процессов перейдут в пусковой режим, имея в своем распоряжении хотя бы четыре такта до конца работы.

Лемма 15.7. Если L исправных процессов перейдут в пусковой режим в конце i -го такта, где $i < 2t$, то все исправные процессы примут в качестве решения значение 1.

Доказательство. Предположим, что в конце i -го такта впервые по ходу выполнения не менее L исправных процессов перешли в пусковой режим. Обозначим символом A множество исправных процессов, которые перешли в пус-

ковой режим в конце i -го такта. Все процессы из множества A — лейтенанты, поскольку генерал неисправен. В конце $(i + 2)$ -го такта все исправные процессы будут одобрять выбор, который сделали лейтенанты из множества A ; мы покажем, что к этому моменту времени все исправные процессы перейдут в пусковой режим.

Случай $i = 1$: Все исправные процессы одобрили выбор лейтенантов из множества A к концу третьего такта и перешли в пусковой режим, поскольку $\#A \geq L = Th(3)$.

Случай $i \geq 2$: Хотя бы один процесс, который мы условимся обозначать символом r , из множества A перешел в пусковой режим в i -м такте, так как он одобрил выбор, который сделали $Th(i)$ лейтенантов (переход в пусковой режим по получении сообщения $\langle \mathbf{bm}, 1 \rangle$ от генерала возможен только в первом такте). Выбор, сделанный $Th(i)$ лейтенантами, получает одобрение *всех* исправных процессов в конце $(i + 1)$ -го такта, но процесс r не относится к числу тех $Th(i)$ одобренных лейтенантов, поскольку он впервые отправляет сообщения $\langle \mathbf{bm}, 1 \rangle$ только в $(i + 1)$ -м такте. Однако все исправные процессы смогут одобрить выбор, сделанный процессом r в конце $(i + 2)$ -го такта; таким образом, в конце $(i + 2)$ -го такта одобрение получают не менее $Th(i) + 1$ лейтенантов. И тогда, коль скоро $Th(i + 2) = Th(i) + 1$, все исправные процессы перейдут в пусковой режим.

Теперь, ввиду того что все исправные процессы перешли в пусковой режим в конце $(i + 2)$ -го такта, все они получают одобрение (со стороны всех исправных процессов) в конце $(i + 4)$ -го такта. Таким образом, все исправные процессы будут одобрять выбор, сделанный по меньшей мере H лейтенантами. Так как мы условились считать, что $i < 2t$, имеет место неравенство $i + 4 \leq 2t + 3$ и все исправные процессы принимают в качестве решения значение 1. $\square$

Чтобы произвольный исправный процесс принял в качестве решения значение 1, необходимо создать «лаvinу» сообщений от не менее чем L исправных процессов, пребывающих в пусковом режиме. Действительно, для того чтобы выбрать в качестве решения значение 1, требуется получить одобрение от не менее чем H процессов, среди которых должны присутствовать L исправных процессов, уже пребывающих в пусковом режиме. Вопрос состоит в том, сумеют ли византийские процессы сговориться так, чтобы отсрочить приход этой лавины сообщений на достаточно долгое время и за счет этого подтолкнуть некоторые исправные процессы к принятию 1-решения, не вынуждая при этом все исправные процессы делать это согласно лемме 15.7. Конечно же, ответ на этот вопрос будет отрицательным, поскольку имеется определенный предел, на протяжении которого заговор византийских процессов может откладывать наступление этой лавины сообщений, и число тактов работы $2t + 3$ выбрано в точности таким, чтобы суметь предотвратить подобный сговор. Причиной тому служит все возрастающий порог требуемого числа одобренных процессов; в более поздних тактах работы порог становится столь высок, что для перевода очередного процесса в пусковой режим необходимо, чтобы L исправных лейтенантов уже пребывали в пусковом режиме.

Лемма 15.8. *Предположим, что по ходу работы алгоритма не менее L исправных процессов перешли в пусковой режим, и предположим, что это впервые произошло в конце i -го такта. Тогда $i < 2t$.*

Доказательство. Чтобы исправный процесс мог перейти в пусковой режим в $2t$ -м такте или позднее, он к этому моменту уже должен одобрить выбор, сделанный не менее чем $Th(2t) = L + (t - 1)$ лейтенантами. Так как генерал неисправен, среди лейтенантов неисправными могут оказаться не более $t - 1$ процессов, и это означает, что не менее L исправных лейтенантов должны были получить одобрение. Таким образом, в одном из более ранних тактов L исправных процессов должны были перейти в пусковой режим. $\square$

Теорема 15.9. *Алгоритм широковещательного распространения информации, предложенный Долевым и др. (алгоритм 15.4), устойчив относительно t неисправностей византийского типа.*

Доказательство. В лемме 15.6 мы установили, что этому алгоритму присущи свойства завершаемости, одновременности и зависимости. Теперь убедимся в том, что свойство согласованности также соблюдается рассматриваемым алгоритмом. Предположим, что какой-то исправный процесс принимает в качестве решения значение 1. Мы уже убедились в том, что для этого не менее L исправных процессов должны были перейти в пусковой режим. Согласно лемме 15.8 впервые это могло случиться в i -м такте работы алгоритма, где $i < 2t$. Но тогда согласно лемме 15.7 все исправные процессы принимают в качестве решения значение 1. $\square$

Чтобы упростить описание алгоритма, мы разрешили процессам повторять в каждом такте все те сообщения, которые были ими отправлены в предшествующих тактах работы. Так как исправные процессы записывают все сообщения, которые были получены ими ранее, никакой необходимости в таком повторе нет, и каждое сообщение достаточно отправлять только один раз. В связи с этим всякий исправный процесс может отправить каждое из $N + 1$ возможных сообщений не более одного раза любому другому процессу. Поэтому сложность по числу обменов сообщениями ограничена величиной $\Theta(N^3)$. По ходу работы алгоритма может передаваться только $N + 1$ различных сообщений, и это означает, что каждое сообщение состоит из $O(\log N)$ битов.

Если в системе содержится более $3t + 1$ процессов, то для выполнения алгоритма выбирается семейство, состоящее из $3t$ *активных лейтенантов*. Это чисто статистический выбор; например, можно выбрать $3t$ процессов, имена которых следуют по порядку вслед за именем генерала. Генерал и все активные лейтенанты оповещают пассивных лейтенантов о том решении, которое они приняли, и все пассивные лейтенанты принимают в качестве решения то значение, которое было принято более чем t процессами. Сложность по числу обменов сообщениями в этом случае будет оцениваться величиной $\Theta(t^3 + tN)$, а сложность по числу битов — величиной $\Theta(t^3 \log t + tN)$.

15.2. Протоколы с аутентификацией

Злонамеренное поведение, которое мы рассматривали до сих пор, проявлялось в том, что процессы неверно ретранслировали информацию и сообщали неверную информацию о своем внутреннем состоянии. К счастью, злой умысел, присущий византийским процессам, можно ограничить, воспользовавшись средствами криптографии, благодаря которым теорема 15.1 теряет свою силу. Действительно, в тех сценариях, которые обсуждаются в доказательстве этой теоремы, неисправные процессы могут отправлять сообщения, которые им положено отправлять в соответствии со сценарием 1, в то время как они получают сообщения, соответствующие сценарию 0.

В этом параграфе будут задействованы средства для создания *цифровой подписи* и *аутентификации* сообщений. Процесс p при отправлении сообщения M добавляет к нему некоторую специальную информацию $S_p(M)$; эта дополнительная информация называется *цифровой подписью* сообщения M , которую поставил процесс p . В отличие от рукописных подписей, цифровые подписи зависят от сообщений M , и благодаря этому простое копирование этой информации для подписи других сообщений представляется бесполезным занятием. Схема подписи обладает следующими свойствами.

1. Если процесс p исправен, то только процесс p может эффективно вычислить $S_p(M)$. Это вычисление называется *подписью* сообщения M .

2. Каждый процесс может эффективно проверить (для заданного имени процесса p , сообщения M и дополнительной информации S), верно ли, что $S = S_p(M)$. Эта верификация называется *аутентификацией* сообщения M . Схемы подписи могут быть с *секретными* и *открытыми ключами*. Наше первое допущение не исключает возможности того, что византийские процессы могут сговориться и открыть друг другу свои секретные ключи, чтобы один византийский процесс умел подделывать подпись любого другого византийского процесса. Но мы будем считать, что исправные процессы сохраняют свои секретные ключи в тайне.

В § 15.2.2–15.2.5 мы изучим методы реализации схем цифровой подписи. В следующей главе мы будем использовать запись $\langle \text{msg} \rangle : p$ для обозначения пары, состоящей из сообщения $\langle \text{msg} \rangle$ и подписи $S_p(\langle \text{msg} \rangle)$, которую поставил процесс p .

15.2.1. Протоколы, обладающие высокой устойчивостью

В работе Долева и Стронга [71] был предложен эффективный протокол широковещательного распространения информации, устойчивый относительно неисправностей византийского типа, который завершается за $t + 1$ тактов и в котором совершается полиномиальное число обменов сообщениями. Аутентификация, используемая в этом протоколе, делает его неограниченно устойчивым. И все же мы обращаем внимание на то, что неисправными могут быть не более N процессов (из общего числа N процессов системы) и если неисправными окажутся N процессов, то все требования будут выполнены сами собой. Поэтому мы будем

считать, что $t < N$. В основу протокола Долева и Стронга был положен более ранний протокол, предложенный Лампортом, Шостаком и Писом в работе [126]. В этом протоколе совершается экспоненциальное число обменов сообщениями, и вначале мы рассмотрим именно его.

В первом такте генерал передает всем процессам сообщение $\langle \text{value}, x_g \rangle : : g$, содержащее подписанное входное значение.

Начиная со второго такта и оканчивая $(t + 1)$ -м тактом процессы подписывают и переправляют сообщения, которые были получены в предыдущих тактах; поэтому сообщения, которыми обмениваются процессы в i -м такте, содержат i подписей. Процесс p считает сообщение $\langle \text{value}, v \rangle : : g : p_2 : \dots : p_i$ *достоверным*, если выполнены следующие требования.

1. Все i подписей поставлены правильно.
2. Все i подписей поставлены i разными процессами.
3. Подпись процесса p отсутствует в списке подписей этого сообщения.

По ходу работы алгоритма процесс p учитывает значения, полученные этим процессом в достоверных сообщениях, занося их в множество W_p . Вначале это множество пусто и значения, содержащиеся в достоверных сообщениях, поочередно добавляются в W_p .

Сообщения, которые переправляются в i -м такте, — это в точности все те сообщения, которые были получены в предыдущем такте. В конце $(t + 1)$ -го такта процесс p принимает решение на основе множества W_p . Если множество W_p состоит только из одного элемента $\{v\}$, то p принимает в качестве решения значение v , а в противном случае процесс p выбирает по умолчанию в качестве решения некоторое фиксированное значение (например, 0). Чтобы сократить число сообщений, процесс p ретранслирует сообщение $\langle \text{value}, v \rangle : : g : p_2 : \dots : p_i : p$ только тем процессам, которые отсутствуют в списке процессов $g, p_2, \dots, p_i$, подписавших это сообщение. Такая модификация не оказывает влияния на поведение рассматриваемого алгоритма, поскольку процессы, входящие в состав указанного списка, все равно сочтут ретранслированное сообщение недостоверным.

Теорема 15.10. *Алгоритм Лампорта, Шостака и Пирса является корректным алгоритмом широковещательного распространения информации, который устойчив относительно любого числа t неисправностей византийского типа, где $t < N$, и завершает работу за $t + 1$ тактов.*

Д о к а з а т е л ь с т в о. Все процессы принимают решение в $(t + 1)$ -м такте, и поэтому алгоритм обладает как свойством завершаемости, так и свойством одновременности.

Если генерал исправен и имеет на входе значение v , то все процессы в первом такте получают его сообщение $\langle \text{value}, v \rangle : : g$, и, следовательно, все исправные процессы включают значение v в множество W . Ни одно другое значение не будет включено в множество W , так как ни одно другое значение не будет подписано генералом. Таким образом, в $(t + 1)$ -м такте у всех исправных процессов будет одно и то же множество $W = \{v\}$, и все они примут в качестве решения значение v . Этим и объясняется свойство зависимости.

Чтобы проверить свойство согласованности, мы покажем, что для любой пары исправных процессов p и q в конце $(t+1)$ -го такта будет выполняться равенство $W_p = W_q$. Предположим, что в конце $(t+1)$ -го такта имеет место включение $v \in W_p$, и пусть известно, что процесс p поместил значение v в множество W_p в i -м такте после получения сообщения $\langle \text{value}, v \rangle :: g : p_2 : \dots : p_i$.

Случай 1. Если имя процесса q содержится в списке $g, p_2, \dots, p_i$, то процесс q уже видел значение v и поместил его в множество W_q .

Случай 2. Если имя процесса q не содержится в списке $g, p_2, \dots, p_i$, и $i \leq t$, то процесс p переправляет сообщение $\langle \text{value}, v \rangle :: g : p_2 : \dots : p_i : p$ процессу q в $(i+1)$ -м такте, и поэтому процесс q ознакомится со значением v самое позднее в $(i+1)$ -м такте.

Случай 3. Если q не содержится в списке $g, p_2, \dots, p_i$, и $i = t+1$, то следует заметить, что данное сообщение, полученное процессом p , было последовательно подписано $t+1$ процессами, среди которых есть хотя бы один исправный процесс. Этот процесс должен был переправить сообщение всем другим процессам, включая процесс q , и это означает, что процесс q уже ознакомился со значением v .

Так как в конце $(t+1)$ -го такта выполняется равенство $W_p = W_q$, процессы p и q принимают одинаковое решение. $\square$

Мы не можем остановить работу алгоритма до конца $(t+1)$ -го такта. Вплоть до t -го такта, исправный процесс может получать сообщения, которые были созданы и ретранслированы только неисправными процессами и при этом не отправлены другим исправным процессам; в результате исправные процессы могут принять противоречивые решения.

Промежуточный результат, установленный нами для предыдущего алгоритма, а именно согласованность на множестве значений, которая достигается всеми исправными процессами, выражает гораздо более сильное условие, чем требование согласованности по одному значению. На это обратили внимание Долев и Стронг, предложившие в своей работе [71] более эффективный алгоритм. На самом деле достаточно показать, что в конце $(t+1)$ -го такта либо а) для каждого исправного процесса p множество W_p состоит из *одного и того же* единственного элемента, либо б) ни для одного исправного процесса p множество W_p не является одноэлементным. В первом случае все процессы принимают в качестве решения значение v , а во втором случае они принимают в качестве решения число 0 (или, если возникает желание модифицировать алгоритм они приходят к решению «генерал неисправен»).

Это более слабое ограничение, которое налагается на множества W , соблюдается в алгоритме Долева и Стронга. Вместо того чтобы ретранслировать каждое достоверное сообщение, процесс p проводит рассылку не более двух сообщений; в одном из этих сообщений содержится первое из значений, полученных процессом p , а во втором — другое значение, которое получил процесс p . Построение полного описания этого алгоритма оставляется читателям в качестве упражнения для самостоятельного решения.

Теорема 15.11. *Алгоритм Долева и Стронга, краткое описание которого было приведено выше, — это протокол широковещательного пространства информации, который является устойчивым относительно неисправностей византийского типа и требует для своего выполнения $t+1$ тактов и не более $2N^2$ обменов сообщениями.*

Доказательство. Свойства завершаемости и одновременности устанавливаются так же, как и для предыдущего протокола: каждый исправный процесс принимает какое-то решение по окончании $(t+1)$ -го такта. И обоснование свойства зависимости проводится так же, как для предыдущего протокола. Если процесс g проводит в первом такте правильную рассылку значения v , то все исправные процессы принимают в этом такте значение v и больше не принимают никаких других значений. Следовательно, все исправные процессы принимают в качестве решения значение v . Указанная в формулировке теоремы сложность по числу обменов сообщениями является просто отражением того факта, что каждый исправный процесс проводит рассылку не более двух сообщений.

Чтобы убедиться в том, что соблюдается свойство согласованности, мы покажем, что для любых двух исправных процессов p и q множества W_p и W_q в конце $(t+1)$ -го такта удовлетворяют следующим соотношениям:

- 1) если $W_p = \{v\}$, то $v \in W_q$;
- 2) если $\#W_p > 1$, то $\#W_q > 1$.

Соотношение 1. Предположим, что процесс p принял значение v в i -м такте после получения сообщения $\langle \text{value}, v \rangle :: g : p_2 : \dots : p_i$, и проведем те же рассуждения, что и при доказательстве теоремы 15.10:

Случай 1. Если имя процесса q встречается в списке имен $g, \dots, p_i$, то процесс q уже принял значение v .

Случай 2. Если имя q не встречается в списке имен $g, \dots, p_i$ и $i \leq t$, то процесс p переправляет полученное значение процессу q , который в этом случае примет значение v .

Случай 3. Если имя процесса q не встречается в указанном списке и $i = t+1$, то хотя бы один из процессов, подписавших полученное сообщение (обозначим его символом r) является исправным. Тогда процесс r должен был уже переправить значение v процессу q , и поэтому v уже содержится в W_q .

Соотношение 2. Предположим, что по окончании работы алгоритма выполняется неравенство $\#W_p > 1$, и пусть w — это второе значение, которое было принято процессом p . И вновь, проводя рассуждения, аналогичные тем, которые применялись для доказательства предыдущей теоремы, мы можем показать, что $w \in W_q$, откуда будет следовать неравенство $\#W_q > 1$. (Равенство множеств W_p и W_q не может быть установлено, поскольку процесс p не будет отправлять третье, а также все последующие принятые им значения.)

После того как мы доказали выполнимость соотношений 1 и 2, предположим, что исправный процесс p принял в качестве решения значение $v \in W_p$ (это возможно, если $W_p = \{v\}$). Тогда согласно соотношению 1, значение v содержится в множестве W_q для каждого исправного процесса q . Как следует из соотношения 2, множество W_q не может содержать больше элементов, чем их содержится

в одноэлементном множестве $W_p = \{v\}$. Поэтому каждый исправный процесс q также принимает в качестве решения значение v .

Предположим теперь, что исправный процесс p принял решение по умолчанию, поскольку множество W_p не является одноэлементным. Если множество W_p пусто, то согласно соотношению 1 у каждого исправного процесса q множество W_q оказывается пустым, а если $\#W_p > 1$, то согласно соотношению 2 для каждого исправного процесса q будет выполняться неравенство $\#W_q > 1$. Следовательно, процесс q также примет решение по умолчанию. $\square$

Долев и Стронг сумели еще более усовершенствовать свой алгоритм и построить такой протокол широкоэмитерного распространения информации, который, будучи устойчивым относительно неисправностей византийского типа, завершает свою работу за то же самое число тактов, но при этом совершает всего лишь $O(Nt)$ обменов сообщениями.

15.2.2. Как устроены схемы цифровой подписи

Ввиду того что подпись $S_p(M)$, которую ставит процесс p , должна служить достаточным свидетельством того, что именно процесс p является автором отправленного сообщения, эта подпись должна представлять собой такую форму информации, которая

- 1) может быть эффективно вычислена процессом p (поставить подпись просто);
- 2) не может быть эффективно вычислена никаким другим процессом, кроме процесса p (подделать подпись трудно).

Здесь необходимо оговорить сразу, что для большинства схем цифровой подписи, используемых в настоящее время, второе требование не удастся обосновать в полной мере, установив, что задача подделки подписи является экспоненциально трудной. Обычно удается показать, что задача подделки подписи так или иначе связана с некоторой вычислительной проблемой (иногда задача подделки подписи оказывается эквивалентной этой вычислительной проблеме), которую математики изучали уже долгое время, но так и не смогли выяснить, является ли она разрешимой за полиномиальное время. Например, научившись подделывать цифровые подписи в схеме Фиата—Шамира, мы могли бы легко разлагать на множители большие числа; поскольку проблема факторизации является (предположительно) вычислительно трудной, задача подделки подписи тоже должна быть вычислительно трудной.

В основу существующих схем цифровой подписи были положены различные задачи, которые принято считать вычислительно трудными; к их числу относятся задачи вычисления дискретного логарифма, задача разложения натуральных чисел на множители, задача о рюкзаке и др. Требования 1) и 2) подразумевают, что процесс p должен иметь какое-то вычислительное «преимущество» по сравнению с другими процессами; это преимущество воплощается в некоторой секретной информации, которой обладает процесс p и которая называется *секретным ключом* процесса p . Поэтому вычисление подписи $S_p(M)$ можно про-

вести эффективно только тогда, когда известен секретный ключ, но без знания этой информации вычисление цифровой подписи становится (предположительно) трудной задачей. Значит, если процессу p удастся сохранить свой ключ в секрете, то это будет означать, что только процесс p сможет за приемлемое время вычислить подпись $S_p(M)$.

Все процессы должны уметь верифицировать подписи, т. е. должны уметь для любого сообщения M и подписи S эффективно проверять, действительно ли подпись S была вычислена для сообщения M при помощи секретного ключа процесса p . Для проведения такой проверки необходимо открыть информацию, касающуюся секретного ключа процесса p ; эту информацию принято называть *открытым ключом* процесса p . Открытый ключ должен позволить провести верификацию подписи, но при этом даже с учетом знания открытого ключа задачи вычисления секретного ключа и подделки подписи должны по-прежнему оставаться вычислительно трудными.

Наиболее успешные схемы цифровой подписи, известные в настоящее время, основаны на теоретико-числовых вычислениях в арифметических кольцах по модулю больших чисел. Основные арифметические операции (сложения, умножения, возведения в степень) в этих кольцах могут быть выполнены за время, полиномиальное относительно длины этих модулей (измеряемой в битах). Деление возможно, если делитель и модуль являются взаимно простыми числами (т. е. не имеют ни одного общего простого делителя); в этом случае деление тоже можно выполнить за полиномиальное время. Так как для постановки и верификации подписи нужно проводить вычисления на сообщениях, каждое сообщение M истолковывается как большое натуральное число.

15.2.3. Схема цифровой подписи Эль-Гамала

В основу схемы цифровой подписи Эль-Гамала [77] положена специальная теоретико-числовая функция, которая называется *дискретным логарифмом*. Для заданного простого числа P мультипликативная группа вычетов по модулю P , которую мы будем обозначать $mulgP$, содержит $P - 1$ элементов и является *циклической*. Это означает, что можно выбрать такой элемент $g \in mulgP$, что все $P - 1$ чисел

$$g^0 = 1, g^1, g^2, \dots, g^{P-3}, g^{P-2}$$

попарно различны и, следовательно, перечисляют все элементы множества $mulgP$. Такой элемент g называется *порождающим элементом* группы $mulgP$ или, по-другому, *первообразным корнем* по модулю P . Порождающий элемент не является уникальным; обычно таких элементов бывает много. Если задана группа вычетов P и порождающий элемент g , то для каждого $x \in \mathbb{Z}_P^*$ существует *единственное* такое натуральное число i , не превосходящее $P - 1$, для которого имеет место равенство $g^i = x$ (отношение равенства здесь ограничено элементами группы $mulgP$). Это число i называется *дискретным логарифмом* (или иногда *индексом*) элемента x . В отличие от основных арифметических операций, которые были упомянуты выше, вычисление дискретного логарифма — это

непростая задача. Ее изучали долгое время, и до сих пор для нее не удалось обнаружить общего эффективного решения; однако не удалось доказать и того, что эта задача не имеет эффективного решения. Обзор наиболее важных результатов по вопросу о вычислении дискретного логарифма можно найти в работе [154].

Схема цифровой подписи Эль-Гамала [77] основывается на трудности вычисления дискретных логарифмов. Всем процессам известно большое простое число P и первообразный корень g группы вычетов $\text{mulg}P$. В качестве своего секретного ключа процесс p выбирает случайным образом число d , лежащее в промежутке между 1 и $P - 2$, а в качестве открытого ключа процесса p используется число $e = g^d$. Заметим, что d — это дискретный логарифм числа e . Процесс p , знающий дискретный логарифм числа e , может эффективно вычислить подпись и тем самым предоставить неявное доказательство того, что автор подписи осведомлен о числе d .

Достоверной подписью сообщения M служит пара (r, s) , удовлетворяющая равенству $g^M = e^r r^s$. Процесс p , владеющий секретным ключом d , может легко отыскать такую пару. Процесс p выбирает случайным образом число a , которое является взаимно простым с числом $P - 1$, и вычисляет

$$r = g^a \bmod P$$

и

$$s = (M - dr)a^{-1} \bmod (P - 1).$$

Эти числа и в самом деле удовлетворяют требуемому равенству:

$$e^r \cdot r^s = e^r (g^a)^{(M-dr)a^{-1}} = g^{dr} g^{M-dr} = g^M.$$

(Отношение равенства здесь ограничено элементами группы $\text{mulg}P$.) В достоверности подписи $S = (r, s)$ для сообщения M легко убедиться, проверив выполнимость равенства $g^M = e^r r^s$.

Алгоритмы для вычисления дискретного логарифма. Поскольку число d , используемое в качестве секретного ключа процесса p , равно дискретному логарифму открытого ключа e , для того чтобы взломать эту схему подписи, достаточно найти способ эффективного вычисления дискретного логарифма по модулю числа P . В настоящее время неизвестно ни одного эффективного алгоритма, который мог бы вычислять дискретный логарифм в общем случае, и ни одного другого способа эффективной подделки подписей.

Общий алгоритм вычисления дискретного логарифма описан в работе Од-лыжко [154]. Его сложность оценивается величиной, имеющей такой же порядок роста, как и оценка наилучших из известных алгоритмов факторизации столь же больших чисел, что и число P . Этот алгоритм вначале вычисляет несколько таблиц, используя для этого только числа P и g , а затем на втором этапе вычисляет логарифмы заданных чисел. Если Q — это наибольший простой делитель числа $P - 1$, то время выполнения вычислений на первом этапе, а также размер полученных таблиц оценивается величиной порядка Q . Поэтому число P желательно выбирать так, чтобы число $P - 1$ имело очень большой простой делитель.

Вычисление логарифмов на втором этапе можно проводить всего за несколько секунд даже на совсем простых компьютерах. Поэтому числа P и g необходимо менять достаточно часто (лучше всего ежемесячно), для того чтобы вспомогательные таблицы, предназначенные для конкретного числа P , становились бесполезными еще до того, как их построение будет завершено.

Рандомизованная подпись. Благодаря рандомизации, которая используется в алгоритме Эль-Гамала, каждая из $\phi(P - 1)$ различных подписей¹⁾ для заданного сообщения может быть получена на выходе процедуры формирования подписи. Таким образом, подписывая один и тот же документ дважды, мы почти наверняка получим две разные достоверные подписи. Рандомизации отведена важная роль в процедуре формирования подписи: если процесс p подписывает два сообщения, используя одно и то же значение a , то секретный ключ процесса p можно вычислить на основании этих двух подписей (см. упражнение 15.6).

15.2.4. Схема цифровой подписи RSA

Если n — это большое число, представляющее собой произведение двух простых чисел P и Q , то вычислять квадратные корни, а также корни более высокой степени по модулю n , не зная разложения n на множители, непросто. Умение вычислять квадратные корни может быть использовано для поиска делителей числа n (см. упражнение 15.7), и это означает, что вычисление квадратных корней — столь же трудная задача, как и задача факторизации чисел.

В схеме цифровой подписи, предложенной Ривестом, Шамиром и Адлеманом в работе [163], в качестве открытого ключа процесса p используется большое число n , делители которого известны процессу p , а также некоторая экспонента e . Для подписи сообщения M процесс p вычисляет корень e -й степени числа M по модулю n , и эту подпись легко проверить при помощи операции возведения в степень. Для вычисления этого корня высокой степени процесс p также применяет операцию возведения в степень. Когда процесс p вырабатывает свой секретный ключ, он вычисляет число d , удовлетворяющее равенству $de = 1 \bmod \phi(n)$. В связи с этим имеет место равенство $(M^d)^e = M$, и это означает, что число M^d является корнем e -ой степени числа M . В качестве секретного ключа процесс p использует только число d , т. е. он не обязан хранить в памяти разложение числа n на множители.

В схеме цифровой подписи RSA процесс p подтверждает свое авторство способностью вычислять корни по модулю n , а это, в свою очередь, предполагает (неявно), что процессу p известно разложение числа n на множители. Предполагается, что только процесс p обладает этим знанием. В рассматриваемой схеме каждый процесс использует разные модули.

¹⁾Функция ϕ известна как *функция Эйлера*; значение функции $\phi(n)$ равно числу элементов в группе $\text{mulg}n$. — Прим. автора.

15.2.5. Схема цифровой подписи Фиата—Шамира

В схеме цифровой подписи Фиата—Шамира (см. [78]) трудности, связанные с вычислением квадратных корней, используются еще более изощренно. В схеме цифровой подписи RSA процесс подтверждает свое авторство, демонстрируя способность вычислять корни по модулю своего открытого ключа, а эта способность вычислять корни подразумевает, что процессу известно разложение открытого ключа на множители. В схеме цифровой подписи Фиата—Шамира все процессы используют *одно и то же* число n , разложение которого на множители известно только некоторому надежному правомочному центру, которому доверяют все процессы. Процессу p вручаются квадратные корни некоторых особых чисел (зависящих от отличительных признаков процесса p), и подпись, которой процесс p сопровождает сообщение M , призвана свидетельствовать о том, что автору подписи известны эти квадратные корни, но сами корни при этом не открываются.

Преимущество схемы Фиата—Шамира по сравнению со схемой RSA состоит в том, что эта схема имеет меньшую арифметическую сложность и в ней отсутствуют отдельные открытые ключи для каждого процесса. Недостатком же является необходимость пользоваться услугами вышестоящего правомочного центра, который вырабатывает секретные ключи. Как уже было упомянуто ранее, в схеме задействовано большое число n , представляющее собой произведение двух больших простых чисел, известных только указанному центру. Кроме того, имеется односторонняя псевдослучайная функция f , отображающая символьные строки в множество $\mathbb{Z}_n$; считается, что эта функция известна всем процессам и они могут ее вычислять, но вычислить эффективно обратное преобразование невозможно.

Секретные и открытые ключи. В качестве секретных ключей процессу p вручаются квадратные корни $s_1, \dots, s_k$ некоторого набора из k чисел, вычисленные по модулю n ; иными словами, $s_j = \sqrt{v_j^{-1}}$, где $v_j = f(p, j)$. Числа v_j следует рассматривать как открытые ключи процесса p , но, коль скоро их можно вычислить по отличительным признакам процесса p , эти ключи не нужно хранить в памяти. Чтобы не утомлять читателей излишними техническими подробностями, мы будем считать, что все эти k чисел являются квадратичными вычетами по модулю n . Квадратные корни этих чисел могут быть вычислены в доверительном центре, которому известно разложение числа n на множители.

Подписание сообщений: первая попытка. Подпись, которую ставит процесс p , призвана служить неявным доказательством того, что автору подписи известны корни всех чисел v_j^{-1} , т. е. для любого числа v_j он может предъявить такое число s , которое удовлетворяет равенству $s^2 v_j = 1$. Процесс p мог бы просто предъявить число s_j , но, передав число s_j , он тем самым открыл бы свой секретный ключ; чтобы избежать раскрытия секретного ключа, в рассматриваемой схеме подписи было решено воспользоваться следующей идеей. Процесс p выбирает случайным образом число r и вычисляет $x = r^2$. Теперь p — это единственный процесс, который может предъявить такое число y , которое удо-

влетворяет равенству $y^2 v_j = x$: для этого достаточно вычислить $y = r s_j$. Таким образом, процесс p может продемонстрировать, что ему известно число s_j , не раскрывая при этом самого числа, а предъявляя вместо этого пару (x, y) , удовлетворяющую равенству $y^2 v_j = x$. Так как процесс p не показывает числа r , извлечь из этой пары число s_j невозможно без вычисления квадратного корня.

Но если мы будем применять указанные пары в качестве подписей, нам придется столкнуться с двумя проблемами. Во-первых, можно сконструировать подобную пару обманным путем: *вначале* можно выбрать число y , а затем вычислить число $x = y^2 v$. Во-вторых, наша подпись не зависит от сообщения, и поэтому всякий процесс, получивший сообщение, подписанное процессом p , может вставить копию этой подписи в подделанное сообщение. Самое главное, что требуется от рассматриваемой схемы подписи, — это заставить процесс p продемонстрировать его способность извлекать квадратный корень из *произведения некоторого набора чисел* v_j , имея при этом в виду, что выделенный набор чисел зависит от подписываемого сообщения и какого-то случайного числа. Если «пропустить» сообщение и это случайное число через одностороннюю функцию f , то тем самым будет устранена возможность подделки путем преждевременного выбора числа y .

Чтобы подписать сообщение M , процесс p поступает так.

1. Процесс p выбирает случайным образом число r и вычисляет $x = r^2$.
2. Затем процесс p вычисляет значение функции $f(M, x)$ и выделяет в этом значении первые k битов $e_1, \dots, e_k$.
3. После этого процесс p вычисляет $y = r \prod_{e_j=1} s_j$. Подпись $S_p(M)$ представляет собой набор чисел $(e_1, \dots, e_k, y)$.

Чтобы провести верификацию подписи $(e_1, \dots, e_k, y)$, которой процесс p заверил сообщение M , нужно поступать так.

1. Вычислить v_j и $z = y^2 \prod_{e_j=1} v_j$.
2. Вычислить $f(M, z)$ и убедиться в том, что первые k битов результата — это числа $e_1, \dots, e_k$.

Если проверяется неподдельная подпись, то значение z , вычисленное на первом шаге верификации, будет равно тому значению x , которое было сформировано при подписании сообщения, и поэтому первые k битов значения функции $f(M, z)$ должны совпадать с числами $e_1, \dots, e_k$.

Подделка и окончательное решение. А теперь мы рассмотрим ту стратегию действий, которую нужно применять для того, чтобы подделать подпись, полученную по заданной схеме, не имея чисел s_j .

1. Выбрать случайным образом k битов $e_1, \dots, e_k$.
2. Выбрать случайным образом число y и вычислить $x = y^2 \prod_{e_j=1} v_j$.
3. Вычислить $f(M, x)$ и проверить, совпадают ли первые k битов полученного значения с ранее выбранными битами $e_1, \dots, e_k$. Если это так, то $(e_1, \dots, e_k, y)$ — это и есть подделанная подпись сообщения M .

Если исходить из предположения о том, что вероятность совпадения вычисленных битов на шаге 3 составляет величину 2^{-k} , то для успешной подделки подписи потребуется совершить в среднем 2^k попыток.

Если полагать $k = 72$ и считать, что проверку одного варианта выбора битов e_j можно осуществить за 10^{-9} секунд, то в среднем для успешной подделки подписи (руководствуясь предложенной стратегией) потребуется $2^{72} \cdot 10^{-9}$ секунд (а это составляет 1,5 миллионов лет), и это означает, что схема подписи является очень надежной. Однако каждый процесс должен хранить в памяти k корней, и если k должно быть ограничено из-за небольшого объема памяти, то ожидаемое время успешной подделки 2^k может оказаться уже недостаточно большим. Сейчас мы покажем, как нужно видоизменить схему подписи, чтобы, используя k корней, увеличить среднее время успешной подделки подписи до величины 2^{kt} для заранее выбранного числа t . Новый замысел состоит в том, чтобы воспользоваться первыми kt битами значения функции f , определить t подмножеств чисел v_j и заставить процесс p продемонстрировать свое знание t различных произведений этих чисел. Чтобы подписать сообщение M , процесс p должен поступать следующим образом.

1. Процесс p выбирает случайным образом t чисел $r_1, \dots, r_t$ и вычисляет их квадраты $x_i = r_i^2$.
2. Далее процесс p вычисляет значение односторонней функции $f(M, x_1, \dots, x_t)$ и определяет первые kt битов этого значения e_{ij} ($1 \leq i \leq t$ и $1 \leq j \leq k$).
3. И, наконец, процесс p вычисляет $y_i = r_i \prod_{e_{ij}=1} s_j$ для $1 \leq i \leq t$. Подпись $S_p(M)$ представляет собой набор чисел $(e_{11}, \dots, e_{tk}, y_1, \dots, y_t)$.
Чтобы верифицировать подпись $(e_{11}, \dots, e_{tk}, y_1, \dots, y_t)$, которой процесс p заверил сообщение M , нужно поступать так.
 1. Вычислить числа v_j , а также $z_i = y_i^2 \prod_{e_{ij}=1} v_j$.
 2. Вычислить значение функции $f(M, z_1, \dots, z_t)$ и проверить, что первые kt битов этого значения совпадают с числами $e_{11}, \dots, e_{tk}$.

Злоумышленник, стремящийся подделать правильную подпись, руководствуясь описанной выше стратегией, может достичь успеха на третьем шаге с вероятностью 2^{-kt} , и поэтому в среднем ему потребуется совершить 2^{kt} попыток. Фиат и Шамир показали в своей статье, что до тех пор, пока никто не научится быстро разлагать число n на множители, нельзя предложить никакого существенно лучшего алгоритма подделки подписи. Поэтому предложенную ими схему цифровой подписи можно сделать сколь угодно надежной, выбирая достаточно большие числа k и t .

15.2.6. Подведение итогов

В первых двух параграфах этой главы мы показали, что в синхронных системах можно построить детерминированные решения задачи широковещательного распространения информации при наличии неисправностей византийского типа. Если не используются средства аутентификации, то наибольший уровень устойчивости таких решений составляет величину $t < N/3$ (см. § 15.4.1), а если разрешается использовать методы аутентификации сообщений, то построенные протоколы имеют неограниченную стойкость (см. настоящий параграф). В изученных нами решениях задачи широковещательной рассылки синхронность моделирова-

лась в рамках весьма жестких допущений импульсной модели; отказоустойчивая реализация импульсной модели обсуждается в § 15.4.3.

Задача синхронизации цепи стрелков. Помимо использования импульсной модели, в основу построенных протоколов было положено еще одно допущение, которое гласит, что всем процессам известен тот такт, с которого начинается широковещательная трансляция (для удобства обозначений этому такту присвоен номер 1). Если это не так, то возникает проблема одновременного запуска алгоритма, после того как один или несколько процессов (спонтанно) обращаются с просьбой выполнить алгоритм широковещательного оповещения. Запрос может поступить от генерала (после того как он вычислил результат, который должен быть объявлен всем процессам) или от лейтенантов (осознавших, что им всем нужна некоторая информация, которой владеет генерал). Эта проблема известна в математической литературе под названием *задачи синхронизации цепи стрелков*. По условию этой задачи один или более процессов *отдают команду*, не обязательно в одном и том же такте, и все процессы должны *сделать залп*. Формальные требования таковы.

1. **Дисциплинированность.** Ни один исправный процесс не стреляет, пока не была отдана команда.
2. **Одновременность.** Если какой-то исправный процесс произвел выстрел, то и все исправные процессы стреляют в том же такте.
3. **Завершаемость.** Если исправный процесс отдал команду, то все исправные процессы стреляют спустя конечное число тактов.

И действительно, если есть решение задачи синхронизации цепи стрелков, то отпадает необходимость заранее оговаривать первый такт широковещательной трансляции сообщений: процессы, заинтересованные в этой трансляции, отдают команду, запуская алгоритм синхронизации цепи стрелков, и широковещательное распространение сообщений начинается с того такта, в котором процессы дают залп. Методы решения задачи широковещательного распространения информации в условиях возникновения неисправностей византийского типа и задачи синхронизации цепи стрелков можно объединить и получить за счет этого более эффективные протоколы, которые могут решать задачу широковещательной трансляции сообщений без учета предварительного соглашения и времени начала трансляции.

Сложность по времени и протоколы с преждевременной остановкой. В этой главе мы уже изучили протоколы, которые завершают работу за $t + 1$ или за $2t + 3$ тактов (коммуникационных туров). В работе Фишера и Линча [80] было установлено, что $t + 1$ — это оптимальное число коммуникационных туров для t -робастных протоколов консенсуса, и этот результат был распространен на протоколы с аутентификацией Долевым и Стронгом в работе [71].

Наиболее тонкая особенность этих доказательств заключается в том, что в предложенных сценариях процесс должен выходить из строя в каждом такте начиная с первого такта и оканчивая t -м тактом, и поэтому полученные оценки соответствуют наихудшему случаю по числу действительных неисправностей,

произошедших по ходу выполнения системы. Но следует учесть, что в большинстве выполнений число вышедших из строя процессов гораздо меньше уровня устойчивости алгоритма. Поэтому была исследована возможность построения таких протоколов, которые были бы способны достичь согласованности раньше предельного срока в тех выполнениях, в которых из строя выходит лишь небольшое число процессов. Протоколы широковещательного распространения информации, обладающие этим свойством, называются протоколами с *преждевременной остановкой*. Долев, Решук и Стронг в работе [70] показали, что любому протоколу потребуется провести не менее $f + 2$ туров в произвольном выполнении, в котором из строя выходят f процессов. Обсуждение некоторых протоколов с преждевременной остановкой для задач консенсуса и широковещательной трансляции можно найти в работе [28].

Протоколы с преждевременной остановкой принимают решение спустя считанное число тактов, после того как исправные процессы убеждаются в том, что в одном из тактов не возникло никаких новых неисправностей. Однако невозможно гарантировать, что все исправные процессы придут к этому выводу в одном и том же такте. (Конечно, если речь не идет о $(t + 1)$ -м такте, в котором принимают решение все процессы: коль скоро выйти из строя могут не более t процессов, среди первых $t + 1$ тактов есть только один такт, в котором не возникает новых неисправностей.) Вследствие этого протоколы с преждевременной остановкой не удовлетворяют условию одновременности. Как было показано в работе Коана и Дворка [54], чтобы соблюсти условие одновременности, выполнения, в которых не выходит из строя ни одного процесса, должны продолжаться не менее $t + 1$ тактов даже в том случае, когда речь идет о рандомизованных протоколах в рамках очень слабой модели неисправностей. Отсюда следует, что протоколам с аутентификацией тоже требуется $t + 1$ тактов, для того чтобы все процессы одновременно пришли к согласию.

Проблемы разрешения и интерактивная непротиворечивость. Если разрешить воспользоваться протоколом широковещательного распространения информации в качестве стандартной процедуры, то фактически для всех проблем разрешения можно построить подходящий синхронный алгоритм при условии достижения *интерактивной непротиворечивости*, т. е. согласованности по множеству входных данных. В задаче интерактивной непротиворечивости процессы принимают в качестве решения *вектор* входных данных, в котором каждому процессу системы выделено по одному элементу. Формальные требования таковы.

1. **Завершаемость.** Каждый исправный процесс p принимает в качестве решения вектор V_p , в котором каждому процессу выделено по одному элементу.
2. **Согласованность.** Векторы решений всех исправных процессов одинаковы.
3. **Зависимость.** Если процесс q исправен, то для всякого исправного процесса p выполняется соотношение $V_p[q] = x_q$.

Интерактивной непротиворечивости можно добиться путем многократной широковещательной трансляции: каждый процесс оповещает все другие процессы

о своих входных данных, и при этом процесс p записывает свое решение во время широковещательного распространения информации процессом q в ячейку вектора $V_p[q]$. Свойства завершаемости, согласованности и зависимости будут унаследованы от соответствующих свойств используемого алгоритма широковещательной трансляции.

Так как каждый исправный процесс вычисляет один и тот же вектор (свойство согласованности), для большинства проблем разрешения можно построить алгоритм, воспользовавшись детерминированной функцией, определенной на векторе решений (и это немедленно гарантирует согласованность). Например, консенсуса можно достичь, выбрав наиболее распространенное значение в векторе решений. Избрание лидера можно провести, выбрав наименьший отличительный признак в векторе (при этом нужно помнить о том, что выбранный процесс может оказаться неисправным).

15.3. Синхронизация часов

В предыдущих параграфах мы убедились в том, что если ограничиться рассмотрением только детерминированных алгоритмов, то устойчивость синхронных систем превосходит устойчивость асинхронных систем. Это было установлено для идеализированной модели синхронных систем, в которых работа процессов разбита на такты. Более высокий уровень устойчивости импульсной модели приводит нас к выводу о том, что не существует детерминированного робастного способа синхронизации вполне асинхронных сетей. В этом параграфе мы покажем, что робастная реализация импульсной модели возможна в модели асинхронных сетей с ограниченной задержкой (ABD-сетей²⁾).

Характерная особенность ABD-модели состоит в том, что процессам открыт доступ к локальным часам и задержка передачи сообщений ограничена сверху. При описании и анализе алгоритмов мы будем широко использовать *шкалу реального времени*, позволяющую пометать каждое событие вещественным числом $t \in \mathbb{R}$, обозначающим время осуществления этого события. Согласно постулатам релятивистской физики не существует единого стандарта или наиболее предпочтительного способа расстановки таких пометок; мы условимся, что в дальнейшем для этой цели нам подойдет какой-нибудь физически осмысленный способ хронометрирования. Процессы системы не видят шкалу реального времени, но им предоставлена опосредованная возможность следить за временем при помощи *часов*, показания которых привязаны к реальному времени. Обозначим символом C_p часы, которыми снабжен процесс p ; показания этих часов можно считывать и переустанавливать (последнее необходимо для синхронизации часов). Если часы не переустанавливают, то с течением времени их показания непрерывно изменяются; запись $C_p(t) = T$ будет использоваться для обозначения того, что в момент реального времени t показания часов равны T .

Заглавные буквы (C, T) будут использоваться для обозначения показаний часов, а прописные буквы (c, t) — для обозначения моментов реального времени.

²⁾Аббревиатура термина «asynchronous bounded-delay networks» — Прим. перев.

Часы можно использовать для управления событиями, например посредством оператора

when $C_p = T$ **then** отправить сообщение,

выполнение которого приводит к тому, что заданное сообщение будет отправлено в момент реального времени $C_p^{-1}(T)$. Функцию³⁾ C_p^{-1} условимся обозначать символом c_p .

Показания совершенных часов увеличиваются на величину Δ каждые Δ единиц времени, т. е. они удовлетворяют соотношению $C(t + \Delta) = C(T = t) + \Delta$. Совершенные часы, будучи однажды синхронизованными, не будут снова нуждаться в корректировке, но, к сожалению, это всего лишь удобная математическая абстракция. Часы, которые используются в распределенных системах, подвержены отклонению, которое ограничено некоторой малой константой ρ , известной заранее (обычно это величина порядка 10^{-5} или 10^{-6}). Говорят, что отклонение часов C ρ -ограничено, если для любой пары последовательных моментов реального времени t_1 и t_2 , между которыми часы C не подвергались переустановке, выполняется соотношение

$$(t_2 - t_1)(1 + \rho)^{-1} \leq C(t_2) - C(t_1) \leq (t_2 - t_1)(1 + \rho). \quad (15.1)$$

В распределенной системе разные часы не обязаны давать одинаковые показания в каждый момент реального времени, т. е. равенство $C_p(t) = C_q(t)$ не всегда соблюдается. Часы называются δ -синхронизованными в момент реального времени t , если выполняется неравенство $|C_p(t) - C_q(t)| \leq \delta$; эти часы называются δ -синхронизованными при показаниях T , если верно неравенство $|c_p(T) - c_q(T)| \leq \delta$. Мы будем считать эти два понятия эквивалентными (см. упражнение 15.8). Цель работы алгоритмов синхронизации состоит в том, чтобы добиться и суметь поддерживать глобальную δ -синхронизованность, т. е. δ -синхронизованность между каждой парой часов. Параметр δ называется *точностью* синхронизации.

Задержка передачи сообщений ограничена снизу величиной δ_{min} и ограничена сверху величиной δ_{max} , причем $0 \leq \delta_{min} < \delta_{max}$. Если говорить формально, то это означает, что для сообщения, отправленного в момент реального времени σ и доставленного в момент реального времени τ , выполняется неравенство

$$\delta_{min} \leq \tau - \sigma \leq \delta_{max}. \quad (15.2)$$

С учетом того, что мы были вольны в выборе шкалы реального времени, допущения (15.1) и (15.2) касаются как самой шкалы реального времени, так и часов в распределенной системе.

15.3.1. Как снимать показания с удаленных часов

В этом параграфе мы займемся изучением вопроса о том, с какой степенью точности процесс p может скорректировать показания своих совершенных ча-

³⁾Обратное отображение возможно при условии, что часы не переустанавливаются. — Прим. перев.

сов по показаниям совершенных часов некоторого надежного сервера s . Если использовать детерминированный протокол, то наилучшая достижимая точность будет оцениваться величиной $\frac{1}{2}(\delta_{max} - \delta_{min})$, и она может быть достигнута при помощи простого протокола с обменом только одним сообщением. Вероятностные протоколы позволяют достичь произвольно высокой точности, но сложность по числу обменов сообщениями зависит от желаемой точности и распределения длительности задержек сообщений.

Теорема 15.12. *Существует детерминированный протокол синхронизации часов C_p и C_s с точностью $\frac{1}{2}(\delta_{max} - \delta_{min})$, в котором происходит обмен одним сообщением. Никакой детерминированный протокол не позволяет достичь лучшей точности.*

Доказательство. Вначале мы опишем простой протокол и покажем, что он позволяет достичь той точности, которая указана в формулировке теоремы. Чтобы провести синхронизацию часов C_p , сервер отправляет одно сообщение $\langle \text{time}, C_s \rangle$. Когда процесс p получает сообщение $\langle \text{time}, T \rangle$, он корректирует показания своих часов, устанавливая на них время $T + \frac{1}{2}(\delta_{max} + \delta_{min})$.

Чтобы убедиться в том, что при этом достигается требуемая точность, обозначим буквами σ и τ соответственно те моменты реального времени, когда было отправлено и принято сообщение $\langle \text{time}, T \rangle$. Тогда $T = C_s(\sigma)$. Поскольку мы имеем дело с совершенными часами, имеет место равенство $C_s(\tau) = T + (\tau - \sigma)$. В момент времени τ процесс p исправляет показания своих часов, и поэтому будут верны равенства $C_p(\tau) = T + \frac{1}{2}(\delta_{max} + \delta_{min})$ и $C_s(\tau) - C_p(\tau) = (\tau - \sigma) - \frac{1}{2}(\delta_{max} + \delta_{min})$. В результате из соотношения $\delta_{min} \leq \tau - \sigma \leq \delta_{max}$ следует, что $|C_s(\tau) - C_p(\tau)| \leq \frac{1}{2}(\delta_{max} - \delta_{min})$.

Чтобы узнать, какой будет нижняя оценка точности синхронизации, рассмотрим произвольный детерминированный протокол. В этом протоколе процессы p и s обмениваются некоторыми сообщениями, и после этого процесс p корректирует показания своих часов. Интерес представляют два сценария поведения протокола, изображенные на рис. 15.5. В первом сценарии часы показывают одно и то же время до начала выполнения протокола, все сообщения от процесса s к процессу p доходят за время δ_{min} , а все сообщения от процесса p к процессу s доходят за время δ_{max} . Если процесс p исправит показания своих часов на величину Δ_1 , то после проведенной синхронизации часы процесса p будут опережать часы C_s в точности на величину Δ_1 .

Во втором сценарии до начала выполнения протокола часы C_p отстают от часов C_s на величину $\delta_{max} - \delta_{min}$, все сообщения от процесса p к процессу s доходят за время δ_{min} , а все сообщения от процесса s к процессу p доходят за время δ_{max} . Если процесс p исправит показания своих часов на величину Δ_2 , то мы обнаружим, что после проведенной синхронизации часы процесса p отстают от часов C_s в точности на величину $\delta_{max} - \delta_{min} - \Delta_2$.

Однако ни p , ни s не могут заметить разницу между этими двумя сценариями из-за неопределенного времени задержки сообщений. Вот поэтому $\Delta_1 = \Delta_2$.

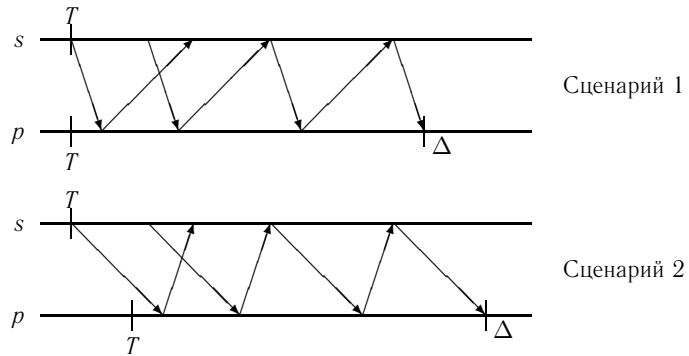


Рис. 15.5. Сценарии работы детерминированного протокола

Отсюда следует, что в худшем случае точность синхронизации будет не меньше величины

$$\min_{\Delta} \max(|\Delta|, |\delta_{\max} - \delta_{\min} - \Delta|).$$

Этот минимум равен $\frac{1}{2}(\delta_{\max} - \delta_{\min})$ (и достигается при $\Delta = \frac{1}{2}(\delta_{\max} - \delta_{\min})$). $\square$

Если оба процесса p и q проводят синхронизацию своих часов с указанной точностью с привлечением сервера, то будет достигнута глобальная $(\delta_{\max} - \delta_{\min})$ -синхронизация; для большинства приложений этого вполне достаточно.

Еще более хорошая точность может быть достигнута, если воспользоваться вероятностным синхронизационным алгоритмом, который был предложен Кристианом в работе [58]. В этом протоколе предполагается, что задержка сообщений — это случайная величина, распределение которой задается функцией $F : [\delta_{\min}, \delta_{\max}] \rightarrow [0, 1]$, причем эта функция может быть неизвестна участникам протокола. Вероятность того, что сообщение будет доставлено за время x , равна $F(x)$, и задержки разных сообщений — это независимые величины. Произвольно высокая точность может быть достигнута только при том условии, что нижняя оценка $\delta_{\min}$ является точной, т. е. для любого $x > \delta_{\min}$ выполняется неравенство $F(x) > 0$.

Протокол прост: процесс p просит процесс s сообщить время, и s немедленно отправляет в ответ сообщение $\langle \text{time}, T \rangle$. Процесс p измеряет время I между отправлением запроса и получением отклика; очевидно, справедливо неравенство $I \geq 2\delta_{\min}$. Задержка доставки ответного сообщения не меньше $\delta_{\min}$ и не больше $I - \delta_{\min}$ и, следовательно, отличается от величины $\frac{1}{2}I$ не более чем на $\frac{1}{2}(I - 2\delta_{\min})$ от $\frac{1}{2}I$. Поэтому процесс p может установить на своих часах время $T + \frac{1}{2}I$ и при этом точность синхронизации будет составлять величину $\frac{1}{2}(I - 2\delta_{\min})$. Если желаемая точность, которую мы хотим достичь, задается величиной ϵ , то в том случае, если $\frac{1}{2}(I - 2\delta_{\min}) > \epsilon$, процесс p отправляет новый запрос, а в противном случае он завершает выполнение протокола.

Лемма 15.13. Вероятностный протокол синхронизации часов достигает точности ϵ , и при этом математическое ожидание числа обменов сообщениями не превосходит $F(\delta_{\min} + \epsilon)^{-2}$.

Доказательство. Вероятность того, что запрос, отправленный процессом p , достигнет адресата за время $\delta_{\min} + \epsilon$, равна $F(\delta_{\min} + \epsilon)$, такова же вероятность того, что отклик будет доставлен процессу p за время $\delta_{\min} + \epsilon$. Следовательно, вероятность того, что процесс p получит ответ на свой запрос за время $2\delta_{\min} + 2\epsilon$, не меньше $F(\delta_{\min} + \epsilon)^2$, и отсюда следует оценка $F(\delta_{\min} + \epsilon)^{-2}$ среднего числа попыток, которые придется провести, прежде чем произойдет успешный обмен сообщениями. $\square$

Сложность этого протокола по времени можно уменьшить, если процесс p будет отправлять новый запрос, как только убедится в том, что за время $2\delta_{\min} + 2\epsilon$ отклик на предыдущий запрос не был доставлен. Ожидаемое время синхронизации, равное $2(\delta_{\min} + \epsilon) F(\delta_{\min} + \epsilon)^{-2}$, в этом случае не будет зависеть от ожидаемой или максимальной задержки передачи сообщений, и протокол при этом будет робастным относительно потери сообщений. (Чтобы отличать устаревшие отклики, нужно использовать нумерацию запросов.)

15.3.2. Синхронизация часов в распределенной системе

В этом параграфе мы познакомимся с распределенным алгоритмом синхронизации часов, который был описан в Махани и Шнейдером в работе [137]. Этот протокол является t -робастным относительно неисправностей византийского типа для $t < N/3$. Долев, Халперн и Стронг в работе [73] установили, что без использования аутентификации устойчивую синхронизацию нельзя провести, если $t \geq N/3$.

Ядро алгоритма синхронизации составляет протокол, при помощи которого достигается *неточная согласованность* показаний часов в среднем. Процессы корректируют показания своих часов и достигают высокой степени синхронизации. Из-за того что часы подвержены отклонению, достигнутая точность синхронизации постепенно размывается, и спустя какое-то время возникает необходимость провести новый тур синхронизации. Предположим, что в момент реального времени t_0 все часы δ_0 -синхронизованы. Тогда все часы будут оставаться $(\delta_0 + 2\rho R)$ -синхронизованными вплоть до момента времени $t_0 + R$. Таким образом, если мы хотим, чтобы часы в распределенной системе были синхронизованы с точностью δ и один тур синхронизации позволяет достичь точности синхронизации δ_0 , то каждый раз очередной тур синхронизации нужно проводить спустя $(\delta - \delta_0)/2\rho$ единиц времени. Поскольку время S , которое требуется для проведения одного тура синхронизации, обычно очень мало по сравнению с периодом времени R , вполне разумно считать, что по ходу синхронизации отклонением часов можно пренебречь, т. е. мы будем считать, что часы являются совершенными.

Неточная согласованность: быстро сходящийся алгоритм. Для синхронизации часов Махани и Шнейдер в работе [137] воспользовались задачей о до-

стижении неточной согласованности. Предположим, что каждый процесс p имеет некоторое вещественное число x_p и при этом для любых двух исправных процессов p и q имеет место неравенство $|x_p - x_q| \leq \delta$. В качестве выходных данных процесс p выдает вещественное число y_p , и точность вычисления выходных данных определяется выражением $\max_{p,q} |y_p - y_q|$. Задача состоит в том, чтобы построить алгоритм, обеспечивающий наивысшую точность вычисления выходных данных.

Перейдем к рассмотрению быстро сходящегося алгоритма (алгоритма 15.6), предложенного Махани и Шнейдером. Определим на конечном подмножестве вещественных чисел $A \subset \mathbb{R}$ две функции $intvl(A) = [\min(A), \max(A)]$ и $width(A) = \max(A) - \min(A)$. Работа алгоритма разбита на два этапа: этап сбора данных и этап вычисления. На первом этапе процесс p обращается к каждому процессу системы с просьбой прислать ему входные данные (рассылая широкоэвещательное сообщение $\langle ask \rangle$) и ожидает откликов в течение $2\delta_{max}$ единиц времени. По истечении этого срока процесс p будет располагать входными данными всех исправных процессов, а также некоторыми ответами, которые ему могут выслать неисправные процессы. К этим ответам процесс p добавляет значения ∞ для тех процессов, которые не откликнулись на его просьбу.

Затем каждый процесс пропускает полученные значения через некий фильтр, который отбирает только те из них, которые соответствуют входным данным исправных процессов, а также те ответы от неисправных процессов, которые достаточно близки к правильным значениям. Так как правильные значения отличаются друг от друга не более чем на δ , среди полученных ответов имеется не менее $N - t$ правильных входных данных, которые отличаются от каждого правильного значения не более чем на δ . Эти значения образуют множества A_p .

Окончательный результат представляет собой среднее арифметическое всех значений с учетом того, что все отвергнутые значения замещаются некоторой оценкой, которая вычисляется путем применения детерминированной оценивающей функции $estimator$ к уцелевшим данным. В качестве оценивающей функции может быть выбрана произвольная функция, удовлетворяющая условию $estimator(A) \in intvl(A)$; например, в роли функции $estimator$ может выступать функция минимума, максимума, среднего арифметического, или $\frac{1}{2}[\max(A) + \min(A)]$.

Теорема 15.14. *Быстро сходящийся алгоритм достигает точности синхронизации $2t\delta/N$.*

Доказательство. Пусть v_{pr} — это значение, которое включается в мультимножество V_p в качестве вклада со стороны процесса r , после того как процесс p прекращает прием сообщений, т. е. v_{pr} равно либо x_r , либо ∞ . Пусть a_{pr} — это значение, которое включается в мультимножество A_p в качестве вклада со стороны процесса r , после того как p вычисляет y_p , т. е. a_{pr} равно либо v_{pr} , либо $esti_p$. Чтобы получить верхнюю оценку точности синхронизации, разобьем суммирование, которое проводится для вычисления результата, на две части, а именно, суммирование по тем значениям, которые были получены от исправных процессов (C), и суммирование по тем значениям, которые соответствуют

вкладу неисправных процессов (B). Для исправных процессов p и q абсолютная разность $|a_{pr} - a_{qr}|$ ограничена 0, если $r \in C$, и величиной 2δ , если $r \in B$.

Первая оценка получается такой, потому что если p и r — исправные процессы, то $a_{pr} = x_r$. Действительно, ввиду того что процесс r вовремя посылает в ответ процессу p сообщение $\langle ask \rangle$, будет верно равенство $v_{pr} = x_r$. Аналогичное равенство $v_{pr'} = x_{r'}$ будет справедливо для всех исправных процессов r' , и, как следует из описания алгоритма, значения, полученные от процессов r , сохраняются после прохождения через фильтр процесса p . Поэтому будет выполняться равенство $a_{pr} = v_{pr}$.

Вторая оценка получается такой, потому что для исправных процессов p и q выполняется неравенство $width(A_p \cup A_q) \leq 2\delta$, когда p и q вычисляют свои решения. Так как добавленные оценки лежат между допустимыми значениями, достаточно рассмотреть максимальную разность значений a_p и a_q , которые прошли через фильтры процессов p и q . Существует не менее $N - t$ процессов r , для которых $|v_{pr} - a_p| \leq \delta$, и не менее $N - t$ процессов r , для которых $|v_{qr} - a_q| \leq \delta$. Отсюда следует, что существует такой исправный процесс r , для которого выполняются оба неравенства $|v_{pr} - a_p| \leq \delta$ и $|v_{qr} - a_q| \leq \delta$. Но поскольку процесс r исправный, верно равенство $v_{pr} = v_{qr}$, и поэтому $|a_p - a_q| \leq 2\delta$.

Таким образом, для исправных процессов p и q мы имеем

$$\begin{aligned} |y_p - y_q| &= \left| \left(\sum_{r \in C} A_p \right) / N - \left(\sum_{r \in C} A_q \right) / N \right| = \\ &= \frac{1}{N} \cdot \left| \left(\sum_{r \in C} a_{pr} + \sum_{r \in B} a_{pr} \right) - \left(\sum_{r \in C} a_{qr} + \sum_{r \in B} a_{qr} \right) \right| = \\ &= \frac{1}{N} \cdot \left| \left(\sum_{r \in C} a_{pr} - \sum_{r \in C} a_{qr} \right) + \left(\sum_{r \in B} a_{pr} - \sum_{r \in B} a_{qr} \right) \right| \leq \\ &\leq \frac{1}{N} \cdot \left[\left(\sum_{r \in C} |a_{pr} - a_{qr}| \right) + \left(\sum_{r \in B} |a_{pr} - a_{qr}| \right) \right] \leq \\ &\leq \frac{1}{N} \cdot \left[(0) + \left(\sum_{r \in B} 2\delta \right) \right] \leq 2t\delta/N. \quad \square \end{aligned}$$

Чтобы получить произвольно высокую точность синхронизации, алгоритм нужно применять повторно: если провести i итераций, то точность достигает величины $(\frac{2}{3})^i \delta$. Эта точность будет даже еще лучше, если доля неисправных процессов будет меньше одной трети от общего числа процессов системы, так как при выводе оценки точности синхронизации t можно истолковывать как число процессов, которые действительно вышли из строя. Точность синхронизации, которую гарантирует этот алгоритм в худшем случае, нельзя повысить за счет подходящего выбора функции $estimator$. И в самом деле, византийский процесс r может навязать процессу p любое значение $a_{pr} \in intvl(A_p)$, просто отправив его процессу p в ответ на его запрос. Удачный выбор функции $estimator$ позволяет достичь хорошей точности в среднем, если что-то известно о наиболее вероятном поведении неисправных процессов.

```

var  $x_p, y_p, esti_p$  : real;      (* Вход, выход, оценщик  $V$  *)
 $V_p, A_p$  : мультимножество чисел типа real;

```

```

begin(* Этап сбора входных данных *)
   $V_p := \emptyset$  ;
  forall  $q \in \mathbb{P}$  do send  $\langle \text{ask} \rangle$  to  $q$  ;
  wait  $2\delta_{max}$  ; (* Обработать сообщения  $\langle \text{ask} \rangle$  и  $\langle \text{val}, x \rangle$  *)
  while  $\#V_p < N$  do insert( $V_p, \infty$ ) ;
  (* Теперь вычислить допустимые значения *)
   $A_p := \{x \in V_p : \#\{y \in V_p : |y - x| \leq \delta\} \geq N - t\}$  ;
   $esti_p := estimator(A_p)$  ;
  while  $\#A_p < N$  do insert( $A_p, esti_p$ ) ;
   $y_p := (\sum A_p)/N$ 
end

```

После получения сообщения $\langle \text{ask} \rangle$ от q :

```

  send  $\langle \text{val}, x_p \rangle$  to  $q$ 

```

После получения сообщения $\langle \text{val}, x \rangle$ от q :

```

  if такого сообщения от  $q$  раньше не поступало
  then insert( $V_p, x$ )

```

Алгоритм 15.6. Быстросходящийся алгоритм

Синхронизация часов. Чтобы синхронизовать часы, применяется быстро сходящийся алгоритм, позволяющий достичь неточной согласованности относительно нового показания часов. Предполагается, что вначале все часы δ -синхронизованы. Имеющийся в нашем распоряжении алгоритм нуждается в некоторой адаптации, поскольку

- 1) неизвестно, какова задержка сообщений, и поэтому процесс не знает, как долго он может ожидать значений от исправных процессов;
- 2) выполнение алгоритма требует определенного времени, и поэтому показания часов — это не постоянные, а возрастающие величины.

Чтобы компенсировать неизвестную задержку, процесс добавляет величину $\frac{1}{2}(\delta_{max} + \delta_{min})$ к полученным показаниям часов (как в детерминированном протоколе, который анализируется в теореме 15.12), и к окончательному значению точности синхронизации добавляется слагаемое $\delta_{max} - \delta_{min}$. Чтобы полученное значение было представлено как показание часов, а не постоянная величина, процесс p хранит в памяти величину Δ_{pr} , которая представляет собой разность между полученным показанием часов (плюс $\frac{1}{2}(\delta_{max} + \delta_{min})$) и показанием его собственных часов. В момент времени t процесс p вычисляет аппроксимацию $C_p(t) + \Delta_{pr}$ показания часов процесса r . В результате мы получаем модифицированный алгоритм 15.7.

Следует обратить внимание на то, что в алгоритме 15.7 фильтр имеет более широкую пропускную способность, нежели в алгоритме 15.6, а именно, он

```

var  $C_p, \Delta_p, esti_p$  : real;      (* Часы, адаптация и оценщик для  $V$  *)
 $D_p, A_p$  : мультимножества чисел типа real;

```

```

begin (* Этап сбора входных данных *)
   $D_p := \emptyset$  ;
  forall  $q \in \mathbb{P}$  do send  $\langle \text{ask} \rangle$  to  $q$  ;
  wait  $2\delta_{max}$  ; (* Обработка сообщений  $\langle \text{ask} \rangle$  и  $\langle \text{val}, x \rangle$  *)
  while  $\#D_p < N$  do insert( $D_p, \infty$ ) ;
  (* Теперь вычислить допустимые значения *)
   $A_p := \{x \in D_p : \#\{y \in D_p : |y - x| \leq \delta + (\delta_{max} - \delta_{min})\} \geq N - t\}$  ;
   $esti_p := estimator(A_p)$  ;
  while  $\#A_p < N$  do insert( $A_p, esti_p$ ) ;
   $\Delta_p := (\sum A_p)/N$  ;
   $C_p := C_p + \Delta_p$ 
end

```

После получения сообщения $\langle \text{ask} \rangle$ от q :

```

  send  $\langle \text{val}, C_p \rangle$  to  $q$ 

```

После получения сообщения $\langle \text{val}, C \rangle$ от q :

```

  if такого сообщения от  $q$  раньше не поступало
  then insert( $D_p, (C + \frac{1}{2}(\delta_{max} + \delta_{min})) - C_p$ )

```

Алгоритм 15.7. Быстрая согласованность часов

допускает отклонения, не превосходящие $\delta + (\delta_{max} - \delta_{min})$, тогда как раньше допустимое отклонение не превышало δ . Более широкая полоса захвата призвана компенсировать неизвестную задержку передачи сообщений, и в приведенном ниже утверждении объясняются причины, по которым был выбран именно такой порог. Обозначим символом d_{pr} ту величину, которую процесс p заносит в мультимножество D_p в качестве вклада со стороны процесса r после первого этапа работы процесса p (сравните эту величину со значением v_{pr} из предыдущего алгоритма).

Предложение 15.15. Если процессы p, q и r исправны, то, после того как процесс p объявляет тайм-аут, будет выполняться неравенство $|d_{pr} - d_{qr}| \leq \delta + (\delta_{max} - \delta_{min})$.

Доказательство. Передача сообщения $\langle \text{val}, C \rangle$ процессу p от процесса q может расцениваться как реализация детерминированного алгоритма считывания показаний часов, описанного в теореме 15.12. Когда процесс p получает это сообщение, абсолютная разность $|C_q - [C + \frac{1}{2}(\delta_{max} + \delta_{min})]|$ ограничена величиной $\frac{1}{2}(\delta_{max} - \delta_{min})$, поэтому d_{pq} отличается от разности $C_q - C_p$ не более чем на $\frac{1}{2}(\delta_{max} + \delta_{min})$. Аналогично, d_{pr} отличается от $C_r - C_p$ не более чем на

$\frac{1}{2}(\delta_{\max} + \delta_{\min})$. С учетом того, что C_q и C_r отличаются друг от друга не более чем на δ , в результате мы получаем искомое неравенство. $\square$

Теорема 15.16. *После выполнения алгоритма 15.7 часы будут синхронизованы с точностью*

$$(\delta_{\max} - \delta_{\min}) + \frac{2t}{N}[\delta + (\delta_{\max} - \delta_{\min})].$$

Доказательство. В приведенном ниже доказательстве мы будем использовать запись C_p для обозначения неоткорректированных часов, а запись C'_p для обозначения откорректированных часов. Это означает, что $C'_p(t) = C_p(t) + \Delta_p$. Чтобы оценить точность откорректированных часов, рассмотрим некоторый момент реального времени t , раньше которого прекращается прием сообщений (объявляется тайм-аут) всеми процессами системы, и положим $w_{pr} = C_p + d_{pr}$. Из доказательства приведенного выше утверждения видно, что для исправных процессов p, q и r выполняется неравенство $|w_{pr} - C_r(t)| \leq \frac{1}{2}(\delta_{\max} - \delta_{\min})$, из которого следует, что $|w_{pr} - w_{qr}| \leq (\delta_{\max} - \delta_{\min})$. Если процесс r неисправен, то разность $|w_{pr} - w_{qr}|$ будет ограничена величиной $2\delta + 3(\delta_{\max} - \delta_{\min})$; доказательство этого соотношения проводится по той же схеме, что и обоснование соответствующего шага в доказательстве теоремы 15.14.

Наконец, если принять во внимание неравенство $C'_p(t) = C_p(t) + \Delta_p = (\sum_r w_{pr})/N$, окончательная оценка точности получается так же, как и в доказательстве теоремы 15.14, т. е. путем рассмотрения по отдельности того вклада, который вносят исправные и неисправные процессы. $\square$

Из описания алгоритма ясно видно, что все сообщения $\langle \text{val}, C \rangle$ отправляются с показаниями *неоткорректированных* часов; это происходит потому, что коррекция часов откладывается, до тех пор пока не будут отправлены отклики на все запросы $\langle \text{ask} \rangle$.

15.3.3. Реализация раундовой модели

Для реализации импульсной модели могут подходить и системы с более слабыми допущениями синхронизации, которые подчиняются следующим требованиям:

- 1) имеется верхняя оценка $\delta_{\max}$ задержки передачи сообщений;
- 2) имеется верхняя оценка γ времени, необходимого для локального вычисления в каждом такте (это время, которое необходимо для перехода процесса из одного локального состояния в другое и для отправления всех сообщений);
- 3) процессы располагают часами, которые δ -синхронизованы в каждый момент времени T (по показаниям этих часов) и подвержены ρ -отклонению.

Для моделирования каждого такта необходимо время $(1 + \rho)(\delta_{\max} + \delta + \gamma)$ (по показаниям процессных часов).

Алгоритм моделирования очень прост. Предположим, что выполнение моделирующего алгоритма начинается в тот момент, когда процессные часы показы-

вают время 0 (моделирование можно начать и в более поздний момент времени, добавив к показаниям часов одну и ту же фиксированную величину). Когда процессные часы показывают время 0, процесс отправляет те сообщения, которые относятся к первому такту. Когда процессные часы показывают время $i(1 + \rho)(\delta_{\max} + \delta + \gamma)$, процесс осуществляет тот переход, который он должен совершить в i -м такте, и отправляет сообщения, относящиеся к $(i + 1)$ -му такту. Поскольку показания часов могут достигать сколь угодно большой величины, каждый исправный процесс, следуя указанной стратегии, может выполнить сколь угодно много тактов.

Остается показать, что исправный процесс p получает все сообщения, которые были ему отправлены (исправными процессами) в i -м такте, до того как приступит к изменению своего состояния в этом такте. Предположим, что процесс q отправляет сообщение процессу p в i -м такте. Процесс q приступил к выполнению перехода, относящегося к предыдущему такту, и отправлению сообщений, относящихся к i -му такту, когда часы этого процесса показывают время $(i - 1)(1 + \rho)(\delta_{\max} + \delta + \gamma)$, т. е. это происходит в момент реального времени $c_q[(i - 1)(1 + \rho)(\delta_{\max} + \delta + \gamma)]$. То допущение, которое касается оценки времени локальной обработки информации, позволяет заключить, что сообщение, относящееся к i -му такту, будет отправлено самое позднее в момент реального времени

$$c_q[(i - 1)(1 + \rho)(\delta_{\max} + \delta + \gamma)] + \gamma,$$

и отсюда (с учетом оценки задержки передачи сообщений) следует, что это сообщение будет получено самое позднее в момент реального времени

$$c_q[(i - 1)(1 + \rho)(\delta_{\max} + \delta + \gamma)] + \gamma + \delta_{\max}.$$

Процесс p приступает к переходу в новое состояние, когда его часы будут показывать время $i(1 + \rho)(\delta_{\max} + \delta + \gamma)$, и это соответствует моменту реального времени

$$c_p[i(1 + \rho)(\delta_{\max} + \delta + \gamma)].$$

Так как часы процессов p и q должны быть δ -синхронизованы в момент времени $(i - 1)(1 + \rho)(\delta_{\max} + \delta + \gamma)$ по показаниям процессных часов, справедливо неравенство

$$c_q[(i - 1)(1 + \rho)(\delta_{\max} + \delta + \gamma)] \leq c_p[(i - 1)(1 + \rho)(\delta_{\max} + \delta + \gamma)].$$

Поскольку часы процесса p подвержены ρ -отклонению, верно неравенство

$$c_p[i(1 + \rho)(\delta_{\max} + \delta + \gamma)] \geq c_p[(i - 1)(1 + \rho)(\delta_{\max} + \delta + \gamma)] + (\delta_{\max} + \delta + \gamma).$$

Из приведенных здесь неравенств мы получаем неравенство

$$c_p[i(1 + \rho)(\delta_{\max} + \delta + \gamma)] \geq c_q[(i - 1)(1 + \rho)(\delta_{\max} + \delta + \gamma)] + \gamma + \delta_{\max},$$

из которого следует, что процесс p получает сообщение еще до того, как он приступит к переходу в другое состояние.

15.4. Упражнения к главе 15

15.4.1

Упражнение 15.1. Сколькими сообщениями обмениваются процессы в протоколе

$Broadcast(N, t)$ (алгоритм 15.2 и 15.3)?

Упражнение 15.2. Какое наибольшее число сообщений отправляют исправные процессы в алгоритме 15.4 в тех выполнениях, которые приводят к решению 0? Рассмотрите случаи, когда генерал исправен и когда генерал неисправен.

15.4.2

Упражнение 15.3. Сколькими сообщениями обмениваются процессы в алгоритме Лампорта, Шостака и Писа широковещательного распространения информации, описанного в параграфе 15.2.1?

Упражнение 15.4. Приведите пример выполнения протокола Долева и Стронга, описанного в параграфе 15.2.1, в котором исправные процессы p и q завершают работу с результатом $W_p \neq W_q$.

Упражнение 15.5. Покажите, что переименование, сохраняющее порядок на отрезке от 1 до N , можно осуществить, как только будет достигнута интерактивная согласованность.

Упражнение 15.6. Предположим, что процесс p подписывает два сообщения M_1 и M_2 , применяя схему цифровой подписи Эль-Гамала (см. § 15.2.3) и используя одно и то же значение a . Покажите, каким образом, располагая этими двумя подписанными сообщениями, можно открыть секретный ключ процесса P .

Упражнение 15.7. Предположим, что число n является произведением двух больших простых чисел. Предположим также, что мы располагаем процедурой («черным ящиком»), которая позволяет вычислять квадратные корни, т.е. эта процедура для заданного квадратичного вычета y вычисляет такое число x , что $x^2 = y$ (равенство по модулю n). Покажите, как можно применить этот «черный ящик» для разложения числа n на множители.

15.4.3

Упражнение 15.8. Покажите, что если часы C_p и C_q подвержены ρ -ограниченному отклонению и являются δ -синхронизованными в момент реального времени t , то они будут $\delta(1 + \rho)$ -синхронизованными в момент процессного времени $T = C_p(t)$.

Упражнение 15.9. Вероятностный протокол синхронизации часов эффективен, если задержка большинства сообщений близка к величине δ_{min} (т.е. даже при малых значениях ϵ значение $F(\delta_{min} + \epsilon)$ уходит от 0 быстро). Постройте «двойственный» протокол, который работает эффективно, если задержка большинства сообщений близка к величине δ_{max} (т.е. даже при малых значениях ϵ значение $F(\delta_{min} + \epsilon)$ отстоит достаточно далеко от 1). Покажите, что ожидаемая сложность по числу обменов сообщениями ограничена величиной $2[1 - F(\delta_{max} - \epsilon)]^{-2}$, и выясните, какова будет оценка ожидаемого времени выполнения протокола.

Упражнение 15.10. Верно ли, что для любого исправного процесса p множество A_p , фигурирующее в алгоритме 15.6, удовлетворяет неравенству $width(A_p) \leq \delta$?

ГЛАВА 16

ОБНАРУЖЕНИЕ НЕИСПРАВНОСТЕЙ

Решить задачу консенсуса для асинхронных систем невозможно (см. § 14.6.1), и поэтому исследователи вынуждены были упростить условия задачи и обратиться к более ограниченным моделям. В упрощенных вариантах задачи допускается рандомизация и возникают более слабые формы координации; в моделях вводится синхронизация. Другой способ усиления вычислительной модели, получивший широкое признание, связан с введением детекторов неисправностей.

Изучение синхронных моделей обусловлено практическими потребностями, поскольку те или иные разновидности часов и таймеров присутствуют в большинстве систем распределенного программирования. Теоретические исследования позволяют понять, для каких целей и в какой мере необходимо использовать эти примитивы. Точно так же обстоят дела и с детекторами неисправностей: очень часто при попытке связаться с вышедшим из строя процессом система динамической поддержки вычислений будет выдавать сообщение об ошибке. Однако на эти сообщения об ошибке нельзя полагаться безусловно. Поэтому было бы неплохо выяснить, насколько достоверны должны быть эти сообщения, чтобы с их помощью можно было решить проблему консенсуса (или какие-то другие задачи).

В отличие от аппарата синхронизации (реализованного на основе физических часов), детекторы неисправностей не имеют столь же простых и ясных принципов реализации. Это означает, что нужно найти какое-то нетривиальное решение для реализации детекторов неисправностей либо как составной части системы динамической поддержки вычислений, либо как отдельного модуля в приложениях. В большинстве реализаций распределенных систем принято рассчитывать на использование таймеров (см. § 16.5.4), и поэтому обращение к детекторам неисправностей вызывает ряд критических замечаний. Возражая против использования детекторов неисправностей, говорят, что сами по себе они не могут решить никаких задач, поскольку их реализация потребует тех же самых вычислительных ресурсов (времени), которые можно было бы выделить для непосредственного решения задачи консенсуса.

В этих замечаниях есть определенный резон, однако следует заметить, что те же самые возражения приводились на протяжении последних пятидесяти лет всякий раз, когда в теории вычислений предлагалось что-то новое. Зачем нужны языки программирования высокого уровня, если программы все равно приходится транслировать в машинные коды, для того чтобы их выполнять? Мы могли бы «с таким же успехом» сразу написать программу на языке машинных команд! Зачем нужны драйверы для обращения к дискам и принтерам из прикладных программ,

если эти драйверы сами по себе являются такими же программами? Мы могли бы «с таким же успехом» вставить код драйвера в нашу прикладную программу и обойтись без всякого драйвера. Конечно же, мы знаем, как ответить на эти возражения, касающиеся языков программирования высокого уровня и драйверов: они упрощают программирование, улучшают понимаемость программ, повышают их совместимость, позволяют оценивать выразительные возможности определенных понятий. Те же самые доводы можно привести и в пользу детекторов неисправностей.

16.1. Модель и определения

Хотя обнаружение неисправностей уже давно практиковалось во многих системах, впервые изучение этого явления на абстрактном уровне было предпринято научно-исследовательской группой под руководством Сэма Туэга в 1991 г. в Корнельском университете. Пять лет спустя была опубликована основополагающая работа [42], из которой позаимствована значительная часть определений и результатов этой главы.

Детектор неисправностей — это специальный модуль, который снабжает каждый процесс системы сведениями о *подозрительных* процессах; логическое условие $j \in D$ выполняется, если процесс j относится к числу подозрительных в момент вычисления условия. В этой главе мы будем рассматривать единственный тип неисправностей — выход процессов из строя. Модули разных процессов не обязаны согласовывать свои подозрения друг с другом (т. е. исправный процесс p может подозревать процесс r , тогда как исправный процесс q может не иметь никаких подозрений), и эти подозрения не обязаны быть безошибочными (исправный процесс p может подозревать исправный процесс r или не подозревать неисправный процесс r). Естественно, чтобы иметь возможность рассуждать о программах с детекторами неисправностей, мы должны вначале описать свойства этих детекторов и, прежде всего, установить взаимосвязь между диагнозами, которые выдает детектор, и действительными неисправностями.

Произошедшие неисправности представляются в виде *шаблона неисправностей*, и мы приведем соответствующее определение этого понятия. Не желая связываться выражениями темпоральной логики, мы будем использовать в наших выражениях в явном виде переменную времени t , которая будет принимать значения из некоторого множества моментов времени T ; в качестве множества T может выступать, например, множество натуральных чисел. Мы просим не забывать о том, что процессы *не могут* наблюдать ни значения переменной t , ни шаблон неисправностей F , ни множеств $\text{Crash}(F)$ и $\text{Corr}(F)$, ни истории детектора неисправностей! Процесс q может «видеть» *только* значение $H(q, t)$, когда он получает доступ к детектору неисправностей в момент времени t .

16.1.1. Четыре основных класса детекторов

Выходы процессов из строя, происходящие при выполнении программы, моделируются посредством *шаблона неисправностей*, который представляет со-

бой функцию $F: T \rightarrow \mathcal{P}(\mathbb{P})$. Значение $F(t)$ следует истолковывать как множество процессов, которые уже вышли из строя к моменту времени t ; поскольку мы не предусматриваем перезапусков системы, из неравенства $t_1 \leq t_2$ следует соотношение $F(t_1) \subseteq F(t_2)$. Совокупность вышедших из строя и исправных процессов определяется при помощи шаблона неисправностей следующим образом: $\text{Crash}(F) = \bigcup_{t \in T} F(t)$ и $\text{Corr}(F) = \mathbb{P} \setminus \text{Crash}(F)$.

У разных процессов могут быть разные подозрения в зависимости от текущего момента времени. Эти подозрения моделируются при помощи функции $H: \mathbb{P} \times T \rightarrow \mathcal{P}(\mathbb{P})$. Значение $H(q, t)$ следует понимать как совокупность процессов, находящихся под подозрением у процесса q в момент времени t .

Чтобы детекторы неисправностей могли быть *недетерминированными* (т. е. проявляли разную реакцию на одну и ту же модель неисправностей), мы представляем детектор неисправностей $\mathcal{D}$ в виде отображения множества шаблонов неисправностей в *совокупность* историй детектора неисправностей. Таким образом, для заданного шаблона неисправностей F значением $H \in \mathcal{D}(F)$ является история детектора неисправностей.

Свойства историй. Само собой разумеется, что если мы хотим, чтобы детектор неисправностей был полезен, должно быть некоторое отношение между его выходными данными (историями детектора неисправностей) и его входными данными (шаблоном неисправностей). Обычно выдвигаются требования двух типов, а именно условия полноты (детектор будет подозревать процессы, вышедшие из строя) и точности (детектор не будет подозревать исправные процессы). Условие полноты ограничивает множество подозреваемых процессов снизу, а условие точности — сверху, но при этом не требуется, чтобы границы совпадали, поскольку это требование нельзя выполнить на практике ни для каких систем.

Мы рассмотрим условие полноты в следующей форме.

Определение 16.1. Детектор неисправностей $\mathcal{D}$ называется *полным*, если каждый вышедший из строя процесс рано или поздно будет заподозрен каждым исправным процессом:

$$\forall F: \forall H \in \mathcal{D}(F): \exists t: \forall p \in \text{Crash}(F): \forall q \in \text{Corr}(F): \forall t' \geq t: p \in H(q, t').$$

(Это условие часто называют условием сильной полноты.)

Мы рассмотрим четыре различных условия точности. Напоминаем, что условие точности предполагает, что исправные процессы находятся вне подозрений. Это требование, будучи представленным в сильной форме, означает, что *все* исправные процессы находятся вне подозрений, а будучи представленным в слабой форме, оно означает, что *хотя бы один* исправный процесс находится вне подозрений. Далее мы рассмотрим «растянутые во времени» варианты этих требований, когда указанные свойства не обязаны выполняться в начале вычисления, но обязательно должны выполняться по истечении некоторого времени.

Определение 16.2. Детектор неисправностей $\mathcal{D}$ называется *сильно точным*, если ни один исправный процесс никогда не подпадает под подозрение:

$$\forall F: \forall H \in \mathcal{D}(F): \forall t: \forall p, q \notin F(t): p \notin H(q, t).$$

Детектор неисправностей $\mathcal{D}$ называется *слабо точным*, если существует такой исправный процесс, который никогда не подпадает под подозрение:

$$\forall F: \forall H \in \mathcal{D}(F): \exists p \in \text{Corr}(F): \forall t: \forall q \notin F(t): p \notin H(q, t).$$

Детектор неисправностей $\mathcal{D}$ называется *сильно точным в конечном итоге*, если существует такой момент времени, после которого ни один исправный процесс никогда не подпадает под подозрение:

$$\forall F: \forall H \in \mathcal{D}(F): \exists t: \forall t' \geq t: \forall p, q \in \text{Corr}(F): p \notin H(q, t').$$

Детектор неисправностей $\mathcal{D}$ называется *сильно точным в конечном итоге*, если существует такой исправный процесс и момент времени, после которого этот процесс никогда не подпадает под подозрение:

$$\forall F: \forall H \in \mathcal{D}(F): \exists t: \exists p \in \text{Corr}(F): \forall t' \geq t: \forall q \in \text{Corr}(F): p \notin H(q, t').$$

Следует обратить внимание на то, что приведенные определения ограничивают круг подозрения только *исправных* процессов q ; подозрения процессов, вышедших из строя, нас не интересуют. Кроме того, как видно из определения сильной точности, нельзя начать подозревать вышедший из строя процесс *еще до того*, как этот процесс действительно вышел из строя. Такого исключения не требуется для трех других форм точности.

Очень просто построить полный детектор неисправностей (например, детектор, который подозревает всех: $H(q, t) = \mathbb{P}$), и так же просто построить точный детектор неисправностей (например, детектор, который не подозревает никого: $H(q, t) = \emptyset$). Но по-настоящему интересные и полезные детекторы всегда сочетают в себе качества полноты и точности, и мы рассмотрим четыре класса детекторов.

Определение 16.3. Детектор неисправностей называется *совершенным*, если он полон и сильно точен; класс совершенных детекторов обозначим буквой $\mathcal{P}$.

Детектор неисправностей называется *строгим*, если он полон и слабо точен; класс строгих детекторов обозначим буквой $\mathcal{S}$.

Детектор неисправностей назовем *в конечном итоге совершенным*, если он полон и сильно точен в конечном итоге; класс в конечном итоге совершенных детекторов обозначим $\diamond \mathcal{P}$.

Детектор неисправностей назовем *в конечном итоге строгим*, если он полон и сильно точен в конечном итоге; класс в конечном итоге строгих детекторов обозначим $\diamond \mathcal{S}$.

16.1.2. О том, как использовать детекторы неисправностей и подвохи

Не составляет большого труда представить себе, как можно применять детекторы неисправностей для проектирования распределенных алгоритмов.

Давайте припомним те коммуникационные операции, с которыми нам пришлось чаще всего сталкиваться при построении асинхронных распределенных алгоритмов в гл. 14:

1. Каждый узел выполняет *оповещение*, т. е. отправляет сообщение каждому узлу.

2. Каждый узел собирает $N - t$ сообщений.

Ни один процесс не дожидается поступления более чем $N - t$ сообщений, поскольку в случае выхода процессов из строя это может привести к блокировке. Такая конструкция никогда не приводит к вечной блокировке (см., например, лемму 14.30), так как по крайней мере $N - t$ процессов всегда остаются живыми. Подвох кроется в том, что даже при отсутствии каких-либо неисправностей множества полученных сообщений у разных процессов могут быть разными. Каждое множество собранных сообщений всегда имеет один и тот же размер $N - t$. Ожидать поступления сообщений от какого-нибудь *выделенного* процесса строго запрещается, потому что это может привести к блокировке в том случае, когда процесс-отправитель выходит из строя.

Стандартный способ коммуникации с использованием детекторов неисправностей протекает следующим образом.

1. Каждый узел отправляет сообщение каждому узлу.

2. Каждый узел ожидает до тех пор, пока либо для каждого процесса q не будет получено сообщение от q , либо процесс q не подпадет под подозрение.

Условие полноты детектора неисправностей гарантирует, что при выполнении обменов сообщениями по указанному сценарию не произойдет блокировки; действительно, каждый процесс, который не отправляет сообщение, из-за того что он вышел из строя, рано или поздно подпадет под подозрение. Так же как и в предыдущем случае, здесь есть подвох: разные сценарии поведения системы приводят к тому, что разные исправные процессы будут располагать разными множествами собранных сообщений.

1. Процесс q выходит из строя, но прежде, чем это случилось, он сумел отправить требуемое сообщение всем или некоторым процессам. Процесс p_1 получает это сообщение, прежде чем он начнет подозревать процесс q , а процесс p_2 поступает иначе и завершает этап приема сообщений, не дожидаясь поступления сообщений от процесса q .

2. Процесс q исправен и отправляет требуемое сообщение, но процесс p_1 считает процесс q подозрительным и не принимает от него сообщений. В то же время процесс p_2 полагает, что процесс q находится вне подозрений, и принимает от него сообщение.

Таким образом, множество собранных сообщений может включать сообщения, поступившие от процессов, которые затем вышли из строя, но при этом не содержать сообщений от некоторых исправных процессов. Размер множества со-

бранных сообщений может варьироваться, и даже в том случае, когда существует верхняя оценка числа неисправных процессов, это множество может содержать менее $N - t$ сообщений, ввиду того что некоторые подозрения были ошибочными. И только верхняя оценка числа процессов, которые могут находиться *под подозрением*, может ограничить снизу размер указанного множества сообщений; в основном это получается уже за счет использования слабо точных детекторов.

Детекторы неисправностей позволяют получать сообщения от выделенных процессов: ожидание поступления сообщения будет прервано возникновением подозрения по отношению к этому процессу. И вновь обращаем внимание читателей на то, что и в этом случае может случиться так, что исправный процесс не дожидется поступления сообщения от другого исправного процесса.

Описанный выше режим получения сообщений настолько распространен, что мы введем для его обозначения специальную форму записи: оператор «collect $\langle \text{message}, \text{par} \rangle$ from q ». Этот оператор ожидает, пока от процесса q не поступит сообщение $\langle \text{message}, \text{par} \rangle$ или пока q не будет признан подозреваемым процессом; при этом вырабатывается булево значение, которое является признаком успешного получения сообщения.

16.2. Решение задачи консенсуса при помощи слабо точных детекторов

В этом параграфе приводится простой алгоритм консенсуса, использующий детекторы неисправностей. Требования правильности, которые налагаются на этот алгоритм, таковы.

1. **Завершаемость.** Каждый исправный процесс принимает решение в точности один раз.

2. **Договоренность.** В каждом выполнении все исправные процессы принимают одинаковые решения.

3. **Обоснованность.** Принятое решение совпадает с входными данными хотя бы одного процесса.

Легко видеть, что эти требования правильности подразумевают нетривиальность решения и что при решении этой задачи невозможно обойтись без обнаружения неисправностей (теорема 14.8).

Алгоритм 16.1 позволяет решить рассматриваемую задачу при помощи слабо точных детекторов неисправностей; обязательно есть процесс, который *всегда* находится вне подозрений, но при этом процессам, конечно же, не дано знать заранее, каков этот процесс! Только после того как процессы системы получат сообщение от этого неизвестного процесса, они все могут быть уверены, что они получили то самое сообщение и, следовательно, обладают *одной и той же* информацией. Чтобы избежать трудностей, связанных с выявлением узла, находящегося вне подозрений, в представленном алгоритме процессы работают по очереди; такой подход получил название «метод циркулирующего координатора». Вычисление алгоритма состоит из N туров, каждый процесс выступает в роли координатора в одном из них.

```

 $x_i := \text{input};$ 
for  $r := 1$  to  $N$ 
  do begin if  $i = r$ 
    then forall  $j$  do send  $\langle \text{value}, x_i, r \rangle$  to  $j$ ;
    if collect  $\langle \text{value}, x', r \rangle$  from  $p_r$ 
      then  $x_i := x'$ 
  end;
decide  $x_i$ 

```

Алгоритм 16.1. Алгоритм циркулирующего координатора для процесса p_i с использованием слабо точных детекторов

Мы условимся считать, что процесс p_j , $1 \leq j \leq N$, выступает координатором в j -м туре. В каждом туре координатор оповещает все процессы о своем входном значении, все узлы принимают сообщения от координатора, и если узел преуспевает, то он заменяет свое входное значение той величиной, которую получает от координатора. Доказательство корректности алгоритма представлено здесь только в виде общей схемы, поскольку все подробности можно легко восстановить.

Теорема 16.4. Алгоритм 16.1 решает задачу консенсуса.

Доказательство. Алгоритм удовлетворяет условию завершаемости, так как никакие исправные процессы не оказываются заблокированными в каком-либо туре. Действительно, координатор может быть либо исправным, и тогда он отправляет сообщение, либо неисправным, и тогда он будет рано или поздно заподозрен согласно условию полноты детектора. Более формально это можно показать, воспользовавшись индукцией по i и подтвердив, что каждый исправный процесс достигает i -го тура.

Требование обоснованности выполняется, потому что каждый процесс либо сохраняет свое значение, либо заменяет на значение, полученное от координатора. Это позволяет нам доказать по индукции, что каждый процесс вступает в очередной тур, располагая тем значением, которое было на входе одного из процессов.

Чтобы подтвердить выполнимость требования договоренности, рассмотрим процесс p_j , который никогда не подпадает под подозрение. В j -м туре каждый процесс, который завершает этот тур, получает значение от процесса p_j , и поэтому все процессы завершают j -й тур с одним и тем же значением. Если все процессы вступают в очередной тур с одним и тем же значением, то они и покидают этот тур с тем же самым значением. Отсюда следует, что по завершении алгоритма все процессы будут иметь одно и то же значение. $\square$

Нужно обратить внимание на то, что параметр устойчивости t нигде не фигурирует в нашей программе. На самом деле неисправными могут оказаться даже $N - 1$ процессов. До сих пор неясно, насколько можно уменьшить число туров, если ограничить устойчивость алгоритма. Причина состоит в том, что исправный координатор может подпадать под подозрение. В нашем алгоритме не придается никакого значения тому, является ли процесс исправным; важно лишь, чтобы

процесс был вне подозрения. Количество туров можно сократить, если повысить точность детектора (см. упражнение 16.5).

16.3. Слабо точные в конечном итоге детекторы

В предыдущем параграфе требование слабой точности детекторов повлекло за собой заключение о том, что в одном из *первых* N туров вычисления алгоритма координатором обязательно будет процесс, находящийся вне подозрений. Таким образом, алгоритм устроен так, чтобы завершить работу после N туров. Однако условие *в конечном итоге* слабой точности хотя и гарантирует появление такого координатора *рано или поздно*, не позволяет оценить, в каком туре это произойдет.

Поэтому в дополнение к требованию достижения договоренности решение должно также *зарегистрировать* момент достижения этой договоренности. Для того, чтобы осуществить эту регистрацию, нам понадобится помимо использования детекторов ограничить устойчивость алгоритма некоторой величиной $t < N/2$. Необходимость этого будет подтверждена в § 16.3.1. Тот алгоритм, который будет описан в § 16.3.2 не достигает указанной границы устойчивости; он работает только в том случае, когда $t < N/3$.

16.3.1. Верхняя оценка устойчивости

Те детекторы неисправностей, которые являются точными *в конечном итоге*, могут вести себя произвольным образом в течение сколь угодно продолжительного, но обязательно конечного периода времени. Проще говоря, это означает, что от них нет никакой пользы, если нужно предотвратить ошибки в *конечном* выполнении. Продолжая череду доказательств невозможности решения тех или иных задач, можно получить новые результаты путем переноса *конечных* ошибочных выполнений в модель вычислений, наделенную точными в конечном итоге детекторами неисправностей. Это имеет место, например, для теоремы 14.16, и мы приведем здесь обобщенный вариант этой теоремы для случая модели с обнаружением неисправностей. Эта теорема сформулирована для совершенных в конечном итоге детекторов; она будет верна и для детекторов, являющихся в конечном итоге строгими, поскольку этот класс детекторов удовлетворяет более слабым требованиям.

Теорема 16.5. Не существует алгоритмов консенсуса, использующих класс детекторов $\diamond P$ и сохраняющих устойчивость в случае выхода из строя $t \geq N/2$ процессов.

Доказательство. Допустим, что вопреки утверждению теоремы, такой алгоритм существует. Выделим в множестве процессов $\mathbb{P}$ два непересекающихся подмножества, состоящих из $N - t$ процессов. Мы рассмотрим два сценария работы системы.

- Согласно сценарию 0 все процессы из множества S имеют на входе значение 0, а все процессы, не входящие в множество S , выходят из строя немедленно

в начале вычисления, и их неисправность сразу же обнаруживается всеми процессами из множества S . Так как S состоит из $N - t$ процессов, все процессы из этого множества принимают решение спустя конечное время. И, поскольку на входе у всех уцелевших процессов есть одно и то же значение 0, их решение тоже будет равно 0.

- Согласно сценарию 1 все процессы из множества T имеют на входе значение 1, а все процессы, не входящие в множество T , выходят из строя немедленно в начале вычисления, и их неисправность сразу же обнаруживается всеми процессами из множества T . Так как T состоит из $N - t$ процессов, все процессы из этого множества принимают решение спустя конечное время. И, поскольку на входе у всех уцелевших процессов есть одно и то же значение 1, их решение тоже будет равно 1.

Заметим, что в обоих сценариях поведение детекторов обнаружения неисправностей согласуется с теми требованиями, которые налагаются на класс $\diamond\mathcal{P}$. Мы не будем распространяться о том, почему решение 0 (или 1) является неизбежным для сценария 0 (соответственно 1).

Ввиду того что множества S и T не пересекаются, оба эти сценария можно скомпоновать в новый сценарий, который мы условно назовем сценарий 2. Предположим, что из строя выходят только те процессы, которые *не входят в множество $S \cup T$* . При этом все процессы из множества S имеют на входе 0, все процессы из множества T имеют на входе 1 и выход из строя процессов, не входящих в состав множества $S \cup T$, обнаруживается немедленно всеми процессами из множеств S и T . Кроме того, процессы из множества S ошибочно подозревают все процессы из множества T , а процессы из множества T ошибочно подозревают процессы из множества S сразу после начала вычисления. Требования, определяющие класс детекторов, совершенных *в конечном итоге*, позволяют сохранять такое положение вещей в течение конечного периода времени. Предполагается также, что сообщения между множествами S и T распространяются очень медленно.

Тогда сценарий 2 неотличим от сценария 0 с точки зрения процессов из множества S и неотличим от сценария 1 с точки зрения процессов из множества T . Значит, все эти процессы примутся выполнять те же самые последовательности шагов. Спустя *конечное время* все эти процессы примут то же самое решение, которое каждый из них принял в первоначальных сценариях, т. е. процессы из множеств S и T примут в качестве решения значения 0 и 1 соответственно. После того как хотя бы в каждой из групп хоть один процесс примет решение, ошибочные подозрения будут сняты и все задержанные сообщения поступят по назначению.

Нам удалось, таким образом, построить сценарий поведения, в котором из строя выходит менее t процессов, детекторы неисправностей ведут себя в полном соответствии с определением совершенных в конечном итоге детекторов, и все сообщения, отправленные исправными процессами, достигают исправных адресатов. Но, коль скоро разные процессы приняли разные решения, требование договоренности оказалось нарушенным. $\square$

Доказательство утверждения о том, что консенсус совсем невозможен (теорема 14.8), основывается на построении *бесконечного* выполнения, не приводящего к решению, и в следующем параграфе мы покажем, что этого бесконечного выполнения можно избежать при наличии точных в конечном итоге детекторов.

16.3.2. Алгоритм консенсуса

В алгоритме 16.1 предполагалось, что детектор неисправностей с самого начала является слабо точным; отсюда следовало, что есть хотя бы один процесс (координатор), находящийся вне подозрений, и за счет этого выполнимость условия договоренности обеспечивалась по окончании первых N туров. В алгоритме 16.2 координатор должен развить дальнейшую активность, чтобы установить, удалось ли уже достичь договоренности. Он собирает текущие значения всех процессов и проверяет, совпадают ли они (см. этап 2). Здесь возможен подвох, так как из-за ошибочных подозрений координатор может эффективно собрать данные *менее чем от половины* активных процессов и затем провозгласить это значение в качестве единогласного результата, хотя на самом деле это значение поддерживает меньшая часть процессов. Именно здесь мы воспользуемся ограничением, налагаемым на количество неисправных процессов: на этапе сбора данных координатор не пользуется детектором неисправностей, а просто ожидает поступления $N - t$ бюллетеней, как это происходило в классических методах. Робастность t алгоритма 16.2 ограничена величиной $t < N/3$. Теперь договоренность, замеченная координатором, означает, что по крайней мере большинство процессов имеет вычисленное значение. Совсем не обязательно, чтобы это значение вычислили все процессы: некоторые несогласованные значения могут быть упущены, поскольку координатор ожидает поступления фиксированного числа бюллетеней.

Решение разрешается принимать, если координатор провозгласил, что он выбрал значение v из единогласных бюллетеней, как это отмечено в бите d в сообщении $\langle \text{outcome}, d, v, r \rangle$, посредством которого объявляются итоги r -го тура. Процессы продолжают свою деятельность после объявления решения, для того чтобы помочь другим процессам также принять решение.

Лемма 16.6. *Каково бы ни было натуральное число r и исправный процесс p , процесс p оканчивает r -й тур.*

Доказательство. Предположим, что все исправные процессы начинают работу в r -м туре. Исправный координатор завершит выполнение первого этапа, поскольку ему будет отправлено достаточно много бюллетеней. Всякий исправный процесс завершит выполнение третьего этапа, поскольку координатор, будучи исправным, отправит ему сообщение; если же координатор неисправен, то рано или поздно он будет заподозрен. Таким образом, все исправные процессы завершат выполнение всего тура. Индуктивный переход обоснован. $\square$

Теперь мы попытаемся обосновать, что то большинство процессов, которое достаточно велико, чтобы обеспечить принятие решения, является устойчивым.

```

 $x_i := \text{input} ;$ 
 $r := 0 ;$ 
while  $true$  do
  begin (* Начать новый тур и определить координатора *)
     $r := r + 1 ; c := (r \bmod N) + 1 ;$ 
    (* Этап 1: все процессы отправляют свои значения координатору *)
    send  $\langle \text{value}, x_i, r \rangle$  to  $p_c ;$ 
    (* Этап 2: координатор подводит итоги *)
    if  $i = c$  then
      begin wait until  $N - t$  mesgs.  $\langle \text{value}, v_j, r \rangle$  have been received;
         $v :=$  majority of received values;
         $d := (\forall j : v_j = v) ;$  (* пробегает по всем полученным сообщениям *)
        forall  $j$  do send  $\langle \text{outcome}, d, v, r \rangle$  to  $p_j$ 
      end;
      (* Этап 3: подводятся итоги тура *)
      if collect  $\langle \text{outcome}, d, v, r \rangle$  from  $p_c$  then
        begin  $x_i := v ;$ 
          if  $(d \wedge (y_i = @))$ 
            then decide( $v$ )
          end
        end
      end
  end

```

Алгоритм 16.2. Алгоритм циркулирующего координатора с использованием в конечном итоге совершенных детекторов

Лемма 16.7. Если в начале тура k процессов, где $k \geq N - t$, имеют значение v , то и по окончании тура не менее k процессов будут иметь значение v .

Доказательство. Процесс может заподозрить координатора и сохранить свое значение неизменным. Значение, хранящееся в процессе, может измениться лишь в том случае, если этот процесс получит от координатора сообщение, в котором содержится новое значение. В качестве этого нового значения координатор выбирает значение, присланное ему большинством из тех $N - t$ процессов, от которых он принял сообщения. Самое большее t процессов могут иметь значения, отличные от v , и поэтому координатор получил не менее $N - 2t$ бюллетеней, содержащих значение v , и при этом $N - 2t > t$. Это означает, что в том случае, если координатор отправил на втором этапе работы алгоритма сообщение, то это сообщение содержало значение v , и поэтому число процессов, хранящих значение v , может только увеличиться, но ни в коем случае не уменьшиться. $\square$

Лемма 16.8. Каждый исправный процесс принимает решение.

Доказательство. Точность в конечном итоге гарантирует, что рано или поздно наступит такой тур r , начиная с которого координатор c уже не будет заподозрен никогда. В этом туре все исправные процессы получают от координатора сообщение с одним и тем же значением, приходят к договоренности относительно этого значения и продолжают делать это далее (лемма 16.7). Сле-

довательно, начиная с r -го тура все сообщения, отправленные координаторами, провозглашают, чтобы значение v было принято единодушно ($d = true$). Поэтому всякий процесс примет решение v , как только он примет сообщение на третьем этапе работы алгоритма. Но, коль скоро процесс c уже никогда более не будет заподозрен, он снова станет координатором в $(r + N)$ -м туре, и по окончании этого тура все процессы примут решение. $\square$

Теорема 16.9. Алгоритм 16.2 является алгоритмом консенсуса с детекторами неисправностей типа $\diamond \mathcal{S}$.

Доказательство. Не составляет никакого труда показать, воспользовавшись индукцией по числу туров r (это оставляется читателям в качестве упражнения), что всякое значение, которое в r -туре хранит какой-либо процесс системы, является входным значением одного из процессов. Отсюда следует обоснованность решения. Из леммы 16.7 следует, что свойство договоренности также выполняется, поскольку начиная с того тура, когда было сформировано решение, уже не будет достаточной поддержки в пользу принятия другого решения. И, наконец, условие завершаемости было обосновано в предыдущей лемме. $\square$

16.4. Реализация детекторов неисправности

var D_p	множество процессов ;
$r_p[\mathbb{P}]$	Таймер для каждого процесса

Инициализация детектора:

```

 $D_p := \emptyset ;$ 
forall  $q$  do  $r_p[q] := \sigma + \mu$ 

```

Процесс-отправитель, спустя каждый интервал времени σ :

```

forall  $q$  do send  $\langle \text{alive} \rangle$  to  $q$ 

```

Процесс-получатель после приема сообщения $\langle \text{alive} \rangle$ от процесса q :

```

 $r_p[q] := \sigma + \mu$ 

```

Детектор неисправностей по истечении срока $r_p[q]$:

```

 $D_p := D_p \cup \{q\}$ 

```

Алгоритм 16.3. Совершенный детектор неисправностей

Безусловное достоинство методов, использующих детекторы неисправностей, состоит в том, что эти методы опираются лишь на *свойства* детекторов, а не на особенности их реализации. Однако нам нужно привести примеры таких реализаций, чтобы показать, что происходит за кулисами сцены. В большинстве случаев детекторы представляют собой асинхронную оболочку, содержащую экономную реализацию таймеров.

16.4.1. Синхронные системы: совершенное обнаружение

В основу алгоритм 16.3 для совершенного детектора положена верхняя оценка коммуникационной задержки μ . Активные процессы отправляют сообщения $\langle \text{alive} \rangle$ с интервалом σ , и в том случае, если сообщение от какого-нибудь процесса не было получено в течение отрезка времени $\sigma + \mu$, то это означает, что процесс вышел из строя.

16.4.2. Частично синхронные системы: в конечном итоге совершенное обнаружение

$\text{var } D_p$ множество процессов ;
 $r_p[\mathbb{P}]$ Таймер для каждого процесса ;
 me_p μ первоначальная оценка равна 1

Инициализация детектора:

$D_p := \emptyset$;
forall q **do** $r_p[q] := \sigma + me_p$

Процесс-отправитель, спустя каждый интервал времени σ :

forall q **do** send $\langle \text{alive} \rangle$ to q

Процесс-получатель после приема сообщения $\langle \text{alive} \rangle$ от процесса q :

if $j \in D_p$ **then**
 begin $D_p := D_p \setminus \{q\}$;
 $me_p := me_p + 1$
end;
 $r_p[q] := \sigma + me_p$

Обнаружение неисправности, по истечении срока $r[q]$ elapses:

$D_p := D_p \cup \{q\}$

Алгоритм 16.4. В конечном итоге совершенный детектор неисправностей

Детектор с чуть более слабыми свойствами можно построить, если неизвестна верхняя оценка времени задержки сообщения или по некоторым соображениям ее нецелесообразно вводить в программу (например, чтобы сохранить переносимость программы). Этот детектор (алгоритм 16.4) начинает работу с малой оценкой значения задержки μ . При этом тайм-аут может иногда происходить преждевременно, т. е. для какого-то процесса истекло время ожидания поступления сообщения и процесс объявляется подозрительным, хотя сообщение $\langle \text{alive} \rangle$ еще пребывает на этапе пересылки. Если поступает сообщение от подозрительного процесса, то он немедленно удаляется из множества D и, кроме того, значение параметра μ увеличивается. После некоторого конечного числа таких коррекций значение параметра μ будет превосходить верхнюю оценку времени задержки, и после этого ошибочных подозрений больше не случится. Поскольку число ошибочных подозрений конечно, все эти подозрения возникнут на начальном отрезке

выполнения. А это означает, что данный детектор является в конечном итоге сильно точным.

16.4.3. Заключительные замечания

Эта глава представляет собой лишь введение в тематику обнаружения неисправностей. Мы ограничились только несколькими элементарными результатами, которые можно легко понять и сгруппировать. Ввиду того что эта тема начала разрабатываться сравнительно недавно и еще не привлекла внимания значительного числа исследователей, пока не представляется возможным провести отбор тех результатов, которые могли бы определять направления дальнейших поисков и разработок. В заключении этой главы мы приведем в качестве примеров несколько результатов, касающихся детекторов неисправностей.

Иерархия детекторов неисправностей основывается на понятии *эмуляции* детекторов неисправностей (см. [42]). Детектор $\mathcal{A}$ эмулирует детектор $\mathcal{B}$, если существует такой распределенный алгоритм, который, используя информацию, полученную от детектора $\mathcal{A}$, реализует детектор $\mathcal{B}$. В этом случае говорят, что детектор $\mathcal{B}$ слабее детектора $\mathcal{A}$.

Определяется слабая форма полноты: детектор считается слабо полным, если каждый вышедший из строя процесс начиная с какого-то момента будет постоянно находиться под подозрением у хотя бы одного исправного процесса; см. [42]. Однако легко показать, что слабо полный детектор способен эмулировать строго полный детектор (см. определение 16.1) без ослабления каких-либо свойств точности, указанных в определении 16.2. Поэтому слабой полноте уделяют мало внимания в научной литературе (и, в частности, в этой книге).

Исследователи также задавались вопросом о том, можно ли решить проблему консенсуса при помощи детекторов, более слабых, чем детекторы типа $\diamond S$ (которые использовались в § 16.5.3). Оказалось, что это не так. Чандра и др. в работе [42] показали, что всякий детектор, который позволяет реализовать алгоритм консенсуса, может эмулировать детекторы класса $\diamond S$. В этой работе была развита математическая техника и методы доказательства результатов, относящихся к обнаружению неисправностей.

Не все реализации детекторов неисправностей работают по принципу тайм-аут: в работе [4] был предложен детектор под названием *сердцебиение*, в котором происходит только асинхронный обмен сообщениями. Однако в этом детекторе требуется использовать совсем другой способ взаимодействия между процессами, нежели тот, который рассматривался в настоящей главе, и решения задач, осуществляемые с его помощью, оказываются весьма замысловатыми. Ряд интересных вопросов о том, какие детекторы можно реализовывать асинхронно, какие задачи можно решать с их помощью и др., представляются слишком сложными и новыми, для того чтобы обсуждать их в этой книге.

16.5. Упражнения к главе 16

16.5.1

Упражнение 16.1. На странице 534 перечислены различные возможности, которые есть у исправных процессов, для того чтобы собирать множества сообщений на этапе получения сообщений. Перечислите, для каких из указанных в этой главе четырех типов детекторов (определение 16.3) эти возможности реализуются.

Упражнение 16.2. Рассмотрим шаблон неисправностей F . Докажите, что существует такое t , для которого выполняется равенство $F(t) = \text{Crash}(F)$. В каком месте доказательства используется условие, что множество процессов $\mathbb{P}$ конечно? Покажите, что этот результат будет неверен, если система состоит из бесконечного множества процессов $\mathbb{P}$.

Упражнение 16.3. Требование сильной точности *сильнее*, нежели требование того, что ни один исправный процесс не подпадает под подозрение:

$$\forall F: \forall H \in \mathcal{D}(F): \forall t: \forall p \in \text{Corr}(F), q \notin F(t): p \notin H(q, t).$$

Приведите пример шаблона неисправностей и истории детектора неисправностей, которые удовлетворяют этому свойству, но при этом не являются допустимыми в случае использования сильно точных детекторов.

Упражнение 16.4. Процесс p отправляет сообщение процессу q , а затем процессу r ; процессы q и r собирают сообщения, полученные от процесса p . Приведите пример, который показывает, что даже в случае *совершенного* обнаружения неисправностей возможна ситуация, при которой процесс r получает сообщение, а процесс q нет. В вашем примере процессы q и r обязаны быть исправными.

16.5.2

Упражнение 16.5. Детектор неисправностей называется l -*точным*, если в каждом вычислении хотя бы l исправных процессов находятся вне подозрений. (Отсюда следует, что устойчивость системы должна быть не меньше $N - l$; слабая точность — это 1-точность.) Покажите, какие изменения нужно внести в алгоритм 16.1, чтобы он завершался в $(N - l + 1)$ -м туре.

Упражнение 16.6. Приведите алгоритм 16.2 к такому виду, который был описан для алгоритма 14.5, чтобы получился *конечный* алгоритм, или покажите, что это невозможно сделать, если используются точные в конечном итоге детекторы.

16.5.3

Упражнение 16.7. Приведите пример таких выполнений алгоритма 16.2, в которых решения принимаются в разных турах.

16.5.4

Упражнение 16.8. Как много времени пройдет при работе алгоритма 16.3 между выходом процесса из строя и обнаружением этого обстоятельства?

Упражнение 16.9. Докажите, что алгоритм 16.4 реализует совершенный в конечном итоге детектор неисправностей. Верно ли, что соотношение $me_p \geq \mu$ в конце концов становится верным в каждом выполнении алгоритма?

ГЛАВА 17

СТАБИЛИЗАЦИЯ

В стабилизирующих алгоритмах, которые изучаются в этой главе, для достижения отказоустойчивости используются приемы, решительно отличающиеся от тех, которые рассматривались нами в предыдущих главах, посвященных робастным алгоритмам. При построении робастных алгоритмов придерживаются так называемого *пессимистического* подхода, требующего относиться с подозрением ко всей поступающей информации и предвзято относиться к результатам вычисления строгой проверки, позволяющей гарантировать достоверность всех вычислительных действий, которые совершают исправные процессы. Достоверность результата вычисления должна быть обеспечена даже при наличии в системе неисправных процессов, и это вынуждает налагать определенные ограничения на число неисправностей и на модели неисправностей.

Стабилизирующие алгоритмы строятся исходя из *оптимистических* предположений, которые допускают противоречивое поведение исправных процессов, но гарантируют восстановление правильного поведения спустя некоторое конечное время, после того как все неисправности будут устранены. Таким образом, стабилизирующие алгоритмы способны противостоять *кратковременным* неисправностям; предполагается, что рано или поздно эти неисправности будут устранены, и это позволяет нам не прибегать к моделям, ограничивающим тип неисправностей и количество неисправностей. В случае кратковременных неисправностей, вместо того чтобы считать процессы неисправными, предполагают, что все процессы работают правильно, но в конфигурации в течение короткого времени могут происходить какие угодно нарушения. Пока есть неисправность, история вычисления не принимается в расчет, и та конфигурация, с которой мы начинаем проводить анализ нашего алгоритма, расценивается как начальная конфигурация исправно работающего алгоритма. Поэтому алгоритм называется стабилизирующим, если его поведение в конечном итоге становится корректным (т.е. будет соответствовать спецификации алгоритма) независимо от начальной конфигурации.

Сама идея стабилизации была предложена Дейкстрой в работе [61], но до конца восьмидесятых годов мало что было сделано в этом направлении; таким образом, эту область исследований можно считать сравнительно новой. Тем не менее, в последующие годы и вплоть до времени написания настоящей книги было предложено немало стабилизирующих алгоритмов и связанных с ними результатов. Некоторые из них будут представлены в этой главе.

Хотя термин «самостабилизирующий» часто встречается в научной литературе, далее в этой книге будет повсеместно употребляться термин «стабилизирующийся».

17.1. Введение

17.1.1. Определения

Стабилизирующиеся алгоритмы моделируются системами переходов без начальных конфигураций (попробуйте сопоставить определение, приведенное ниже, с определением 2.1).

Определение 17.1. *Системой переходов* называется пара $S = (\mathcal{C}, \rightarrow)$, в которой $\mathcal{C}$ — это множество *конфигураций*, а $\rightarrow$ — это бинарное *отношение переходов* на множестве конфигураций $\mathcal{C}$. Выполнением системы S называется всякая максимальная последовательность $E = (\gamma_0, \gamma_1, \gamma_2, \dots)$, в которой для любого натурального числа $i \geq 0$ выполняется соотношение $\gamma_i \rightarrow \gamma_{i+1}$.

В отличие от определения 2.2, каждый непустой суффикс вычисления теперь тоже будет являться вычислением¹⁾. Корректность алгоритма, т.е. желаемое «согласованное поведение процессов», определяется спецификацией, которая является не чем иным, как предикатом (обычно его обозначают буквой P), заданным на последовательностях конфигураций.

Определение 17.2. Система S является *стабилизирующейся* относительно спецификации P (или, иначе говоря, система S *стабилизируется* относительно P), если существует подмножество $\mathcal{L} \subseteq \mathcal{C}$ *легитимных конфигураций*, обладающих следующими свойствами.

1. **Корректность.** Каждое выполнение, начинающееся конфигурацией из множества $\mathcal{L}$, удовлетворяет спецификации P .
2. **Сходимость.** В каждом выполнении есть конфигурация из множества $\mathcal{L}$.

Множество легитимных конфигураций обычно бывает замкнутым, т.е. если $\gamma \in \mathcal{L}$ и $\gamma \rightarrow \delta$, то $\delta \in \mathcal{L}$, но, ввиду того что эта особенность не используется в доказательстве следующего утверждения, мы не включили свойство замкнутости в состав приведенного определения.

Теорема 17.3. *Если система S стабилизируется относительно P , то в каждом выполнении системы S есть непустой суффикс, удовлетворяющий предикату P .*

Доказательство. Согласно свойству сходимости в каждом выполнении содержится легитимная конфигурация, и тот суффикс, который начинается этой конфигурацией, удовлетворяет спецификации P согласно свойству корректности. $\square$

¹⁾По-видимому, здесь речь идет о выполнении. — Прим. перев.

Как обосновать свойство стабилизации. Использование легитимных конфигураций позволяет нам применять стандартные методы верификации для подтверждения стабилизируемости алгоритма. Для доказательства сходимости применяются функции нормировки.

Лемма 17.4. *Допустим, что*

- 1) *все заключительные конфигурации принадлежат множеству $\mathcal{L}$;*
- 2) *существует такая функция $f: \mathcal{C} \rightarrow W$, где W — некоторое вполне упорядоченное множество, что для любого перехода $\gamma \rightarrow \delta$ выполняется хотя бы одно из двух соотношений $f(\gamma) > f(\delta)$ или $\delta \in \mathcal{L}$.*

Тогда система $S = (\mathcal{C}, \rightarrow)$ обладает свойством сходимости.

Обоснование корректности можно проводить с помощью классических алгоритмов анализа, рассматривая при этом $\mathcal{L}$ как множество начальных конфигураций; и действительно, интерес представляют лишь те выполнения, которые начинаются конфигурациями из множества $\mathcal{L}$.

Свойства стабилизирующихся алгоритмов. Стабилизирующиеся алгоритмы обладают следующими тремя принципиально важными преимуществами по сравнению с классическими алгоритмами.

1. *Отказоустойчивость.* Как было замечено во введении в эту главу, стабилизирующийся алгоритм обеспечивает полную автоматическую защиту против любых кратковременных сбоев в работе процессов, поскольку этот алгоритм способен восстановить свое поведение из любой конфигурации, независимо от того, насколько серьезно были повреждены его данные.

2. *Инициализация.* Исчезает необходимость в точной и согласованной инициализации алгоритма, поскольку процессы могут начинать свою работу в произвольных состояниях, и все равно рано или поздно они достигнут согласованного поведения.

3. *Динамическая топология.* Стабилизирующийся алгоритм, вычисляющий функцию, зависящую от топологии коммуникаций системы (таблицы маршрутизации, остовное дерево), гарантирует, что после всех изменений в топологии будет построено новое решение.

Четвертое преимущество, а именно возможность осуществлять «последовательную» композицию, не проводя при этом обнаружения завершения вычисления, обсуждается в § 17.3.1. И наконец, многие стабилизирующиеся алгоритмы, известные в настоящее время, устроены проще, чем их классические аналоги, предназначенные для решения тех же сетевых задач. Однако данное качество еще не является окончательным, поскольку эти алгоритмы еще не оптимизированы относительно тех или иных мер сложности; поэтому указанное «преимущество» может исчезнуть, как только мы продвинемся еще дальше в изучении стабилизирующихся алгоритмов.

С другой стороны, налицо имеются и следующие три недостатка.

1. *Начальная несогласованность.* Пока не будет достигнута легитимная конфигурация, результаты, выдаваемые алгоритмом, могут быть несогласованными.

2. *Высокая сложность.* Известные в настоящее время (2002 г.) стабилизирующиеся алгоритмы обычно гораздо менее эффективны, нежели их классические аналоги, предназначенные для решения тех же самых задач.

3. *Нельзя обнаружить стабилизацию.* Находясь внутри системы, невозможно зарегистрировать достижение легитимной конфигурации; поэтому процессы никогда не уверены в том, когда же, наконец, их поведению уже можно будет доверять.

Псевдостабильзация. То свойство стабилизирующихся алгоритмов, которое было установлено в теореме 17.3, можно взять в качестве альтернативного определения, а именно можно просто потребовать, чтобы каждое выполнение имело суффикс, удовлетворяющий требуемой спецификации. Это свойство, однако, не равносильно изначальному определению; Бернс и др. в работе [39] дали ему название *псевдостабильности*.

Чтобы увидеть разницу, нужно рассмотреть систему S с конфигурациями a и b , а также с тремя переходами $a \rightarrow a$, $a \rightarrow b$ и $b \rightarrow b$. В спецификации P утверждается, что «все конфигурации²⁾ одинаковы». Поскольку система S может перейти из конфигурации a в конфигурацию b не более одного раза, легко видеть, что в каждом выполнении есть суффикс, состоящий из одинаковых конфигураций. Конфигурацию a нельзя признать легитимной, так как выполнение $(a, b, b, \dots)$, начинающееся этой конфигурацией, не удовлетворяет требованию P . Таким образом, выполнение $(a, a, a, \dots)$ системы S хотя и удовлетворяет спецификации P , но при этом не содержит легитимных конфигураций.

Бернсу и др. удалось показать, что более слабые требования, которые предъявляются к псевдостабильзации, позволяют получать решения для таких задач, для которых не существует стабилизирующихся алгоритмов; одним из примеров такой задачи является передача последовательностей данных. С другой стороны, в псевдостабильствующих решениях (которые не являются стабилизирующимися) нельзя указать верхнюю оценку числа шагов, которые должна сделать система, прежде чем будет выполнена спецификация P , в то время как для стабилизирующегося алгоритма такую оценку получить можно. В этой главе мы ограничимся изучением стабилизирующихся алгоритмов.

17.1.2. Коммуникация в стабилизирующихся системах

В большинстве предшествующих глав настоящей книги мы имели дело с системами процессов, асинхронно взаимодействующими друг с другом посредством обменов сообщениями. Однако такая модель уже не пригодна для изучения стабилизирующихся алгоритмов.

Действительно, давайте рассмотрим алгоритм с асинхронным обменом сообщениями, в котором есть процесс p , осуществляющий отправку сообщений в каждом состоянии (т.е., p может совершать отправления сообщений в каждом локальном состоянии). Одно из возможных выполнений такой системы состоит только из действий отправления сообщений, которые совершает процесс p , в то

²⁾Одного и того же выполнения — *Прим. перев.*

время как все прочие процессы оказываются «замороженными» в своих начальных состояниях. Ясно, что подобное поведение не может удовлетворять никакой разумной нетривиальной спецификации.

Далее, рассмотрим такой алгоритм, что в нем у каждого процесса есть состояния, в которых этот процесс не отправляет сообщений (а может только принимать сообщения или выполнять внутренние действия). Всякая конфигурация, в которой каждый процесс пребывает в указанном состоянии и все каналы пусты, является заключительной конфигурацией и поэтому должна удовлетворять заданной спецификации. И вновь становится ясно, что никакой нетривиальной спецификации не могут удовлетворять сразу все такие конфигурации.

Стабилизирующийся вариант алгоритма Netchange был представлен в работе [188]; для этого в каждом процессе используются таймеры и введена верхняя оценка времени доставки сообщений. Такая модель может исправно работать на практике, но анализ этого алгоритма затрагивал только технические детали, связанные с отсчетом времени. Поэтому мы воспользуемся более привычной моделью взаимосвязи посредством разделяемых переменных, в которой один процесс может осуществлять запись в одну из переменных, а остальные процессы могут считывать хранящиеся в этой переменной значения.

В модели *со считываемыми состояниями* процесс может автоматически считывать целиком все состояние своего соседа. В такой модели каждый сосед процесса p считывает одно и то же состояние, и поэтому процесс не может передавать разную информацию разным соседям. В модели *со сцепленными регистрами* взаимосвязь между двумя процессами осуществляется при помощи двух регистров, доступ к которым имеют эти два процесса. Каждый из них может проводить запись в один из этих регистров и считывать данные из другого. Процесс, имея возможность записывать разные значения в сцепленные регистры, может тем самым передавать разную информацию разным соседям.

Далее мы будем различать модели с *последовательным* и *параллельным считыванием*, в которых процесс за один неделимый шаг может считать состояние (или содержимое сцепленных регистров) только одного или, соответственно, всех своих соседей.

17.1.3. Пример Дейкстры: кольцо с маркерами

Первый стабилизирующийся алгоритм был предложен Дейкстрой в работе [61], и этот алгоритм предназначен для решения задачи *взаимно исключенного доступа* в кольце процессов. Условия задачи таковы: предполагается, что процессы могут время от времени выполнять *критические фрагменты* своего кода, но при этом в каждый момент времени самое большее один процесс может выполнять такой критический фрагмент. У каждого процесса есть локальный предикат, который принимает значение «истина» в знак свидетельства того, что процесс обладает *привилегией* выполнения критического фрагмента. Спецификация этой задачи такова.

1. В каждой конфигурации самое большее один процесс имеет указанную привилегию.

2. У каждого процесса эта привилегия оказывается бесконечно часто.

Эту задачу можно решить, введя в рассмотрение маркер, циркулирующий по кольцу и закрепляющий привилегию за тем процессом, который в данный момент владеет маркером. Сильная сторона того решения задачи взаимно исключенного доступа, которое было предложено Дейкстрой, состоит в том, что оно автоматически восстанавливается (спустя $O(N^2)$ процессных шагов) даже в тех случаях, когда в системе вообще нет маркеров или имеется сразу несколько маркеров.

Описание решения. Состояния i -го процесса, которые будут обозначаться символами σ_i , — это целые числа из отрезка $0, \dots, K-1$, где K — некоторое целое число, превосходящее N . Процесс с номером i может считывать состояние σ_{i-1} (а также σ_i), а процесс с номером 0 может считывать состояние σ_{N-1} . Все процессы, кроме процесса с номером 0, совершенно одинаковы. Процесс с номером i , $i \neq 0$, обладает привилегией, если $\sigma_i \neq \sigma_{i-1}$, а процесс с номером 0 обладает привилегией, если $\sigma_0 = \sigma_{N-1}$. Процесс, обладающий привилегией, может также изменять свое состояние (предполагается, что он это делает всякий раз по окончании выполнения критического фрагмента или когда ему не нужно вообще выполнять этот фрагмент).

Изменение состояния процесса немедленно влечет за собой утрату его привилегированного положения. Процесс с номером i , $i \neq 0$, может установить значение σ_i равным σ_{i-1} (когда он работоспособен, т.е. когда $\sigma_i \neq \sigma_{i-1}$), выполнив присваивание $\sigma_i := \sigma_{i-1}$. Процесс с номером 0 может установить значение σ_0 отличным от σ_{N-1} (когда эти состояния одинаковы), выполнив присваивание $\sigma_0 := (\sigma_{N-1} + 1) \bmod K$.

Доказательство стабилизации. Из определения привилегированного положения следует, что в каждой конфигурации хотя бы один процесс обладает привилегией (т.е. в этой системе нет заключительных конфигураций). И в самом деле, если никакой процесс, кроме процесса с номером 0, не обладает привилегией, то $\sigma_i = \sigma_{i-1}$ для каждого номера $i > 0$; тогда $\sigma_0 = \sigma_{N-1}$ и процесс с номером 0 находится в привилегированном положении. Далее заметим, что на каждом шаге число привилегированных процессов не увеличивается, поскольку всякий процесс, изменивший свое состояние, утрачивает привилегию, и единственный процесс, который получает привилегию на этом шаге, — это его последователь.

Чтобы убедиться в том, что стабилизация обязательно наступит, будем полагать, что множество $\mathcal{L}$ состоит из всех тех конфигураций, в которых в точности один процесс обладает привилегией.

Лемма 17.5. *Выполнения, начинающиеся легитимными конфигурациями, удовлетворяют спецификации P .*

Доказательство. В легитимной конфигурации можно совершить в точности один шаг, и этот шаг может совершить единственный привилегированный процесс. Сделав этот шаг, процесс утратит свою привилегию, но, ввиду того что конфигураций без привилегий не бывает, последователь этого процесса обязательно обретет привилегию. Следовательно, привилегия циркулирует в кольце,

и каждый процесс становится обладателем привилегии через каждые N шагов выполнения. $\square$

Лемма 17.6. *Алгоритм Дейкстры для кольца с маркером сходится к $\mathcal{L}$.*

Доказательство. Напомним, что каждое выполнение нашей системы бесконечно, так как заключительные конфигурации отсутствуют.

Каково бы ни было выполнение, процесс с номером 0 совершает в нем бесконечно много шагов, поскольку без его участия может совершиться не более $\frac{1}{2}N(N-1)$ шагов. Действительно, поскольку каждый шаг i -го процесса лишает этот процесс привилегии и может передать привилегию только $(i+1)$ -му процессу, функция нормировки

$$F = \sum_{i \in S} (N - i),$$

где $S = \{i: i > 1 \text{ и процесс } i \text{ имеет привилегию}\}$,

убывает с каждым шагом какого-либо процесса, имеющего номер, отличный от 0.

В начальной конфигурации γ_0 процессы могут пребывать не более чем в N различных состояниях; поэтому имеется по меньшей мере $K - N$ состояний, которые не фигурируют в конфигурации γ_0 . Так как процесс с номером 0, выполнив очередной шаг, увеличивает свое состояние (по модулю K), он перейдет в одно из тех состояний, которых нет в конфигурации γ_0 , спустя не более чем N шагов (выполненных этим процессом). Все те процессы, номера которых отличны от 0, только копируют состояния, и это означает, что как только процесс с номером 0 перейдет впервые в одно из таких новых состояний, он будет единственным процессом системы, пребывающим в этом состоянии. И, коль скоро это состояние процесса с номером 0 является уникальным, он не получит в очередной раз привилегии, до тех пор пока не образуется конфигурация, в которой выполнены равенства $\sigma_1 = \sigma_2 = \dots = \sigma_{N-1} = \sigma_0$. А эти равенства описывают одну из легитимных конфигураций.

Так как система достигает легитимной конфигурации еще до того, как будет выполнен $(N+1)$ -й шаг процесса с номером 0, и при этом все остальные процессы могут каждый раз совершать последовательно не более $\frac{1}{2}N(N-1)$ шагов, легитимная конфигурация будет достигнута спустя не более $(N+1)\frac{1}{2}N(N-1) + N = O(N^3)$ шагов.

Проведя более строгий анализ, можно показать, что верхняя оценка числа шагов может быть ограничена величиной $O(N^2)$ (см. упражнение 17.2). $\square$

Следствие 17.7. *Кольцо с маркерами, предложенное Дейкстрой, является стабилизирующимся алгоритмом, решающим задачу взаимно исключенного доступа.*

Униформные решения. Стабилизирующийся алгоритм называется *униформным*, если все его процессы совершенно одинаковы и не имеют отличительных признаков. Поиску униформных решений задачи взаимно исключенного доступа,

а также ряда других задач, было уделено немало внимания в последние годы, и этот интерес обусловлен, главным образом, теоретической значимостью задач. Однако униформные решения могут оказаться привлекательными и для практических приложений, поскольку отличительные признаки процесса обычно хранятся в его памяти, подобно всем другим данным, и в том случае, если возникает неисправность и конфигурация оказывается испорченной, это может нарушить и уникальность отличительных признаков процессов.

Униформный стабилизирующийся алгоритм также можно рассматривать как алгоритм решения соответствующей задачи для анонимных сетей; поэтому те задачи для анонимных сетей, которые не имеют решения (см. гл. 9), не имеют и униформных решений. Проведя анализ симметричных конфигураций и установив, что симметрия может сохраняться неопределенно долго, Дейкстра в работе [62] показал, что в том случае, когда размер кольца является составным числом, построить униформное детерминированное кольцо с маркерами нельзя. В работе Бернса и Пачля [38] было установлено, что униформное решение существует, если размер кольца является простым числом, которое известно всем процессам.

Невозможность создания детерминированных униформных стабилизирующихся алгоритмов для решения задач избрания лидера, построения остова дерева и ряда других задач, в которых происходит разрушение изначальной симметрии системы, доказывается аналогичным образом. Решение обычно удается получить, если разрешается воспользоваться рандомизацией. Коль скоро настоящая глава посвящена стабилизации, а не анонимности, мы будем в дальнейшем, не оговаривая этого особо, исходить из предположения о том, что процессы системы имеют уникальные отличительные признаки или в системе имеется процесс-лидер.

17.2. Графовые алгоритмы

Если целью алгоритма является достижение постуловия ψ , то его спецификация выражается следующим утверждением: «каждая конфигурация удовлетворяет условию ψ ». Мы будем говорить, что система S стабилизируется на условии ψ , если существует такое множество конфигураций $\mathcal{L}$, что

- 1) множество $\mathcal{L}$ замкнуто, и условие $\gamma \in \mathcal{L}$ влечет выполнимость $\psi(\gamma)$;
- 2) в каждом вычислении³⁾ достигается конфигурация $\mathcal{L}$.

Алгоритм может завершать выполнения, но этого, вообще говоря, не требуется. В § 17.2.1 представлен кольцевой алгоритм, выполнения которого в конечном итоге сходится к бесконечной последовательности различных (но ориентированных) конфигураций. В § 17.2.2 вниманию читателей предлагается алгоритм поиска максимального покрытия, в котором все легитимные конфигурации являются заключительными.

17.2.1. Ориентация колец

В задаче ориентации колец рассматривается неориентированное кольцо, состоящее из N процессов, и каждый процесс при этом пометил один из примыкаю-

³⁾правильнее сказать «в каждом выполнении». — Прим. перев.

щих к нему каналов связи меткой *succ* (последователь), а другой канал — меткой *pred* (предшественник). Процесс немедленно изменяет одну из этих пометок, как только обнаруживает, что обе пометки оказались одинаковыми. Метка, которой процесс *p* пометил канал *pq*, будет обозначаться записью l_{pq} . Вначале не предполагается никакой глобальной согласованности пометок. Цель алгоритма ориентации — вычислить согласованное восприятие направления в кольце.

Решение задачи ориентации предусматривает достижение всей системой конфигурации, удовлетворяющей постулову ψ , которое выражается следующим требованием: «для каждого ребра *pq* пометка $l_{pq} = succ$ ставится в том и только том случае, когда $l_{qp} = pred$ ». Как было показано Израэли и Джальфоном в работе [110], задача ориентации не имеет детерминированного решения в модели распределенных систем со считываемыми состояниями. Но мы приведем решение, полученное Израэли и Джальфоном для модели со сцепленными регистрами. Предложенный ими алгоритм является униформным.

В алгоритме 17.1, помимо сцепленных регистров, в которых хранятся пометки *pred* или *succ*, используется переменная *s*, которая может принимать значения *S*, *R*, *I*. На каждом шаге процесс *p* обзвевает свое состояние, состояние соседа *q* и содержимое сцепленных регистров и изменяет свое состояние, если выполняется одно из следующих пяти условий-предохранителей.

var s_p : (*S*, *R*, *I*) ;

(1) $\{state_p = I \wedge s_q = S \wedge l_{qp} = succ\}$
 $s_p := R$; **if** $l_{pq} = succ$ **then flip**

(2) $\{s_p = S \wedge l_{pq} = succ \wedge s_q = R \wedge l_{qp} = pred\}$
 $s_p := I$

(3) $\{s_p = R \wedge l_{pq} = pred \wedge \neg(s_q = S \wedge l_{qp} = succ)\}$
 $s_p := S$

(4) $\{s_p = s_q = S \wedge l_{pq} = l_{qp} = succ\}$
 $s_p := R$; **flip**

(5) $\{s_p = s_q = I \wedge l_{pq} = l_{qp} = succ\}$
 $s_p := S$

procedure flip: reverse *succ* and *pred* for *p*

Алгоритм 17.1. Алгоритм ориентации кольца

Процессы передают друг другу маркеры, и каждый процесс объявляет последователем тот процесс, которому он переправил маркер в последний раз. При столкновении двух маркеров оба они уничтожаются. В конце концов все оставшиеся маркеры будут передаваться в одном и том же направлении, и это приведет

к тому, что в кольце образуется ориентация, которая будет определяться направлением движения уцелевших маркеров.

В состоянии *S* процесс пребывает в ожидании того момента, когда можно будет отправить маркер своему последователю (текущему), а в состоянии *R* процесс пребывает в ожидании получения маркера. В состоянии *I* процесс ничем не занят. Будем говорить, что процесс *обладает маркером*, если он пребывает в состоянии *S* или он пребывает в состоянии *R*, а предшественник его при этом не находится в состоянии *S*. Маркер передается от процесса *p* процессу *q* при выполнении действия (2) нашего алгоритма. Маркер может быть также уничтожен или создан процессом *p* при выполнении, соответственно, действий (4) или (5). Все указанные действия мы будем называть *шагами маркера*. Действия (1) и (3) не оказывают влияние на расположение маркеров, и они будут называться *неслышными шагами*. После шага (1) процесс совершает шаг (3), а после шага (3) он может совершить шаг (4) или шаг (2), и, таким образом, общее число неслышных шагов находится в линейной зависимости от общего числа шагов маркера.

Действие (1) совершается, когда процесс *q* хочет переправить маркер бездействиющему процессу *p* и *p* соглашается принять этот маркер (хотя маркер еще не перемещается) и признать процесс *q* своим предшественником. Действие (2) совершается, когда процесс *p* обнаруживает, что его последователь готов принять тот маркер, которым обладает *p*, и перейти в бездействующее состояние; в этом случае маркер перемещается. Действие (3) совершается, когда процесс *p* обнаруживает, что процесс *q*, от которого он уже получил маркер, перешел в бездействующее состояние; в таком случае процесс *p* предпринимает попытку вручить маркер своему последователю. Последние два действия осуществляются в симметричных ситуациях. Действие (4) относится к столкновению двух маркеров, движущихся в разных направлениях, а действие (5) относится к случаю, когда два бездействующих процесса имеют несогласованную ориентацию. В обоих случаях оба процесса могут выполнить соответствующие действия, и тот из процессов, который выполнит действие первым, нарушит возникшую симметрию⁴⁾. Осуществляя действие (4), процесс *p* принимает маркер от процесса *q* и неявно уничтожает свой собственный маркер. Осуществляя действие (5), процесс *p* порождает маркер, который изменит ориентацию процесса *q*.

О том эффекте, который оказывает каждое действие, можно судить по таблице 17.2, в которой состояние и ориентация каждого процесса условно изображены при помощи стрелок, направленных к последователям этих процессов.

Конфигурация называется легитимной тогда и только тогда, когда она является ориентированной, т. е. все стрелки указывают в одном и том же направлении.

Лемма 17.8. Множество легитимных конфигураций $\mathcal{L}$ замкнуто, и из условия $\gamma \in \mathcal{L}$ следует выполнимость $\psi(\gamma)$.

⁴⁾В работе Израэли и Джальфона рассматривается более высокая степень грануляции действий, при которой требуется явное нарушение симметрии. Если воспользоваться терминологией работы [110], то алгоритм 17.1 работает с центральным демоном. — *Прим. автора*

Номер действия	q	p	
(1)	$\vec{S}$	$I \rightarrow \vec{R}$	Действие (1) применимо вне зависимости от исходной ориентации процесса p
(2)	$\overleftarrow{R}$	$\vec{S} \rightarrow \vec{I}$	
(3)	$\neg \vec{S}$	$\vec{R} \rightarrow \vec{S}$	
(4)	$\vec{S}$	$\vec{S} \rightarrow \vec{R}$	
(5)	$\vec{I}$	$\vec{I} \rightarrow \vec{S}$	

Таблица 17.2. Результат применения действий из алгоритма 17.1.

Доказательство. Легитимные конфигурации являются ориентированными, и отсюда немедленно следует вторая часть утверждения леммы. В каждой легитимной конфигурации могут осуществляться только действия (1), (2) и (3), поскольку действия (4) и (5) могут выполняться только в том случае, когда соседние процессы ориентированы по-разному. Шаги (2) и (3) никогда не изменяют ориентацию процесса, а шаг (1) также не изменяет ориентацию процесса p , если она является точно такой же, как и ориентация процесса q . Отсюда следует замкнутость множества конфигураций $\mathcal{L}$. $\square$

Предложение 17.9. *Всякая заключительная конфигурация является легитимной.*

Доказательство. Рассмотрим заключительную конфигурацию γ . В этой конфигурации ни один процесс не может пребывать в состоянии R , так как в том случае если $s_p = R$, либо p является активным процессом, способным выполнить действие (3), либо его предшественник является активным процессом, способным выполнить действие (2).

Если конфигурация γ не является ориентированной, то в ней найдутся два процесса, указывающие друг на друга. Это такие процессы p и q , для которых верны $l_{pq} = l_{qp} = succ$. Если только один из этих процессов пребывает в бездействующем состоянии, то осуществимо действие (1), если оба процесса пребывают в бездействии, то осуществимо действие (5), а если оба процесса хотят переслать маркер, то осуществимо действие (4). Поэтому заключительная конфигурация, в которой нет процессов, осуществляющих прием, является ориентированной. Значит, любая заключительная конфигурация является ориентированной. $\square$

Теорема 17.10. *Протокол сходится к легитимной конфигурации.*

Доказательство. Порождение маркеров возможно только в ситуации $\vec{I} \overleftarrow{I}$, а такая пара никогда не образуется при выполнении какого-либо из указанных пяти действий. Значит, маркеры порождаются только изначально бездействующими процессами на первом шаге их работы. Отсюда следует, что общее число маркеров (т.е. исходных маркеров, к которым добавляются маркеры, порожденные по ходу выполнения алгоритма), с которыми приходится иметь дело по ходу одного выполнения алгоритма, не превосходит числа процессов N .

Если маркер уже существовал в некоторой конфигурации по ходу выполнения алгоритма и этот маркер уже переместился k раз, то последовательность, состоящая из $k + 1$ процессов, в которых побывал этот маркер, имеет согласованную ориентацию. Все эти процессы уже настроили должным образом свою ориентацию и переправили друг другу этот маркер, и никакой маркер, который мог бы снова нарушить сложившуюся ориентацию, не остался «позади» него. Значит, как только какой-либо маркер совершит $N - 1$ переходов, все процессы будут ориентированы одинаково и конфигурация станет легитимной.

Поскольку по ходу выполнения в системе возникает не более N разных маркеров, легитимная конфигурация будет достигнута спустя не более чем N^2 шагов маркера. $\square$

17.2.2. Максимальное покрытие

Покрытием вершин в графе называется множество ребер, никакие два из которых не примыкают к одной и той же вершине. Покрытие называется *максимальным*, если его нельзя расширить за счет добавления других ребер. Максимальное покрытие можно построить за время, линейное относительно числа ребер. Все ребра рассматриваются по очереди, каждое ребро включается в покрытие, если оно не инцидентно ни одному из ранее включенных в покрытие ребер. Этот алгоритм, однако, по сути своей является последовательным алгоритмом и не предназначен для распределенных выполнений. В этом параграфе мы предлагаем вниманию читателей стабилизирующийся алгоритм построения максимальных покрытий, предложенный Хсу и Хуаном в работе [108].

Алгоритм построения покрытия. Каждый процесс p располагает переменной $pref_p$, которая принимает значения из множества $Neigh_p \cup \{nil\}$. Значение этой переменной следует истолковывать как имя *наиболее предпочтительного* соседа процесса p . Если $pref_p = q$, то процесс p остановил свой выбор на соседе q , с тем чтобы включить соединяющее их ребро pq в покрытие; в том случае, если $pref_p = nil$, то это означает, что процесс p еще не выбрал подходящего партнера. В зависимости от предпочтений процесса p и его соседей можно выделить следующие пять случаев. Если процесс p остановил свой выбор на процессе q , то p либо пребывает в ожидании (решения, которое примет q), если процесс q еще не совершил своего выбора, либо фиксирует свой выбор, если процесс q предпочел выбрать процесс p , либо изменяет свой выбор, если процесс q выбрал иного своего соседа, отличного от p . Процесс p , еще не совершивший свой выбор, считается мертвым, если все его соседи уже участвуют в покрытии, и считается свободным, если хотя бы один из его соседей еще не выбрал своего партнера.

Более формально это представляется так:

$$\begin{aligned} \text{wait}(p) &\equiv \text{pref}_p = q \in \text{Neigh}_p \wedge \text{pref}_q = \text{nil}, \\ \text{match}(p) &\equiv \text{pref}_p = q \in \text{Neigh}_p \wedge \text{pref}_q = p, \\ \text{chain}(p) &\equiv \text{pref}_p = q \in \text{Neigh}_p \wedge \text{pref}_q = r \in \text{Neigh}_q \wedge r \neq p, \\ \text{dead}(p) &\equiv \text{pref}_p = \text{nil} \wedge \forall q \in \text{Neigh}_p : \text{match}(q), \\ \text{free}(p) &\equiv \text{pref}_p = \text{nil} \wedge \exists q \in \text{Neigh}_p : \neg \text{match}(q). \end{aligned}$$

Постусловие этого алгоритма таково:

$$\psi \equiv \forall p: (\text{match}(p) \vee \text{dead}(p)).$$

Предложение 17.11. Если выполнено условие ψ , то множество ребер $M = \{(p, \text{pref}_p) : \text{pref}_p \neq \text{nil}\}$ образует максимальное покрытие.

Доказательство. Если для ребра pq имеет место включение $pq \in M$, то согласно определению множества M верно одно из двух равенств $\text{pref}_p = q$ или $\text{pref}_q = p$. Поскольку процесс q не пребывает в ожидании и не связан другими обязательствами, последнее равенство также означает, что $\text{pref}_p = q$. Отсюда следует, что в состав множества M включается не более одного ребра, примыкающего к вершине p . Значит, M — это покрытие.

Чтобы убедиться в том, что это покрытие максимально, допустим, что множество ребер $M \cup \{pq\}$, где $pq \notin M$, также является покрытием. В таком случае множество M не содержит ребер, инцидентных вершине p , и поэтому для вершины p выполняется соотношение $\text{dead}(p) \vee \text{free}(p)$. Коль скоро условие ψ также выполнено, мы вынуждены признать, что имеет место свойство $\text{dead}(p)$. Но свойство $\text{dead}(p)$ подразумевает, что в множестве M соседняя вершина q связана обязательствами (выполняется свойство $\text{match}(q)$). Это означает, что в множестве M содержится ребро, инцидентное вершине q , вопреки предположению о возможности расширения покрытия M за счет добавления ребра pq . $\square$

var $\text{pref}_p : \text{Neigh}_p \cup \{\text{nil}\} ;$

$\mathbf{M}_p:$ $\{\text{pref}_p = \text{nil} \wedge \text{pref}_q = p\}$
 $\text{pref}_p := q$

$\mathbf{S}_p:$ $\{\text{pref}_p = \text{nil} \wedge \forall r \in \text{Neigh}_p : \text{pref}_r \neq p \wedge \text{pref}_q = \text{nil}\}$
 $\text{pref}_p := q$

$\mathbf{U}_p:$ $\{\text{pref}_p = q \wedge \text{pref}_q \neq p \wedge \text{pref}_q \neq \text{nil}\}$
 $\text{pref}_p := \text{nil}$

Алгоритм 17.3. Алгоритм построения максимального покрытия

Алгоритм 17.3 решения рассматриваемой задачи описан для модели со считываемыми состояниями всех соседей; каждый процесс p может выполнять три действия. Процесс p может образовать пару со своим соседом q (действие $\mathbf{M}_p$),

если p свободен и при этом процесс q уже остановил свой выбор на процессе p . Если процесс p свободен, но не может образовать пару, то он выбирает свободного соседа при условии, что такой выбор можно сделать (действие $\mathbf{S}_p$). Если же тот процесс, который был выбран процессом p , не отвечает ему взаимностью, то p отменяет свой выбор (действие $\mathbf{U}_p$).

Анализ алгоритма.

Лемма 17.12. Конфигурация γ является терминальной тогда и только тогда, когда выполняется условие $\psi(\gamma)$.

Доказательство. Процесс p может выполнить действие $\mathbf{M}_p$ только тогда, когда его сосед пребывает в ожидании. Для выполнения действия $\mathbf{S}_p$ требуется, чтобы процесс p был свободен. А для выполнения действия $\mathbf{U}_p$ нужно, чтобы процесс p выбрал того соседа, который не отвечает ему взаимностью. Следовательно, в том случае, когда выполняется условие $\psi(\gamma)$, в конфигурации γ нельзя совершить никакого действия.

Если условие ψ не выполняется, то найдется такой процесс p , который либо пребывает в ожидании, либо свободен, либо сделал неподтвержденный выбор. Если выбор, сделанный процессом p , не подтверждается тем соседом, которому было оказано предпочтение, то выполняется действие $\mathbf{U}_p$, а если процесс p способен подтвердить тот выбор, который был сделан его соседом q , то можно осуществить действие $\mathbf{M}_p$. И, наконец, предположим, что процесс p еще не сделал своего выбора и q — это соседний процесс, который еще не выразил своего предпочтения. Поскольку p еще не связан никакими обязательствами, процесс q не может быть мертвым. Если процесс q пребывает в ожидании, то один из его соседей может образовать с ним пару, выполнив соответствующее действие. Если процесс q предпочел выбрать соседа, который не отвечает ему взаимностью, то может быть выполнено действие $\mathbf{U}_q$. А если процесс q свободен, то процессы p и q могут выбрать друг друга. Следовательно, если не выполняется условие ψ , то конфигурация не является заключительной. $\square$

Лемма 17.13. Алгоритм 17.3 достигает заключительной конфигурации после выполнения $O(N^2)$ шагов.

Доказательство. Рассмотрим функцию нормировки F , значениями которой являются пары $(c+f+w, 2c+f)$, где c, f и w обозначают соответственно число процессов системы, которые выбрали соседа, не отвечающего взаимностью, остаются свободными и пребывают в ожидании выбора соседа. Мы покажем, что по ходу выполнения значения функции F убывают (в лексикографическом порядке). Заметим, что процессы, образовавшие пару или ставшие мертвыми, остаются в таком состоянии навсегда; поэтому значение выражения $c + f + w$ не может возрастать.

Действие $\mathbf{M}_p$. Это действие применимо только в том случае, когда процесс p свободен и один из его соседей q ожидает подтверждения своего выбора. В результате выполнения этого действия два процесса образуют пару и значение выражения $c + f + w$ уменьшается на 2. (На самом деле эта сумма может еще более

уменьшиться, если некоторые из соседей процессов p и q станут после этого мертвыми.)

Действие S_p . Это действие применимо только тогда, когда процесс p свободен. В результате выполнения этого действия процесс p выбирает один из свободных процессов и начинает ожидать подтверждения своего выбора. Таким образом, значение выражения $2c + f$ уменьшается на 1. При этом ни один процесс не освобождается и никакие ожидания не обманываются, поскольку это действие применимо только тогда, когда ни один из процессов не ожидает от p подтверждения своего выбора, и поэтому ни один свободный процесс не делает неподтвержденного выбора.

Действие U_p . Это действие применимо только тогда, когда процесс p сделал выбор, который не отвечает взаимностью, и в результате выполнения этого действия процесс p становится либо свободным (если есть хотя бы один незанятый сосед), либо мертвым (если все соседи заняты). Вследствие этого значение выражения $2c + f$ уменьшается по крайней мере на 1. При этом те соседи процесса p , которые выбрали p , но не получили от p взаимности, могут перейти в состояние ожидания, и поэтому значение величины c может уменьшиться еще более.

Так как значение выражения $c + f + w$ ограничено величиной N , а значение выражения $2c + f$ — величиной $2N$, функция F может принимать не более $(N + 1)(2N + 1)$ различных значений. Поэтому всякое выполнение завершается спустя не более $2N^2 + 3N$ шагов. $\square$

Легко видеть, что алгоритм 17.3 стабилизируется на условии ψ , если в качестве легитимных конфигураций взять все конфигурации, удовлетворяющие требованию ψ .

17.2.3. Выборы лидера и построение остовных деревьев

Аффек, Каттен и Янг в работе [2] предложили стабилизирующийся алгоритм вычисления в сети такого остовного дерева, в котором процесс с наибольшим номером является корневым узлом. Мы опишем этот алгоритм для модели вычисления со считываемыми состояниями всех соседей (хотя его можно использовать также и в модели, в которой состояния соседей считываются поочередно), предполагая, что сеть является связной. В этом параграфе мы будем использовать заглавные буквы для обозначения переменных, значениями которых являются имена процессов, и прописные буквы для обозначения предикатов.

В процессе p переменные $Root_p$, par_p и Dis_p используются для описания устройства дерева. Мы введем в рассмотрение следующие предикаты:

$$\begin{aligned} root(p) &\equiv Root_p = p \wedge Dis_p = 0, \\ child(p, q) &\equiv Root_p = Root_q > p \wedge par_p = q \in Neigh_p \wedge Dis_p = Dis_q + 1, \\ tree(p) &\equiv root(p) \vee \exists q: child(p, q), \\ lmax(p) &\equiv \forall q \in Neigh_p: Root_p \geq Root_q, \\ sat(p) &\equiv tree(p) \wedge lmax(p). \end{aligned}$$

Постусловие ψ для рассматриваемого алгоритма задается формулой $\forall p: sat(p)$.

Лемма 17.14. Из постусловия ψ следует, что множество ребер $\{(p, q): child(p, q)\}$ образуют остовное дерево, имеющее в качестве корневого узла процесс с наибольшим именем.

Доказательство. Так как все процессы p обладают свойством $lmax(p)$ и сеть является связной, значение переменной $Root$ одинаково у всех процессов. Процесс p_0 , имеющий минимальное значение переменной Dis , не может иметь соседа q , удовлетворяющего соотношению $Dis_p = Dis_q + 1$; поэтому процесс p_0 обладает свойством $root(p_0)$, и общим значением всех переменных $Root$ является имя процесса p_0 . Но тогда каждый процесс p , отличный от p_0 , обладает свойством $child(p)$. Ввиду того что имеет место соотношение $child(p, q) \implies (Dis_p > Dis_q)$, бинарное отношение $child$ является ациклическим. Отсюда следует утверждение леммы. $\square$

Описание алгоритма. Как видно из описания алгоритма 17.4, чтобы добиться выполнимости постусловия ψ , те процессы p , для которых предикат $sat(p)$ уже принял значение $true$, не должны изменять значения всех тех своих переменных,

```

var  $Root_p, par_p, Dis_p$ ;          (* Описывают устройство дерева *)
     $Req_p, From_p, To_p, Dir_p$ ;    (* Используются для передачи запросов *)

 $B_p$ : (* Становится корнем *)
{  $\neg tree(p)$  }
 $Root_p := p$ ;  $Dis_p := 0$ ;  $Req_p := p$ ;  $To_p := q$ ;  $Dir_p := Ask$ 

 $A_p$ : (* Просит разрешения присоединиться *)
{  $tree(p) \wedge \neg lmax(p)$  }
Select  $q \in Neigh_p$  with maximal value of  $Root_q$ ;
 $Req_p := p$ ;  $From_p := p$ ;  $To_p := q$ ;  $Dir_p := Ask$ 

 $J_p$ : (* Соединение двух деревьев *)
{  $tree(p) \wedge \neg lmax(p) \wedge grant(To_p, p)$  }
 $par_p := q$ ;  $Root_p := Root_q$ ;  $Dis_p := Dis_q + 1$ ;
 $Req_p := From_p := To_p := Dir_p := undef$ 

 $C_p$ : (* Сброс значений переменных запроса *)
{  $sat(p) \wedge \neg \exists q: forw(p, q) \wedge \neg idle(p)$  }
 $Req_p := From_p := To_p := Dir_p := undef$ 

 $F_p$ : (* Передается запрос *)
{  $sat(p) \wedge idle(p) \wedge asks(q, p)$  }
 $Req_p := Req_q$ ;  $From_p := q$ ;  $To_p := par_p$ 

 $G_p$ : (* Дается разрешение на присоединение *)
{  $sat(p) \wedge root(p) \wedge forw(p, q) \wedge Dir_p = Ask$  }
 $Dir_p := Grant$ 

 $R_p$ : (* Разрешение ретранслируется *)
{  $sat(p) \wedge grant(par_p, p) \wedge Dir_p = Ask$  }
 $Dir_p := Grant$ 

```

Алгоритм 17.4. Алгоритм построения остовного дерева

которые определяют устройство дерева. Процесс p , для которого предикат $sat(p)$ еще имеет значение *false*, пытается сделать условие $sat(p)$ истинным, объявляя себя сыновней вершиной того из своих соседей, который имеет наибольшее значение переменной $Root$ (действие J_p).

Имя, которое является значением переменной $Root_q$, будем называть *фальшивым корнем*, если в сети не существует процесса с таким именем; фальшивые корни могут быть в самом начале работы алгоритма, но они не будут возникать по ходу его выполнения. Главная задача, которую приходится решать при проектировании нашего алгоритма, состоит в следующем: нужно сделать так, чтобы процессы не могли становиться сыновними вершинами узлов с фальшивыми корнями бесконечно часто. Для этого присоединение процесса p к дереву, в котором находится процесс q , проводится в три этапа. Сначала процесс p становится корневой вершиной (действие B_p), затем он просит разрешения присоединиться (действие A_p), и, наконец, он присоединяется, как только получит разрешение от процесса q (действие J_p).

Остальные четыре действия (C_p , F_p , G_p и R_p) служат для реализации механизма запросов и откликов; они выполняются только в том случае, когда процесс p удовлетворяет условию $sat(p)$. Значение переменной Req_p — это имя того процесса, запрос о присоединении которого обрабатывается процессом p ; значение переменной $From_p$ — это имя того соседа, запрос которого прочитывает процесс p ; значение переменной To_p — это имя того соседа, которому процесс p переадресует запрос; переменная Dir_p может принимать значения *Ask* и *Grant*, которые служат признаком того, был ли удовлетворен поступивший запрос. Мы будем рассматривать следующие предикаты:

$$\begin{aligned} idle(p) &\equiv Req_p = From_p = To_p = Dir_p = undef, \\ asks(p, q) &\equiv ((root(p) \wedge Req_p = p) \vee child(p, q)), \\ &\quad \wedge To_p = q \wedge Dir_p = Ask, \\ forw(p, q) &\equiv Req_p = Req_q \wedge From_p = q \wedge To_q = p \wedge To_p = par_p, \\ grant(p, q) &\equiv forw(p, q) \wedge Dir_p = Grant. \end{aligned}$$

Процесс p сбрасывает старые значения переменных, чтобы приступить к обработке поступившего запроса (действие C_p), если он не занят обработкой какого-либо запроса и значения этих переменных уже не являются неопределенными. Затем, в том случае, если процесс p пребывает в бездействии и один из его соседей q обращается к p с запросом (это может быть корневой узел, обращающийся с просьбой о присоединении, или сыновний узел, переадресующий поступивший ему запрос), процесс может приступить к обработке этого запроса (действие F_p). Если p — это корневой узел, то при получении запроса он удовлетворяет эту просьбу (действие G_p), а если p переадресует запрос в родительский узел и эта просьба удовлетворяется, то p ретранслирует это согласие (действие R_p).

Корректность алгоритма. Афек и др. в работе [2] устанавливают корректность предложенного алгоритма, рассматривая различные варианты его поведения и устанавливая определенные свойства выполнений этого алгоритма.

Описанный механизм запросов и откликов устроен так, что процесс может присоединиться к дереву с фальшивым корнем только конечное число раз, поскольку несуществующий корень не может удовлетворить просьбу о присоединении, а вначале может существовать только конечное число фальшивых разрешений на присоединение. Далее, никакая переменная $Root$ не может сколь угодно долго иметь в качестве значения имя фальшивого корня. Это объясняется тем, что процесс, переменная Dis которого имеет в качестве значения наименьшее имя, не удовлетворяет условию *tree*, если у этого процесса фальшивый корень, и поэтому он вынужден установить в качестве значения переменной $Root$ свой собственный отличительный признак (имя). Отсюда следует, что в конечном итоге все фальшивые корни исчезнут и процесс с наибольшим отличительным признаком станет корневым узлом, поскольку это единственный способ добиться выполнения условия *sat* в том случае, когда значение переменной $Root$ нельзя превзойти.

Всякий узел, не принадлежащий дереву, корневая вершина которого имеет наибольший отличительный признак, но зато имеющий одного из соседей, принадлежащих этому дереву, заведомо не удовлетворяет предикату *sat* и попытается присоединиться к указанному дереву, в то время как все вершины, принадлежащие этому дереву, никогда его не покинут. Коль скоро все некорректные запросы, перемещающиеся в указанном дереве, рано или поздно исчезнут, присоединение новых узлов к этому дереву будет продолжаться до тех пор, пока оно не будет охватывать все узлы сети.

17.3. Общие методы стабилизации

Стремясь добиться стабилизируемости, мы существенно усложняем проектирование распределенных алгоритмов; однако есть некоторые общие принципы проектирования, которые полезно использовать. В § 17.3.1 мы обсудим, как можно «последовательно составлять» стабилизирующиеся вычисления, хотя стабилизируемость тех из них, которые выполняются вначале, нельзя обнаружить. Мы будем опираться на результаты работы Хермана [103], хотя эти методы были известны еще до публикации этой работы. В § 17.3.2 мы покажем, что стабилизирующиеся алгоритмы можно построить для решения любой задачи, которую можно сформулировать как проблему поиска «минимального пути» в подходящем метрическом пространстве.

17.3.1. Композиция протоколов

Во многих случаях работа алгоритма, стремящегося достичь выполнимости требуемого постуловия ϕ , может быть разбита на два этапа: на первом этапе алгоритм добивается выполнимости некоторого постуловия θ , а на втором этапе исходя из постуловия θ достигается выполнимость постуловия ϕ . Примеры подобных составных алгоритмов довольно многочисленны, и некоторые из них рассматривались в этой книге.

1. *Маршрутизация.* Большинство методов маршрутизации начинают работу с того, что строят таблицы маршрутизации в каждом узле, а затем, руководствуясь этими таблицами, переправляют пакеты.

2. *Избрание лидера.* Выборы лидера в сетях обычно проводятся для того, чтобы впоследствии можно было применять централизованные алгоритмы.

3. *Обнаружение блокировки.* Алгоритм глобальной разметки, занимающийся обнаружением блокировок (алгоритм 10.8), вначале вычисляет моментальное состояние основного вычисления, а затем проводит анализ полученной конфигурации.

4. *Раскраска графов.* Раскраску планарного графа с использованием шести различных цветов можно вычислить, обнаружив подходящую ациклическую ориентацию этого графа (см. определение 5.12) и затем раскрашивая вершины по порядку в соответствии с этой ориентацией.

Наиболее естественным образом провести передачу управления от одного классического распределенного алгоритма другому можно при помощи протокола, обнаруживающего завершение вычисления. Этот протокол (см. гл. 8) следит за выполнением первого этапа алгоритма, и, как только он обнаруживает, что первый этап завершен (т. е. выполнено условие θ), все процессы приступают к выполнению второго этапа. Но если проектируемый алгоритм должен быть стабилизирующимся, то такой прием применять нельзя, поскольку невозможно следить за завершением первого этапа, занимаясь при этом стабилизацией.

К счастью, подобная композиция стабилизирующихся алгоритмов устроена гораздо проще: к выполнению второго этапа можно приступить в любое время даже тогда, когда первый этап еще не завершился. Если бы классический алгоритм (работающий на втором этапе) начал выполняться еще до того, как выполнилось его предусловие, он мог бы перейти в такое ошибочное состояние, из которого он бы уже не выбрался; именно поэтому сперва необходимо установить, что первый этап выполнения уже завершен. Однако в том случае, если вычисление на втором этапе обладает способностью стабилизироваться, оно рано или поздно добьется выполнимости своего постусловия, несмотря на те ошибочные шаги, которые были сделаны до того, как предусловие второго этапа приняло значение *true*.

Сопутствующая композиция алгоритмов. Херман в своей работе [103] сумел формализовать приведенные здесь соображения, введя в рассмотрение оператор алгоритмической композиции. Попробуем представить себе программу как совокупность переменных и описаний элементарных действий над этими переменными.

Определение 17.15. Пусть S_1 и S_2 — это две такие программы, что ни одна из переменных, значение которой изменяется в программе S_2 , не входит в состав программы S_1 . Тогда *сопутствующая композиция* программ S_1 и S_2 , для обозначения которой мы будем использовать запись $S_1 \triangleright S_2$, — это программа, которая состоит из всех переменных и действий обеих программ S_1 и S_2 .

В этом определении под программой S_1 следует понимать ту программу, которая работает на первом этапе вычисления, а под программой S_2 — ту программу, которая работает на втором этапе. Ограничение, налагаемое на устройство этих программ, подразумевает, что те промежуточные результаты, которые вычисляются по ходу работы программы S_2 не используются программой S_1 , т. е. между программами S_2 и S_1 нет обратной связи и все переменные программы S_1 в программе S_2 играют роль констант⁵⁾.

Будем считать, что θ — это предикат над множеством переменных программы S_1 , а ψ — это предикат над множеством переменных программы S_2 . По ходу работы составного алгоритма условие θ будет установлено программой S_1 , и вслед за этим условие ψ будет установлено программой S_2 . При этом нужно иметь в виду два следующих технических требования. Во-первых, всякое выполнение составной программы должно быть *справедливым* по отношению к обеим ее компонентам; из рассмотрения исключаются все те выполнения, в которых все шаги выполнения относятся только к одному этапу, в то время как на другом этапе не совершается ни одного шага.

Определение 17.16. Выполнение сопутствующей композиции $S_1 \triangleright S_2$ называется *справедливым* по отношению к программе S_i , если по ходу этого выполнения программа S_i либо совершает бесконечно много шагов, либо начиная с какого-то шага не имеет ни одного действия, которое может быть выполнено.

Во-вторых, мы условимся считать, что переменные, значения которых считывает программа S_2 , не изменяются программой S_1 , едва только будет выполнено условие θ . Таким образом мы исключаем из рассмотрения такие искусственно придуманные ситуации, когда программа S_1 «мешает» программе S_2 правильно работать, постоянно изменяя значения некоторых переменных, но соблюдая при этом выполнимость условия θ .

Теорема 17.17. Если выполнены следующие четыре требования:

- 1) программа S_1 обладает способностью стабилизироваться на постусловии θ ;
- 2) программа S_2 обладает способностью стабилизироваться на условии ψ , если начальные конфигурации удовлетворяют предусловию θ ;
- 3) как только выполняется условие θ , программа S_1 прекращает изменять значения тех переменных, которые используются для считывания данных программой S_2 ;
- 4) все выполнения являются справедливыми по отношению к обеим программам S_1 и S_2 ,

то сопутствующая композиция $S_1 \triangleright S_2$ обладает способностью стабилизироваться на условии ψ .

⁵⁾Последнее замечание не совсем верно. Если действия программ S_1 и S_2 могут чередоваться, то, с точки зрения программы S_2 , значения некоторых переменных могут «чудесным образом» изменяться, несмотря на то что действия программы S_2 не модифицировали значения этих переменных. Далее автор уточняет это замечание. — Прим. перев.

Доказательство. Чтобы доказать это утверждение, мы проведем анализ последовательности конфигураций, образующей выполнение составной программы $S_1 \triangleright S_2$. Для всякой конфигурации γ выполнения системы $S_1 \triangleright S_2$ условимся использовать запись $\gamma^{(i)}$ для обозначения проекции набора значений переменных в конфигурации γ на множество переменных программы S_i . Для всякого выполнения $E = (\gamma_0, \gamma_1, \dots)$ системы $S_1 \triangleright S_2$ мы будем обозначать символом $E^{(i)}$ прореженную последовательность проекцией $(\gamma_0^{(i)}, \gamma_1^{(i)}, \dots)$, из которой удалены повторные элементы.

Рассмотрим произвольное выполнение $E = (\gamma_0, \gamma_1, \dots)$ системы $S_1 \triangleright S_2$, а также проекцию этого выполнения $E^{(1)}$. Так как программа S_2 не изменяет значений переменных, входящих в состав S_1 , все изменения, которые происходят в $\gamma^{(1)}$, обусловлены действиями программы S_1 . Отсюда следует, что $E^{(1)}$ — это выполнение программы S_1 . Коль скоро выполнение E является справедливым относительно S_1 , то в том случае, если последовательность $E^{(1)}$ конечная, последняя конфигурация последовательности является заключительной конфигурацией программы S_1 . Поскольку S_1 обладает способностью стабилизироваться на условии θ , выполнимость постуловия θ должна быть рано или поздно достигнута, т.е. существует такое i , что $\theta(\gamma_i)$ принимает значение *true* для всех $j \geq i$.

Рассмотрим теперь суффикс E_i последовательности E , начинающийся конфигурацией γ_i , т.е. рассмотрим последовательность $(\gamma_i, \gamma_{i+1}, \dots)$. Каждая конфигурация этой последовательности удовлетворяет условию θ , и поэтому (согласно третьему требованию, приведенному в формулировке теоремы) значения переменных, используемых программой S_2 для считывания, не изменяются программой S_1 . Следовательно, последовательность $(\gamma_i^{(2)}, \gamma_{i+1}^{(2)}, \dots)$ представляет собой выполнение программы S_2 , которое, будучи конечным, обязательно оканчивается заключительной конфигурацией программы S_2 . Так как S_2 обладает способностью стабилизироваться на условии ψ , если начальные конфигурации удовлетворяют предусловию θ , выполнимость условия ψ будет достигнута и сохранена. $\square$

Если стабилизируемость программ S_1 и S_2 доказывается при помощи подходящих функций нормировки f_1 и f_2 (принимающих значения в областях W_1 и W_2 соответственно), то можно построить функцию нормировки для композиции $S_1 \triangleright S_2$. Рассмотрим функцию $f(\gamma) = (f_1(\gamma^{(1)}), f_2(\gamma^{(2)}))$. Так как каждое действие программы S_1 уменьшает значение функции f_1 , а каждое действие программы S_2 уменьшает значение функции f_2 и оставляет неизменным значение функции f_1 , значения функции f убывают в лексикографическом порядке с каждым шагом композиции $S_1 \triangleright S_2$. Если, однако, одна из программ добивается выполнимости своего постуловия без завершения вычисления (как это делает, например, программа ориентации кольца), то приходится использовать допущение справедливости, чтобы обеспечить прогресс в выполнении обеих программ. Более подробно вопросы, связанные с допущениями справедливости, обсуждаются в работе [89].

Применение теоремы: раскраска планарных графов в шесть цветов. В качестве примера применения принципа композиции протоколов рассмотрим стабилизирующийся алгоритм раскраски планарных графов в шесть цветов. Мы будем

считать, что в каждом процессе p есть переменная c_p , принимающая значения из множества $\{1, 2, 3, 4, 5, 6\}$. Граф будет правильно раскрашен в шесть цветов, если любые две соседние вершины будут окрашены в разные цвета, т.е. будет истинен предикат ψ , который задается формулой

$$\psi \equiv (\forall pq \in E: c_p \neq c_q).$$

Построение нашего алгоритма строго следует доказательству существования указанной раскраски, основанному на следующем факте.

Предложение 17.18. *В любом планарном графе есть хотя бы одна вершина, степень которой не превосходит 5.*

Лемма 17.19. *В любом планарном графе можно задать такую ациклическую ориентацию ребер, при которой полустепень исхода каждой вершины не будет превосходить 5.*

Доказательство. Воспользуемся индукцией по числу вершин. Если граф состоит из единственной вершины, все очевидно. Предположим, что в графе имеется не менее двух вершин. Среди них есть такая вершина, степень которой не превосходит 5. Граф $G - \{v\}$ является планарным, и в этом графе число вершин меньше, чем в исходном. Значит, согласно индуктивной гипотезе в графе $G - \{v\}$ можно задать ориентацию ребер так, чтобы из каждой вершины исходило не более пяти дуг. Тогда ориентация ребер графа G будет устроена следующим образом: ребра, исходящие из вершин подграфа $G - \{v\}$, ориентируем так, как это допускается на основании индуктивной гипотезы, а все ребра, примыкающие к вершине v , превратим в дуги, исходящие из v . Легко видеть, что получившаяся в результате этого ориентация ребер будет ациклической и полустепень исхода каждой вершины не будет превосходить 5. $\square$

Теорема 17.20 (о шести красках). *Всякий планарный граф можно правильно раскрасить в шесть цветов.*

Доказательство. Рассмотрим произвольный планарный граф G и зададим в этом графе такую ациклическую ориентацию ребер, при которой полустепень исхода каждой вершины не превосходит 5. Так как ориентация ребер является ациклической, вершины графа можно так занумеровать натуральными числами, чтобы для любой дуги, ведущей из вершины v_j в вершину v_i , выполнялось неравенство $j > i$. Будем раскрашивать вершины в порядке возрастания номеров начиная с вершины v_1 и оканчивая вершиной v_n . Выбранная нами нумерация вершин позволяет быть уверенными в том, что, выбирая всякий раз подходящий цвет для вершины v , мы вынуждены будем считаться с цветами тех соседей вершины v , в которые ведут дуги, исходящие из v . Это обстоятельство, а также установленное нами ранее свойство выбранной ориентации ребер приводят нас к выводу о том, что прежде чем дело дойдет до выбора подходящего цвета для вершины v , раскрашенными окажутся не более пяти из соседних с ней вершин. Таким образом, для раскраски вершины v всегда остается возможность выбрать цвет, отличающийся от цветов, в которые раскрашены ее соседи. $\square$

В доказательстве теоремы о шести красках, по сути дела, предложен алгоритм шестичетной раскраски графа: на первом этапе согласно лемме 17.19 проводится ациклическая ориентация ребер, а на втором этапе проводится раскраска вершин в том порядке, который определяется введенной ориентацией ребер. Для представления ациклической ориентации в каждом процессе p вводится целочисленная переменная x_p ; мы будем считать, что ребро pq направлено из вершины p в вершину q , обозначая этот факт при помощи записи $\vec{pq}$, если

$$x_p < x_q \vee (x_p = x_q \wedge p < q).$$

Ясно, что каждое ребро может быть ориентировано таким образом только в одну из двух сторон (т. е. имеет место либо $\vec{pq}$, либо $\vec{qp}$) и подобная ориентация может быть только ациклической. Выражение $out(p)$ будет использоваться нами для обозначения числа дуг, исходящих из вершины p , т. е.

$$out(p) = \#\{q \in Neigh_p : \vec{pq}\}.$$

На первом этапе требуется добиться выполнения постуловия θ , которое задается следующей формулой:

$$\theta \equiv (\forall p : out(p) \leq 5).$$

Программа S_1 , предназначенная для выполнения постуловия θ , требует выполнения каждым процессом p одной и той же операции — ориентировать все входящие в вершину p дуги по направлению к p , если обнаруживается, что из вершины p исходит более пяти дуг:

$$\mathbf{D}_p : \{ out(p) > 5 \} \\ x_p := \max\{x_q : q \in Neigh_p\} + 1$$

Теорема 17.21. *Программа S_1 обладает способностью стабилизироваться на условии θ , и, после того как условие θ будет выполнено, значения переменных x_p остаются неизменными.*

Доказательство. Конфигурация становится заключительной, как только будет выполнено условие θ . Поэтому, как только программа завершает работу, постуловие будет выполнено, и если условие θ оказывается выполненным, то значения всех переменных x_p будут оставаться неизменными.

Для доказательства того, что программа S_1 рано или поздно остановит свою работу, мы будем использовать в качестве эталона ту ациклическую ориентацию, которая фигурирует в лемме 17.19; но это, однако, совсем не означает, что программа S_1 вычислит именно указанную ориентацию. Из леммы 17.19 следует существование такой нумерации вершин $v_1, v_2, \dots$, что у каждой из вершин есть не более пяти соседей, номера которых превосходят номер этой вершины. Назовем ребро $v_i v_j$ *неправильным*, если оно имеет ориентацию $v_i \vec{v}_j$, в то время как $i > j$, и воспользуемся записью $\hat{f}(\gamma)$ для обозначения списка целых чисел $(n_1, n_2, \dots)$, в котором n_i — это число неправильных ребер, инцидентных вершине v_i . При каждом выполнении действия $\mathbf{D}_p$ в этом списке происходят изменения, и преобразованный список будет предшествовать (в лексикографическом порядке) исходному списку. Рассмотрим выполнение указанного действия в узле v_j , и пусть

v_i — это та соседняя вершина с наименьшим номером, в которую ведет дуга из вершины v_j . Так как из вершины v_j исходит не менее шести дуг, но при этом у нее есть не более пяти соседей, номера которых превосходят номер v_j , выполняется неравенство $i < j$. Поэтому после выполнения действия $\mathbf{D}_p$ неправильное ребро $v_i v_j$ перестает быть таковым, и в результате этого значение компоненты списка n_i уменьшается, а значения всех компонент n_k , где $k < i$, остаются неизменными. $\square$

Программа S_2 , стремящаяся выполнить постуловие ψ , также требует выполнения каждым процессом одной и той же операции, в которой вершина p изменяет свой цвет в том случае, если прежняя окраска c_p совпадала с цветом одной из вершин q , следующих за p ; в качестве нового цвета вершина p выбирает тот цвет b , который не был использован вершинами r , следующей за p :

$$\mathbf{C}_p : \{ (\exists q : \vec{pq} \wedge c_p = c_q) \wedge (\forall r \text{ таких, что } \vec{pr} : c_r \neq b) \} \\ c_p := b$$

Теорема 17.22. *Если программа S_2 завершает свою работу и начальная конфигурация удовлетворяет условию θ , то заключительная конфигурация удовлетворяет условию ψ .*

Доказательство. Чтобы убедиться в том, что работа программы завершается, воспользуемся тем, что ориентация ребер является ациклической и поэтому существует такая нумерация вершин $v_1, v_2, \dots$, при которой номера вершин в любом ориентированном маршруте убывают. Для каждой конфигурации γ мы будем рассматривать двоичный список $g(\gamma) = (m_1, m_2, \dots)$, в котором элемент m_i равен 0, если цвет c_i вершины v_i отличен от цвета любой из тех вершин, которые следуют за v_i , и равен 1 в противном случае. Если в узле v_i совершилось действие $\mathbf{C}_p$, то значение элемента m_i изменится с 1 на 0 и при этом лишь те элементы m_j , для которых $j > i$, изменят значения с 0 на 1. Это означает, что значения функции g убывают в лексикографическом порядке по ходу выполнения программы.

Предположим теперь, что в начале работы программы было выполнено условие θ , и рассмотрим такую конфигурацию, в которой никакое действие $\mathbf{C}_p$ неприемлемо. У процесса p есть не более пяти последователей, и, следовательно, всегда есть такой цвет $b \in \{1, 2, 3, 4, 5, 6\}$, который не был использован ни одним из этих последователей. Коль скоро предохраняющее условие действия $\mathbf{C}_p$ ложно, окраска c_p отличается от окраски любого последователя процесса p .

Остается заметить, что для каждого ребра qr либо вершина r является последователем вершины q , либо вершина q является последователем вершины r . Следовательно, $c_r \neq c_q$. $\square$

Сопутствующая композиция программ S_1 и S_2 представлена как алгоритм 17.5.

Теорема 17.23. *Алгоритм 17.5 обладает способностью стабилизироваться на условии ψ .*

Доказательство. Сопутствующую композицию можно построить, потому что все те переменные, в которые программа S_2 осуществляет запись (это

```

var  $x_p$  : integer;
 $c_p$  : {1, 2, 3, 4, 5, 6};

 $\vec{p}\vec{q} \equiv x_p < x_q \vee (x_p = x_q \wedge p < q)$ ;
 $out(p) \equiv \#\{q \in Neigh_p : \vec{p}\vec{q}\}$ ;

 $D_p$ : {  $out(p) > 5$  }
 $x_p := \max\{x_q : q \in Neigh_p\} + 1$ 

 $C_p$ : {  $(\exists q : \vec{p}\vec{q} \wedge c_p = c_q) \wedge (\forall r \text{ s.t. } \vec{p}\vec{r} : c_r \neq b)$  }
 $c_p := b$ 

```

Алгоритм 17.5. Алгоритм раскраски планарного графа с использованием 6 красок

переменные c_p), отсутствуют в программе S_1 . Как было показано выше, программа S_1 обладает способностью стабилизироваться на условии θ и не изменяет значения переменных x_p , после того как будет достигнута выполнимость этого условия (теорема 17.21). Программа S_2 обладает способностью стабилизироваться на условии ψ , если начальная конфигурация удовлетворяет условию θ (теорема 17.22).

Теперь, чтобы прийти к заключению о том, что композиция $S_1 \succ S_2$ согласно теореме 17.17 обладает способностью стабилизироваться на условии ψ , остается убедиться в том, что эта композиция удовлетворяет требованию справедливости. Согласно теореме 17.22 между любыми двумя действиями программы S_1 программа S_2 может выполнить лишь конечное число действий; это влечет за собой справедливость по отношению к программе S_1 . Согласно теореме 17.21 программа S_1 завершает свою работу, и это влечет за собой справедливость по отношению к программе S_2 . $\square$

17.3.2. Вычисление минимальных путей

В этом параграфе представлены стабилизирующиеся алгоритмы для одного класса задач, в которых требуется найти путь наименьшей стоимости в том случае, когда функция стоимости определена подходящим образом на множестве путей. Примерами задач такого рода являются не только задачи непосредственного вычисления таблиц маршрутизации, но также задача построения деревьев поиска в глубину и даже, как это ни странно, задача избрания лидера (см. упражнение 17.12). Вначале дадим алгебраическую постановку задачи вычисления минимального пути и опишем один из алгоритмов ее решения (алгоритм 17.6), который обладает способностью стабилизироваться на минимальных путях. В конце параграфа мы рассмотрим некоторые приложения этого алгоритма.

Свойства функции стоимости и минимальных путей. Рассмотрим функцию D , сопоставляющую каждому пути в графе некоторый элемент линейно упорядоченного множества U ; для заданного пути π значение функции $D(\pi)$ мы бу-

дем называть *стоимостью* пути π . Стоимость *пустого пути* (p) длины 0 из вершины p в вершину p условимся обозначать символом c_p ; считается, что эта стоимость известна процессу, располагающемуся в вершине p . Мы ограничимся рассмотрением таких функций стоимости пути, которые могут вычисляться шаг за шагом; это означает, что стоимость непустого пути вычисляется по следующей схеме:

$$D(p_0, \dots, p_k, p_{k+1}) = f_{p_{k+1}p_k}(D(p_0, \dots, p_k)).$$

Здесь f_{pq} — это некоторая функция вида $f_{pq} : U \rightarrow U$, которая считается заданной для каждого ребра pq и называется *реберной функцией ребра*. Стоимость пути $\pi = (p_0, p_1, \dots, p_{k-1}, p_k)$ в таком случае определяется следующим выражением:

$$D(p_0, \dots, p_{k-1}, p_k) = f_{p_k p_{k-1}}(\dots (f_{p_1 p_0}(c_{p_0}) \dots)).$$

Условимся использовать запись $f_\pi(x)$ для обозначения выражения $f_{p_k p_{k-1}}(\dots (f_{p_1 p_0}(x) \dots))$; тогда стоимость пути $D(\pi)$ равна значению выражения $f_\pi(c_{p_0})$. Мы будем предполагать, что стоимость пути и реберные функции удовлетворяют следующим аксиомам.

1. Монотонность. Каково бы ни было ребро pq и элемент $x \in U$, если π — это простой путь, ведущий в вершину q и не проходящий через вершину p , то

$$D(\pi) < x \implies f_{pq}(D(\pi)) \leq f_{pq}(x).$$

2. Возрастание по циклу. Если путь π является циклом, то $f_\pi(x) > x$ для любого элемента $x \in U$.

3. Возрастание по длине. Существует такое число B , что для каждого пути ρ длины B или более, для каждого элемента $x \in U$ и для каждого простого пути π длины $N - 1$ или менее выполняется неравенство $f_\rho(x) > D(\pi)$.

Проблема минимальных путей состоит в том, чтобы уметь вычислять для любой заданной вершины p минимальную стоимость пути, ведущего в вершину p , а также определять ту вершину q , которая непосредственно предшествует вершине p в этом пути минимальной стоимости (в том случае, если минимальный путь является пустым, будем полагать $q = nil$). Стоимость минимального пути условимся обозначать записью $\chi(p)$, а для обозначения предшественника вершины p в минимальном пути будем использовать запись $\varphi(p)$. Следующая теорема гарантирует существование минимальных путей и описывает их основные свойства.

Теорема 17.24. 1. Для каждой вершины p существует такой простой путь $\pi(p)$, оканчивающийся в вершине p , что стоимость любого пути, оканчивающегося в вершине p , не превосходит $D(\pi(p))$.

2. Существует такой остовный лес (который называется лесом минимальных путей), что для каждой вершины p путь (единственный), ведущий из одного из корней этого леса в вершину p имеет стоимость $D(\pi(p))$.

Доказательство. 1. Поскольку имеется лишь конечное число простых путей, оканчивающихся в вершине p , мы можем взять в качестве $\pi(p)$ произвольный простой путь минимальной стоимости, оканчивающийся в вершине p .

Согласно выбору $\pi(p)$ все простые пути, оканчивающиеся в вершине p , имеют стоимость, не меньшую чем $D(\pi(p))$. Стоимость всякого пути, в котором содержится цикл, превосходит (согласно аксиоме возрастания по циклу и аксиоме монотонности) стоимость того простого пути, который образуется после удаления циклов. Таким образом, величина $D(\pi(p))$ является точной нижней гранью стоимости всякого пути, ведущего в вершину p , и путь $\pi(p)$ является искомым путем.

2. Поскольку может быть несколько разных минимальных путей, нужно внимательно отнестись ко второй части теоремы: граф, образованный всеми теми ребрами, которые содержатся хотя бы в одном минимальном пути, может не быть лесом. Так же как и оптимальное входное дерево, фигурирующее в теореме 4.2, лес минимальных путей может быть не единственным, но один из таких лесов можно построить, руководствуясь следующими соображениями.

Для каждой вершины p родительскую вершину $\varphi(p)$ будем выбирать так: если пустой путь (p) является оптимальным, то положим $\varphi(p) = nil$, а в противном случае в качестве $\varphi(p)$ возьмем предпоследнюю вершину q , которая встречается в одном из минимальных путей, ведущих в p . Рассмотрим граф, ребра которого образованы всеми парами вида $(p, \varphi(p))$, в которых $\varphi(p) \neq nil$.

Из аксиомы монотонности следует, что для каждой вершины p в том случае, если $\varphi(p) \neq nil$, имеет место неравенство $\chi(p) \geq f_{p\varphi(p)}(\chi(\varphi(p)))$. Значит, если C — это цикл, проходящий в образовавшемся графе через вершину p , то $\chi(p) \geq f_C(\chi(p))$, а это неравенство противоречит аксиоме возрастания по циклу. Поэтому построенный нами граф является лесом.

Далее индукцией по глубине можно установить требуемое свойство этого леса.

□

Заметим, что в приведенном нами доказательстве не было использовано свойство возрастания по длине; оно будет нам нужно только для того, чтобы убедиться в том, что ошибочная информация, которая может содержаться в начальной конфигурации, в конце концов будет исправлена стабилизирующимся алгоритмом вычисления минимальных путей.

Постусловие, которое должно быть выполнено алгоритмом решения проблемы минимальных путей, теперь можно сформулировать так:

$$\psi \equiv (\forall p : K_p = \chi(p) \wedge L_p = \varphi(p)).$$

Классический (нестабилизирующийся) алгоритм, удовлетворяющий постуловию ψ , выполняет для каждого ребра только одну операцию, которая называется *проталкиванием* ребра:

$$\mathbf{P}_{pq}: \{ f_{pq}(K_q) < K_p \} \\ K_p := f_{pq}(K_q); L_p := q.$$

В начале вычисления переменная K_p имеет значение c_p . Можно убедиться в том, что утверждение $K_p \geq \chi(p)$ является инвариантом. Чтобы убедиться в выполнении свойства живучести, мы должны предполагать, что проталкивание проводится справедливо по отношению к каждому ребру, т.е. в каждом бесконечном

выполнении каждое ребро проталкивается бесконечно часто. То, что неравенство $K_p \leq \chi(p)$ будет выполнено, можно показать, воспользовавшись индукцией по числу вершин в минимальном пути, ведущем в вершину p ; читатель может обнаружить определенное сходство с алгоритмом Чанди—Мисры вычисления минимальных путей (алгоритм 4.7).

Алгоритм Update. Чтобы суметь стабилизироваться на условии ψ , информация о существующих путях должна проталкиваться по ребрам и растекаться по всему графу. Однако при этом ошибочную информацию, которая могла содержаться в начальной конфигурации, нужно удалить из системы; эта информация представлена теми значениями K_p , для которых не существует соответствующего пути, ведущего в вершину p . Это достигается за счет перемещения этой информации по все более и более протяженным путям, до тех пор пока, вследствие аксиомы возрастания по длине, ошибочная информация в конце концов не будет отвергнута в пользу тех данных, которые относятся к существующему пути.

```
var  $K_p$  :  $D$  ;
     $L_p$  :  $Neigh_p \cup \{nil\}$  ;
```

```
 $C_p$ :  $K_p := \min(c_p, \min\{f_{pq}(K_q) : q \in Neigh_p\})$  ;
    if  $K_p = c_p$  then  $L_p := nil$ 
    else  $L_p := q$  так, чтобы  $K_p = f_{pq}(K_q)$ 
```

Алгоритм 17.6. Алгоритм Update для минимальных путей

В каждом узле алгоритм Update выполняет одно и то же действие, вычисляющее наименьшую из величин, представляющих стоимость пустого пути для данной вершины и стоимость тех путей, которые хранятся в памяти соседних узлов (см. алгоритм 17.6). Проводя анализ этого алгоритма, мы будем исходить из предположения о справедливости его выполнения по отношению ко всем процессам, как мы это делали при анализе алгоритма проталкивания, т.е. в каждом выполнении каждый процесс срабатывает бесконечно часто.

Теорема 17.25. Алгоритм Update стабилизируется на минимальных путях.

Доказательство. Так как действие C_p может быть выполнено всегда, каждое выполнение продолжается бесконечно долго. Допущение справедливости позволяет нам разбить выполнение на *туры* следующим образом. Первый тур завершается, когда каждый процесс выполнил хотя бы одно действие; $(i+1)$ -й тур завершается, когда каждый процесс сработал хотя бы один раз после завершения i -го тура.

Прежде всего мы покажем, что в конечном итоге значение переменной K_p будет составлять нижнюю оценку величины $\chi(p)$. Условимся считать, что запись $\chi_i(p)$ будет обозначать минимальную стоимость среди всех путей, ведущих в вершину p и имеющих длину, не превосходящую числа i . Мы утверждаем, что

в любой момент времени после окончания i -го тура выполняется неравенство $K_p \leq \chi_{i-1}(p)$. Для обоснования этого утверждения воспользуемся индукцией по i .

Базис: $i = 1$. Пустой путь, имеющий стоимость c_p , — это единственный путь длины 0, ведущий в вершину p , и, как видно из описания алгоритма, после выполнения первых C_p шагов имеет место неравенство $K_p \leq c_p$.

Индуктивный переход: $i \rightarrow i + 1$. Рассмотрим путь π , имеющий наименьшую стоимость среди всех путей, ведущих в вершину p , длина которых не превосходит числа i . Если путь π имеет длину 0, то неравенство $K_p \leq D(\pi)$ соблюдается после выполнения каждого шага процесса p и, следовательно, будет оставаться верным и по окончании $(i + 1)$ -го тура. Предположим, что π — непустой путь, который получен присоединением к пути ρ ребра qr . Тогда длина пути ρ не превосходит величины $i - 1$. Согласно индуктивному предположению неравенство $K_q \leq D(\rho)$ будет верно по окончании i -го тура. Отсюда следует, что, после того как завершится i -й тур, неравенство $K_p \leq f_{pq}(D(\rho)) = D(\pi)$ будет оставаться верным после каждого шага процесса p . К концу $(i + 1)$ -го тура процесс p должен совершить хотя бы один такой шаг, и это приводит к тому, что неравенство $K_p \leq D(\pi)$ будет оставаться верным после окончания этого тура.

Так как простой путь не содержит циклов, его длина не превосходит величины $N - 1$, и отсюда следует, что $\chi(p) = \chi_{N-1}(p)$. Поэтому неравенство $K_p \leq \chi(p)$ будет оставаться верным по окончании N -го тура.

Чтобы убедиться в том, что значение переменной K_p в конечном итоге будет составлять верхнюю оценку величины $\chi(p)$, мы покажем, что либо по окончании i -го тура значение K_p будет характеризовать один из существующих путей, либо оно образовалось из начальной информации, прошедшей не менее чем по i звеньям. Обозначим символом K_r^* начальное значение переменной K_r . Мы утверждаем, что если $K_p = K$, то либо существует путь π , ведущий в вершину p , для которого имеет место равенство $D(\pi) = K$, либо существует путь ρ , ведущий из вершины r в вершину p , для которого имеет место равенство $K = f_\rho(K_r^*)$. Кроме того, если значение K вычислено в i -м туре, то длина пути ρ будет не меньше i . Обоснование проведем при помощи индукции по числу шагов выполнения алгоритма.

Базис индукции. В самом начале выполнения мы имеем равенство $K_r = f_{(r)}(K_r^*)$, где (r) — это путь длины 0; при этом, конечно же, K_r может как соответствовать, так и не соответствовать существующему пути.

Индуктивный переход. Рассмотрим значение K , вычисленное на очередном шаге процессом p . Если $K = c_p$, то это стоимость существующего пути (длины 0), ведущего к вершине p , и наше утверждение верно. В противном случае имеет место равенство $K = f_{pq}(K_q)$. Тогда согласно индуктивному предположению, верна одна из двух альтернатив: либо 1) K_q — это стоимость пути π , ведущего в вершину q , либо 2) существует такой путь ρ из вершины r в вершину q , для которого выполняется равенство $K_q = f_\rho(K_r^*)$. В первом случае мы можем построить новый путь, добавив к существующему пути π ребро qr , и в результате получим путь, стоимость которого будет равна $f_{pq}(D(\pi)) = f_{pq}(K_q) = K$. Отсюда следует справедливость нашего утверждения. Во втором случае, добавив к пути ρ ребро qr , мы получим новый путь ρ' , для которого будут выполнены равенства

$K = f_{pq}(f_\rho(K_r^*)) = f_{\rho'}(K_r^*)$. Заметим также, что если значение K было вычислено в i -м туре, то значение K_q должно быть вычислено либо в $(i - 1)$ -м туре, либо позднее. Согласно индуктивному предположению длина пути ρ должна быть не меньше $i - 1$, и поэтому длина пути ρ' будет не меньше i .

Теперь для завершения доказательства мы воспользуемся аксиомой возрастания по длине. По окончании B -го тура значение выражения $f_\rho(K_r^*)$ будет превосходить стоимость любого существующего в графе простого пути, и поэтому величина K_p будет ограничена снизу стоимостью некоторого существующего пути, ведущего в вершину p . Следовательно, по окончании B -го тура для каждой вершины p будет выполнено равенство $K_p = \chi(p)$. Нетрудно убедиться в том, что значение L_p удовлетворяет всем тем требованиям, которые предъявляются к вершине $\varphi(p)$. Отсюда следует, что ребро (p, L_p) будет входить в состав леса минимальных путей. $\square$

Если функция стоимости путей не обладает свойством возрастания по длине, то это свойство может быть навязано ей при условии, что размер сети N известен заранее. Для этого достаточно немного видоизменить стоимость пути, включив туда длину пути: сравнивая друг с другом стоимости двух путей, мы будем полагать, что стоимость всякого пути длины N или более всегда превосходит стоимость любого пути длины $N - 1$ или менее. Чтобы описать указанную модификацию более формально, рассмотрим некоторую заданную функцию стоимости D . На ее основе определим функцию стоимости D' при помощи равенства $D'(\pi) = (D(\pi), |\pi|)$. Теперь стоимость всякого пустого пути, ведущего в вершину p , будет задаваться парой $c'_p = (c_p, 0)$, а стоимости ребер будут задаваться парами $f'_{pq}(C, k) = (f_{pq}(C), k + 1)$. Для сравнения стоимостей двух путей мы будем использовать следующее соотношение:

$$(C_1, k_1) <' (C_2, k_2) \iff (k_1 < N \wedge k_2 \geq N) \vee (C_1 < C_2).$$

На множестве всех простых путей функция D' определяет в точности такой же порядок, какой был определен функцией D . Отсюда следует, что новая функция стоимости сохраняет свойства монотонности возрастания по циклу. И поскольку все пути, длина которых меньше N , имеют меньшую стоимость, нежели пути, длина которых больше или равна N , свойство возрастания по длине будет также выполнено.

Приложение: маршрутизация. Алгоритм Update находит непосредственное применение для решения задачи вычисления таблиц маршрутизации; так же как и в алгоритме Netchange, вычисление проводится отдельно для каждого узла-адресата. Пусть задан некоторый узел-адресат v , и будем полагать значение $D(\pi)$ равным весу маршрута π (который равен сумме весов всех ребер, входящих в состав π), если π — это путь, начинающийся в узле v , и равным ∞ в противном случае. Эту функцию можно вычислять шаг за шагом, полагая значение c_p равным 0, если $p = v$, и равным ∞ во всех остальных случаях, а также используя соотношение $f_{pq}(C) = C + \omega_{pq}$ (где ω_{pq} — это вес ребра pq).

Монотонность является следствием аддитивности функций ребра, а свойство возрастания по циклу вытекает из допущения о том, что все циклы в сети имеют

положительную стоимость. Чтобы убедиться в выполнимости свойства возрастания по длине, заметим, что в сети может быть лишь конечное число простых циклов. Пусть δ — это наименьший вес пустого цикла, а ω — это наибольший вес, который может быть у ребер сети. Тогда путь длины $N - 1$ или менее имеет вес, не превосходящий величины $N\omega$, в то время как всякий путь длины $B = N(N\omega)/\delta$ содержит более $(N\omega)/\delta$ простых циклов и имеет вес, превосходящий величину $N\omega$.

Следовательно расстояние до вершины v (а также наиболее предпочтительно-го соседа в кратчайшем пути по направлению к вершине v) можно вычислить при помощи алгоритма Update. Чтобы полностью вычислить таблицу маршрутизации, алгоритм нужно применять параллельно по отношению к каждому узлу-адресату.

Приложение: дерево поиска в глубину. Теперь мы покажем, что, прокладывая по направлению к каждому процессу лексикографически наименьший простой путь, мы получим остовное дерево поиска в глубину, корнем которого будет процесс с наименьшим именем. Таким образом, дерева поиска в глубину можно построить при помощи алгоритма Update.

Стоимость пути π будем считать список вершин, образующих этот путь, т. е. $c_p = (p)$, и значение $f_{pq}(\sigma)$ будем полагать равным списку, образованному за счет присоединения к списку σ вершины p . Сравнение стоимостей проводится на основании лексикографического порядка, но при этом считается, что простые пути всегда меньше путей, содержащих хотя один цикл:

$$\sigma < \tau \iff (\sigma \text{ — простой путь, } \tau \text{ — нет}) \vee (\sigma <_L \tau).$$

Лексикографический порядок обеспечивает выполнимость свойства монотонности, а условие возрастания по циклу соблюдается, ввиду того что стоимость пути, содержащего цикл, всегда превосходит стоимость простого пути. Свойство возрастания по длине также соблюдается, потому что пути длины N или более содержат циклы, и стоимость их превосходит стоимость простых путей. Следовательно, алгоритм Update может вычислять простые пути наименьшей стоимости, ведущие к каждой вершине. Остается лишь проанализировать свойства полученного в результате его применения леса минимальных путей.

Во-первых, заметим, что этот лес состоит из одного-единственного дерева, корнем которого является узел с наименьшим именем; обозначим этот узел символом r . Действительно, для каждого узла p существует простой путь из узла r в узел p ⁶⁾. Простой путь из узла r имеет меньшую стоимость, чем путь из всякой другой вершины до узла p , и поэтому путь минимальной стоимости, ведущий в вершину p , начинается в узле r .

Во-вторых, для любых двух соседних узлов p и q либо префиксом x_p является вершина q , либо префиксом x_q является вершина p . Действительно, рассмотрим списки $x(p)$ и $x(q)$ и предположим, не ограничивая общности, что $x(p) < x(q)$. Так как пути минимальной стоимости — это простые пути, выполняется неравенство $x(p) <_L x(q)$. Отсюда следует, что вершина q не содержится в списке $x(p)$, поскольку в противном случае начальная часть этого списка до самой вершины q

⁶⁾При условии, что сеть является связной. — Прим. перев.

соответствовала бы пути в вершину q и при этом лексикографически предшествовала бы списку $x(q)$ вопреки определению $x(q)$. Но в таком случае, добавив к списку $x(p)$ вершину q , мы получили бы список x' , который соответствует стоимости *простого* пути, ведущего в вершину q , и это означало бы, что $x(q) \leq_L x'$. Таким образом, $x(p) <_L x(q) \leq_L x'$, откуда следует, что $x(p)$ является префиксом $x(q)$, или, что то же самое, вершина p является предшественником вершины q в дереве минимальных путей.

Указанные два свойства, а именно то, что

- 1) лес состоит из одного единственного дерева и
- 2) для любых двух соседних узлов p и q , либо префиксом x_p является вершина q , либо префиксом x_q является вершина p ,

приводят нас к выводу о том, что лес минимальных путей, вычисляемый алгоритмом Update, является деревом поиска в глубину. Описанное здесь приложение алгоритма Update было предложено Херманом в работе [103]; ему удалось также так обобщить этот алгоритм, чтобы использовать его для вычисления разделяющих вершин, при удалении которых сеть перестает быть связной.

17.3.3. Заключительные выводы и обсуждение

В этом параграфе мы изучили две стратегии, которые оказываются полезными для построения стабилизирующихся алгоритмов. Сопутствующую композицию можно применять для построения алгоритмов, работу которых можно разбить на два (и даже большее число) следующих друг за другом этапов, а алгоритм Update можно применять для решения тех задач, которые можно сформулировать как проблему поиска минимального пути, ведущего к каждому процессу.

Существуют методы, позволяющие автоматически превращать произвольный алгоритм в стабилизирующийся алгоритм, который при этом удовлетворяет той же самой спецификации. Для этого Катц и Перри в работе [114] предложили использовать следующий метод. Параллельно с выполнением заданного классического алгоритма P система постоянно вычисляет моментальные состояния P и проверяет, не была ли уже достигнута требуемая конфигурация для алгоритма P . Если этого еще не произошло, то протокол перезапуска возвращает системе в какое-нибудь начальное состояние. Однако при воплощении этого замысла приходится сталкиваться с множеством технических трудностей, обусловленных необходимостью вычислять моментальные состояния и проводить перезапуск системы.

17.4. Упражнения к главе 17

17.4.1

Упражнение 17.1. Докажите лемму 17.4.

Упражнение 17.2. Докажите, что кольцевому алгоритму с маркерами, предложенному Дейкстрой, требуется совершить $O(N^2)$ шагов для достижения легитимной конфигурации. Упростите анализ алгоритма, воспользовавшись толь-

ко одной функцией нормировки, квадратично зависящей от N , которая убывает с *каждым* шагом алгоритма.

Упражнение 17.3. Докажите, что алгоритм Дейкстры стабилизируется уже тогда, когда $K \geq N$ (а не только тогда, когда $K > N$, как это утверждалось нами при проведении анализа этого алгоритма). См. [106].)

17.4.2

Упражнение 17.4. Внесите в алгоритм ориентации кольца (алгоритм 17.1) необходимые изменения, для того чтобы этот алгоритм мог останавливаться, как только постуловие оказывается выполненным.

Курсовая работа 17.5. Разработайте стабилизирующийся алгоритм ориентации тором, стараясь добиться того, чтобы этот алгоритм завершал свои выполнения.

Упражнение 17.6. Покажите, что алгоритм построения максимального покрытия можно реализовать *униформно* в модели со сцепленными регистрами и параллельным считыванием.

Упражнение 17.7. Покажите, что алгоритм построения максимального покрытия стабилизируется спустя $N^2 + O(N)$ шагов. Покажите также, что нижняя оценка сложности алгоритма в наихудшем случае составляет величину, равную $\Omega(N^2)$.

Упражнение 17.8. Постройте стабилизирующийся алгоритм вычисления максимального независимого множества и определите, сколько шагов требуется совершить этому алгоритму, пока не будет достигнута стабилизация.

Курсовая работа 17.9. При помощи какой функции нормировки можно доказать завершаемость алгоритма 17.4.

17.4.3

Курсовая работа 17.10. Докажите, что внешне планарные графы можно раскрасить в три цвета. Постройте стабилизирующийся алгоритм вычисления правильной раскраски внешне планарных графов в три цвета.

Курсовая работа 17.11. Постройте стабилизирующийся алгоритм раскраски планарных графов в пять цветов. (См. [145] или какой-нибудь другой учебник, в котором рассказывается об алгоритмах на графах.)

Упражнение 17.12. Покажите, что алгоритм Update можно применять для избрания лидера и вычисления остовного дерева поиска в ширину, выбрав для этого подходящую функцию стоимости пути.

Упражнение 17.13. Постройте стабилизирующийся алгоритм вычисления размера сети.

Упражнение 17.14. Покажите, как можно вычислить глубину дерева при помощи алгоритма Update.

ПРИЛОЖЕНИЕ А

СОГЛАШЕНИЯ ОБ ИСПОЛЬЗОВАНИИ ПСЕВДОКОДА

Вместо того чтобы давать описания алгоритмов на одном из существующих языков программирования, в этой книге мы используем для этой цели псевдокод. Псевдокод является строгим и вместе с тем «дружественным» языком, в то время как реальные языки программирования вынудили бы автора пожертвовать ясностью изложения в угоду жестко ограниченному синтаксису.

Паскалеподобный псевдокод. Используемый в этой книге псевдокод очень сильно напоминает язык Паскаль. Описание распределенного алгоритма обычно ограничивается описанием локального алгоритма отдельного процесса системы, и имя этого процесса используется в качестве индекса, которым помечены переменные этого алгоритма. Имена процессов могут также применяться в качестве индексов массива; помимо массивов в алгоритмах используются переменные, значениями которых являются множества. Значением переменной $Neigh_p$ является множество процессов, с которыми связан процесс p (эти процессы называются *соседями* процесса p).

Оператор присваивания « $a := expression$ », оператор ветвления (условный оператор) «**if condition then statement**» с необязательной ветвью «**else statement**» и оператор итерации (оператор цикла) «**while condition do statement**» имеют тот же самый смысл, что и в языке Паскаль. Оператор **forall** имеет следующий вид «**forall $x \in X$ do statement**», где x — это формальный параметр, а X — множество; смысл этого оператора таков: оператор *statement* будет поочередно выполняться для каждого элемента x из множества X . В этом операторе особо не оговаривается тот порядок, в котором выбираются элементы из множества X , но сам оператор применяется лишь в том случае, когда конкретный порядок не играет никакой роли. Например, если мы хотим, чтобы у каждого соседа счетчик полученных сообщений был обнулен, то для этого применяется такой оператор:

forall $q \in Neigh_p$ do $rec_p[q] := 0$.

Последовательность операторов, разделенных точкой с запятой и окаймленная ключевыми словами **begin** и **end**, расценивается как один оператор. Пустые строки и отступы служат для разбиения программ на блоки; этим средствам форматирования текста программы не придается никакого особого смыслового значения.

Передача сообщений. Операции «**send**» и «**receive**» применяются в псевдокоде для описания передачи сообщений. Сообщение состоит из спецификатора *типа сообщения* и нескольких (быть может, ни одного) *полей данных*, прису-

щих этому типу сообщений. Предположим, например, что каждый процесс должен передать свое имя (отличительные признаки) и свою степень (количество смежных узлов) своим соседям. Для этого можно воспользоваться сообщением типа **info** с двумя полями данных, в которых записаны имя и степень отправителя. Для обозначения такого сообщения применяется запись $\langle \text{info}, a, b \rangle$.

Оператору отправления сообщения нужно указать само сообщение (его тип и значения полей данных), а также тот узел, для которого предназначено сообщение (соседа отправителя). Значение каждого поля данных задается выражением, значение которого вычисляет сам процесс-отправитель. Процессу p , чтобы отправить свое имя и свою степень процессу q , нужно выполнить оператор

send $\langle \text{info}, id_p, |Neigh_p| \rangle$ to q .

Если нужно отправить некоторое сообщение всем соседям данного процесса, то мы будем использовать сокращенную запись «shout», т.е. оператор «shout $\langle \text{info}, id_p, |Neigh_p| \rangle$ » следует понимать как процедуру

forall $q \in Neigh_p$ do send $\langle \text{info}, id_p, |Neigh_p| \rangle$ to q .

Чтобы принять сообщение, нужно задать его тип и формальные параметры для полей данных этого сообщения, а также указать имя отправителя сообщения. Чтобы получить информацию об имени и степени соседнего узла, процесс p может выполнить оператор

receive $\langle \text{info}, a, b \rangle$ from q .

В результате выполнения этого оператора одно сообщение типа **info** изымается из коммуникационной системы; при этом первое и второе поле сообщения, а также имя отправителя сообщения становятся значениями переменных a , b и q соответственно. В операторе приема сообщений, представленном в такой форме, не указывается, от какого соседа должно быть получено сообщение, но после выполнения операции приема сообщения имя его отправителя становится известно.

Может случиться так, что процессу нужно принять сообщения от некоторого определенного процесса, например, от процесса q_0 ; в этом случае используется оператор

receive $\langle \text{info}, a, b \rangle$ from this q_0 .

Иногда бывает необходимо предусмотреть, чтобы могли быть приняты сообщения разных типов, и последующие действия при этом могут зависеть от типа полученного сообщения. Это можно выразить так (см. алгоритмы 7.7 и 7.9):

receive m from q ; (* or: fromthis q *)
if m is $\langle \text{info}, a, b \rangle$ then *statement* else *statement*

Параметр q , который используется процессом p для указания того соседа, которому предназначено сообщение или от которого ожидается поступление сообщения, называется *адресом* этого соседа. В случае *прямой адресации* адрес процесса — это имя процесса q ; это означает, что все соседи процесса q используют один и тот же адрес при обращении к процессу q . В случае *косвенной адресации*

адрес — это локальное имя, известное процессу p ; тогда соседи процесса q могут использовать различные адреса при обращении к процессу q (см. рис. 2.5).

Связь между процессами может быть *асинхронной* или *синхронной* (см. § 2.5.1). Если поддерживается асинхронная связь (это будет всегда подразумеваться, если не оговорено противное), оператор отправления сообщения может быть всегда немедленно выполнен. Отправленное сообщение попадает в коммуникационную подсистему и остается в ней, до тех пор пока оно не будет изъято оттуда в результате выполнения соответствующего оператора приема сообщения. Оператор приема сообщения приостанавливает выполнение процесса, до тех пор пока сообщение заданного типа не будет доступно для приема. Если поддерживается синхронная связь, то никакое временное хранение отправленного сообщения невозможно. В этом случае обе операции отправления и приема сообщения могут приостанавливать выполнение процесса, потому что выполнение операции отправления и соответствующей операции приема сообщения синхронизованы. Кто бы ни выполнил первым свою операцию, процесс-отправитель или процесс-получатель, его работа приостанавливается, до тех пор пока оба процесса не будут готовы выполнить операцию. Вот тогда происходит обмен сообщением, и после этого оба процесса продолжают выполнение своих алгоритмов.

Алгоритмы с потоком управления и с реакцией на события. Есть два способа представления управляющей структуры алгоритма: либо явно с описанием *потока управления*, либо неявно с указанием *реакции на события*. Для описания каждого алгоритма можно использовать любую из альтернатив, но во многих случаях одна из этих альтернатив оказывается предпочтительнее другой.

var $rec_p[q]$ for each $q \in Neigh_p$: **bool** init *false* ;

begin while $\#\{q : rec_p[q] \text{ is false}\} > 1$ **do**
 begin receive **(tok)** from q ; $rec_p[q] := \text{true}$ **end** ;
 send **(tok)** to q_0 with $rec_p[q_0]$ is *false* ;
 receive **(tok)** from q_0 ; $rec_p[q_0] := \text{true}$;
 decide
end

Алгоритм А.1. Древесный волновой алгоритм (описание потока управления)

При использовании потока управления описание алгоритма состоит из декларативной части, в которой определяются переменные, и списка операторов, разделенных точкой с запятой и оканчиваемых ключевыми словами **begin** и **end**. В этом случае выполнение локального алгоритма предполагает однократное выполнение всего указанного списка операторов. В самом начале выполнения переменные могут быть инициализированы, но допускается также возможность инициализации переменных при их определении.

В качестве примера рассмотрим алгоритм А.1, который представляет собой один из вариантов древесного волнового алгоритма (см. алгоритм 6.2), при опи-

сании управляющей структуры которого использовался поток управления. Так как список операторов целиком задает поток управления локального алгоритма, недетерминизм должен быть явно указан в самой программе. В алгоритме А.1 недетерминизм присутствует в операторе приема сообщений, поскольку соответствующий отправитель явно не указывается. Труднее выразить недетерминированный выбор между отправлением и приемом сообщений, но и это можно сделать, если есть возможность обращаться к каналам связи и проверять, не содержится ли в них сообщения:

```

if a message  $\langle \text{info}, a, b \rangle$  is available on channel  $q$ 
  then begin receive  $\langle \text{info}, a, b \rangle$  ; ... end
  else begin send ... end

```

Гораздо проще выразить недетерминизм, указывая, какова должна быть реакция алгоритма на те или иные события. В этом случае описание алгоритма состоит из декларативной части, в которой определяются переменные, и списка *действий*. Каждое действие предваряется некоторым булевым выражением, которое называется *предохранителем* действия), и некоторым оператором, который называется *телом*. Действие считается *разрешенным* (или *применимым*), когда его предохранитель принимает значение *true*. Тела действий могут выполняться сколь угодно часто и в произвольном порядке. Есть только одно ограничение — действие выбирается для выполнения лишь тогда, когда оно разрешено.

```

var  $rec_p[q]$  for each  $q \in Neigh_p$  : bool init false ;
     $sent_p, dec_p$  : bool init false ;

```

```

 $R_p$ :{ Поступило сообщение  $\langle \text{tok} \rangle$  }
  begin receive  $\langle \text{tok} \rangle$  from  $q$  ;  $rec_p[q] := \text{true}$  end

```

```

 $S_p$ :{  $\#\{q : rec_p[q] \text{ is false}\} = 1$  and  $sent_p = \text{false}$  }
  begin send  $\langle \text{tok} \rangle$  to  $q_0$  with  $rec_p[q_0]$  is false ;
     $sent_p := \text{true}$ 
  end

```

```

 $D_p$ :{  $\#\{q : rec_p[q] \text{ is false}\} = 0$  and  $dec_p = \text{false}$  }
  begin decide ;  $dec_p := \text{true}$  end

```

Алгоритм А.2. Древесный волновой алгоритм (описание потока событий)

В качестве примера рассмотрим алгоритм А.2, который представляет собой не что иное, как тот же самый древесный алгоритм, в описании которого указывается реакция на возможные события. Поскольку каждый оператор может быть выбран для выполнения сколь угодно часто (при условии, что его предохранитель принимает значение *true*), нужно заранее указать в программе, что оператор отправления сообщения и оператор *decide* выполняются только один раз. В алгоритме А.2 это делается при помощи флагов $sent_p$ и dec_p .

Мы не выдвигаем никаких допущений справедливости применительно к программам, реагирующим на события; это означает, что мы не требуем от модели выполнения, чтобы то действие, которое разрешено бесконечно долго, обязательно в конце концов выполнялось. Таким образом, самому программисту вменяется в обязанность добиться того, чтобы ни одно существенное действие не пострадало от удушения (т.е. никогда не выбиралось по той причине, что всякий раз предпочтение отдавалось другим действиям программы).

ПРИЛОЖЕНИЕ Б

ГРАФЫ И СЕТИ

Поскольку топология распределенной системы обычно моделируется графом, при изучении распределенных алгоритмов полезно знать некоторые сведения из теории графов и уметь пользоваться терминологией этой теории. Кроме того, графы иногда используются в доказательствах и математических обоснованиях, касающихся распределенных алгоритмов (см., например, § 8.5.2).

В § Б.4.1 приводятся общие определения и термины из теории графов, а в § Б.4.2 рассказывается о некоторых специальных классах графов.

Б.1. Определения и термины

Граф задается совокупностью точек (они называются *вершинами* или *узлами* графа), некоторые из которых соединены друг с другом линиями (они называются *ребрами*); см. рис. Б.1. В случае ориентированных графов ребра являются направленными (ориентированными), и направление (ориентация) обозначается стрелкой. В случае взвешенных графов каждому ребру приписывается некоторая числовая величина.

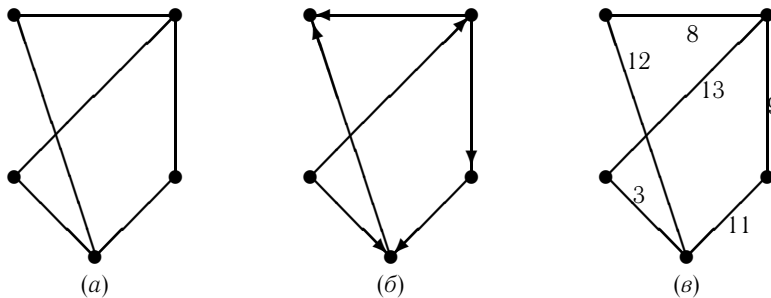


Рис. Б.1. Графы: неориентированный (а), ориентированный (б) и взвешенный (в)

Б.1.1. Неориентированные графы

Неориентированным графом G называется пара (V, E) , в которой множество V называется *множеством вершин* графа G , а множество E представляет собой совокупность пар элементов из множества V . Иначе говоря, каждый элемент из множества E представляет собой пару $\{u, v\}$, где $u, v \in V$. Чтобы упростить

обозначения, мы будем использовать запись $uv \in E$ вместо $\{u, v\} \in E$. Поскольку каждая пара является неупорядоченной, запись $uv \in E$ эквивалентна записи $vu \in E$. Ребро uv называется ребром, *инцидентным* вершине u (а также вершине v). Если $uv \in E$, то вершины u и v называются *смежными* или *соседними*. *Степенью* вершины называется число инцидентных ей ребер, или, по другому, число соседних вершин. *Регулярным* графом (степени δ) называется граф, все вершины которого имеют одну и ту же степень (δ).

Путь длины k из вершины v_0 в вершину v_k называется такая последовательность вершин $P = \langle v_0, \dots, v_k \rangle$, в которой для каждого $i < k$ выполняется включение $v_i v_{i+1} \in E$. В этом пути вершина v_0 считается начальной вершиной, а вершина v_k — конечной вершиной. *Циклом* называется всякий путь, начальная и конечная вершины которого совпадают. Путь называется *простым*, если все его вершины, начиная от v_0 и оканчивая v_k , попарно различны. Цикл называется *простым*, если все его вершины начиная от v_1 и оканчивая v_k попарно различны.

Расстоянием между вершинами u и v называется длина кратчайшего пути между ними; для обозначения расстояния будем использовать запись $d(u, v)$. *Диаметром* графа G называется максимальное расстояние между его вершинами. Неориентированный граф называется *связным*, если между любыми двумя вершинами этого графа существует путь. Неориентированный граф называется *ациклическим*, если в нем нет простых циклов длины три и более ¹⁾.

Граф $G' = (V', E')$ называется *подграфом* графа G , если $V' \subseteq V$ и $E' \subseteq E$. Подграф G' называется *остовным*, если $V' = V$. Говорят, что G' является *индуцированным подграфом* графа G , если $E' = \{uv \in E | u \in V' \wedge v \in V'\}$. *Компонентой связности* графа G называется максимальный связный индуцированный подграф G' графа G ; иными словами, G' является связным индуцированным подграфом, но, если к множеству V' добавить хоть одну вершину, индуцированный подграф перестанет быть связным. Компоненты задают разбиение графа, и это разбиение состоит из одной компоненты тогда и только тогда, когда граф является связным.

Граф называется *планарным*, если его можно нарисовать на плоскости так, чтобы его ребра не пересекались. Граф называется *внешне планарным*, если его можно нарисовать на плоскости так, чтобы все вершины располагались по периметру рисунка, а ребра не пересекались. Интерес представляют следующие свойства планарных и внешне планарных графов.

1. Всякий планарный граф с N вершинами имеет не более $3N - 6$ ребер.
2. Всякий планарный граф имеет хотя бы одну вершину степени пять или менее.
3. Всякий внешне планарный граф имеет хотя бы одну вершину степени два или менее.

¹⁾Учитывая, что E является множеством, а не мультимножеством, и пара вершин, образующая ребро, является неупорядоченной парой, мы заключаем, что в любом графе отсутствуют циклы длины 2. — Прим. перев.

Б.1.2. Ориентированные графы

Ориентированным графом G называется пара (V, E) , где V — это множество вершин, а E — это совокупность упорядоченных пар вершин из множества V . Элементами множества E (они называются *дугами*) являются упорядоченные пары (u, v) , где $u, v \in V$. Чтобы упростить обозначения, мы будем использовать запись $uv \in E$ вместо $(u, v) \in E$. Поскольку пары являются упорядоченными, запись $uv \in E$ не эквивалентна записи $vu \in E$. Большинство определений для неориентированных графов после небольших видоизменений переносятся на случай ориентированных графов.

Говорят, что дуга uv *исходит* из вершины u и *входит* в вершину v . Если $uv \in E$, то вершина u называется *входным соседом* вершины v , а вершина v — *выходным соседом* вершины u . *Полустепенью захода* вершины называется число дуг, входящих в эту вершину, а *полустепенью исхода* — число дуг, исходящих из вершины. Степень вершины равна сумме полустепени исхода и полустепени захода²⁾.

Путем длины k из вершины v_0 в вершину v_k называется такая последовательность вершин $P = \langle v_0, \dots, v_k \rangle$, что для любого $i < k$ выполняется равенство $v_i v_{i+1} \in E$. Вершина v_0 называется начальной вершиной, а вершина v_k — конечной вершиной пути. *Циклом* называется путь, в котором начальная вершина совпадает с конечной. Так же как и для неориентированных графов, путь называется *простым*, если все его вершины начиная с v_0 и оканчивая v_k являются попарно различными, а цикл называется *простым*, если все его вершины, начиная с v_1 и оканчивая v_k , являются попарно различными.

Расстоянием от вершины u до вершины v (обозначается $d(u, v)$) называется длина кратчайшего пути из вершины u в вершину v . *Диаметром* ориентированного графа G называется наибольшее расстояние от одной из вершин графа до другой. Ориентированный граф называется *сильно связным*, если для любой пары вершин существует путь из одной вершины в другую. Ориентированный граф называется *ациклическим*, если в нем нет циклов длины два или более.

Подграфы, остовные подграфы и индуцированные подграфы определяются для ориентированных графов точно так же, как и для неориентированных графов.

Б.1.3. Взвешенные графы

(Ориентированный или неориентированный) граф называется *взвешенным*, если для каждой пары вершин u, v , соединенных ребром (дугой) $uv \in E$, определена числовая величина ω_{uv} . Весовая оценка называется *симметричной*, если для каждой пары выполняется равенство $\omega_{uv} = \omega_{vu}$. Для взвешенного графа весом пути называется сумма весовых величин, приписанных всем ребрам (дугам) пути. Если в графе ребрам или дугам не приписываются весовые величины, то он называется *невзвешенным*.

²⁾В том случае, если в графе нет петель, т. е. дуг вида (v, v) . — Прим. перев.

Б.2. Наиболее распространенные виды графов

В этом параграфе мы определим и обсудим некоторые виды графов, с которыми приходится сталкиваться наиболее часто при изучении распределенных алгоритмов, а именно кольца, деревья, леса, решетки, торы, клики и гиперкубы. Как можно видеть из результатов, представленных в этой книге, вычисления распределенных систем могут быть выполнены более эффективно на сетях с регулярной топологией, нежели на (связных) сетях с произвольной топологией.

Далее мы будем полагать, что $G = (V, E)$ — это граф диаметра D и в этом графе имеется N вершин.

Б.2.1. Кольца

В кольцевой топологии все узлы сети располагаются по кругу; такое устройство часто имеет управляющая топология в распределенных вычислениях. В некоторых сетях (например, в кольцах с маркерами [182, § 4.3]) узлы действительно образуют кольцо, но в других случаях кольца используются просто в качестве встроенной управляющей структуры. Чтобы соблюсти очередность выполнения своих функций каждым процессом по принципу кругового обслуживания, процессы передают друг другу сообщение: процесс v_0 передает его процессу v_1 , процесс v_1 — процессу v_2 , и т. д., до тех пор пока его не получит процесс v_{N-1} , который вновь передаст его процессу v_0 .

Определение Б.1. *Кольцом* называется неориентированный связный регулярный граф степени два.

Кольца можно описывать разными способами.

Теорема Б.2. *Для неориентированного графа G следующие утверждения равносильны.*

1. *Граф G является кольцом.*
2. *Существует такое взаимно однозначное соответствие между множеством вершин V и множеством натуральных чисел $\{0, \dots, N-1\}$, что соседями вершины, соответствующей числу i , являются вершины с номерами $i-1$ и $i+1$ (по модулю N).*

Сеть с произвольной топологией может воспользоваться преимуществами кольцевой топологии; для этого достаточно отыскать в графе $G = (V, E)$ остовное кольцо, т. е. выбрать такое множество ребер $E' \subseteq E$, чтобы граф (V, E') представлял собой кольцо. К сожалению, не всякий граф содержит остовное кольцо (гамильтонов цикл), а сама задача проверки того, содержится ли остовное кольцо в заданном графе, является NP-полной (см. [98]). Можно, однако, определить *виртуальное кольцо*; в этом случае соседние вершины кольца не обязаны быть соседними вершинами в исходном графе. Беккер и Ван Льювен в работе [20] описали распределенный алгоритм для построения виртуального кольца, в котором расстояние между последовательно расположенными вершинами не превышает трех.

Б.2.2. Деревья, леса и звезды

Дерево — это граф, в котором имеется минимально необходимое число ребер для соединения его вершин (см. теорему Б.4), и поэтому вычисления на древовидных сетях могут иметь очень малую коммуникационную сложность. Для децентрализованных вычислений дополнительным фактором, благодаря которому вычисления на деревьях имеют малую сложность, является то обстоятельство, что для древесных сетей существует эффективный децентрализованный волновой алгоритм (см. алгоритм 6.2).

Определение Б.3. *Деревом* называется неориентированный связный ациклический граф.

В приведенной ниже теореме перечислены некоторые характеристические свойства деревьев (см. [55]).

Теорема Б.4. *Для всякого неориентированного G графа следующие утверждения равносильны.*

1. Граф G — дерево.
2. Между любыми двумя вершинами пролегает единственный простой путь.
3. Граф G связный, но при удалении любого ребра становится несвязным.
4. Граф G связный, и $|E| = N - 1$.
5. Граф G ациклический, и $|E| = N - 1$.
6. Граф G ациклический, но при добавлении любого нового ребра в нем появляется цикл.

Дерево $T = (V, E)$ называется *корневым*, в нем особо выделена одна вершина r , которая называется *корнем*. Если вершина u лежит на единственном простом пути, пролегающем между вершинами v и r , то вершина u называется *предком* вершины v , а вершина v называется *потомком* вершины u . Если к тому же u и v оказываются соседними вершинами, то вершина v называется *сыновней* (или *дочерней вершиной* для u , а вершина u называется *родительской* (или *отцовской вершиной*) для v . Подграф, образованный вершиной u и всеми ее потомками, является корневым деревом (его корнем служит вершина u); его называют *поддеревом, растущим из вершины u* . *Высотой* дерева называется максимальная длина простого пути, ведущего из корня дерева в одну из его вершин.

В каждом связном графе $G = (V, E)$ существует остовное дерево. Это означает, что можно выбрать такое множество ребер $E' \subseteq E$, что граф (V, E') будет деревом. Следует заметить, что если сам граф G не является деревом, то остовное дерево для него может быть выбрано многими разными способами. Ребра, входящие в множество E' , называются *древесными ребрами*, оставшиеся ребра из множества $E \setminus E'$ называются *стягивающие ребрами*.

Для распределенных вычислений интерес представляют несколько типов остовных деревьев.

1. *Остовные деревья с малым диаметром.* Если нужно выбрать такое остовное дерево, чтобы минимизировать общее время вычисления на соответствующей ему сети, то желательно, чтобы диаметр дерева был как можно меньше. Оптимальное входное дерево, построенное в доказательстве теоремы 4.2, имеет диаметр, который не более чем в два раза превосходит диаметр всей сети. Такое дерево можно построить, воспользовавшись алгоритмами из гл. 4 и § 12.6.4.

2. *Остовное дерево минимального веса.* Остовное дерево графа G всегда имеет $N - 1$ ребер (см. теорему Б.4 (4)), но для взвешенного графа вес дерева (сумма весов, приписанных ребрам дерева) обычно бывает разным для разных остовных деревьев. Если мы хотим понизить общие расходы на коммуникацию при проведении вычисления на остовном дереве, то нам нужно выбирать такое поддерево, чтобы его вес был минимальным. Если все веса, приписанные ребрам графа, попарно различны, то существует только одно остовное дерево минимального веса (см. утверждение 7.18). Распределенный алгоритм вычисления этого единственного дерева был предложен Галладжером и др. в работе [96]; он рассматривается в § 7.3.2.

3. *Остовные деревья ограниченной степени.* Если мы хотим понизить издержки, связанные с организацией связи, которые приходятся на один узел, то желательно выбирать такое остовное дерево, в котором каждая вершина имеет как можно меньшую степень. Распределенное вычисление таких деревьев описано Корачем и др. в работе [120].

4. *Деревья поиска в глубину.* Остовное дерево называется остовным деревом поиска в глубину, если каждое стягивающее ребро соединяет вершину и ее потомка. Деревья поиска в глубину применяются во многих алгоритмах, работающих на графах, в частности, в алгоритмах проверки планарности или двухсвязности. Они используются также в компактной маршрутизации для построения интервальных схем разметки (см § 4.4.2). Распределенные алгоритмы построения деревьев поиска в глубину приведены в § 6.6.4.

Обобщением деревьев являются леса, а звезды являются специальной разновидностью деревьев. Лесом называется граф, состоящий из некоторого числа изолированных деревьев.

Определение Б.5. *Лес* — это неориентированный ациклический граф.

Теорема Б.6. *Для всякого неориентированного графа G следующие утверждения равносильны.*

1. Граф G — лес.
2. Между любыми двумя вершинами графа G пролегает не более одного простого пути.
3. Каждая компонента связности графа G является деревом.
4. При удалении любого ребра число компонент связности графа G увеличивается на 1.
5. Число компонент связности графа G равно $|V| - |E|$.

Лес называется корневым, если в каждом дереве этого леса выделена корневая вершина. В каждом графе можно выделить остовный лес, который состоит из остовных деревьев всех компонент связности графа.

Звездой называется всякий граф, в котором выделена одна специальная вершина, которая называется центром, а все остальные вершины соединены только с центром.

Определение Б.7. Звезда — это корневое дерево высоты 1. Корень звезды называется ее центром.

Звезды нечасто встречаются в топологии физических соединений распределенной системы, но именно в виде звезд представляется виртуальная топология коммуникаций, которая возникает в распределенных вычислениях под управлением центрального процесса. В таких вычислениях все процессы обмениваются сообщениями только с центром, и поэтому из всей коммуникационной сети задействованным оказывается только остовный подграф, который является звездой (см., например, алгоритм 6.5).

Теорема Б.8. Для всякого неориентированного графа G следующие утверждения равносильны.

1. Граф G — звезда.
2. G — связный граф, в котором есть самое большое одна вершина, степень которой превосходит 1.

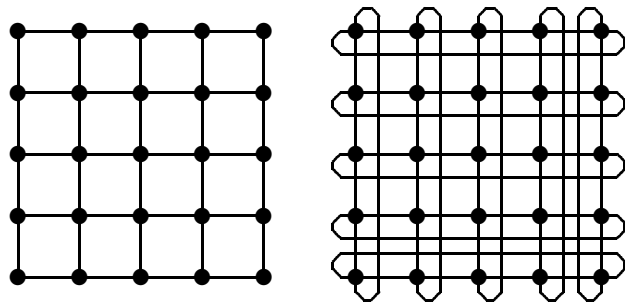


Рис. Б.2. Решетка и тор

Б.2.3. Клики

В кликах, которые иногда называют *полными графами*, каждая пара вершин непосредственно соединена ребром.

Определение Б.9. Клика — это граф диаметра 1.

Так же как и звезды, клики редко встречаются в топологии физических соединений распределенной системы. Это обусловлено тем, что для реализации большого числа каналов связи, соответствующих ребрам такого графа, потребовалось

бы много аппаратуры. В большинстве сетей отнюдь не каждая пара узлов непосредственно соединена друг с другом, и для коммуникации между несмежными узлами требуется взаимодействие всех узлов на пути между этими несмежными узлами. Если же это взаимодействие осуществляется на «нижних» уровнях системы и поэтому невидимо, то с точки зрения «верхних» уровней сеть можно рассматривать как клику.

Теорема Б.10. Для всякого неориентированного графа G следующие утверждения равносильны.

1. Граф G — клика.
2. $E = \{uv : u, v \in V \text{ и } u \neq v\}$.
3. $|E| = \frac{1}{2}N(N-1)$.
4. Каждая вершина графа G имеет степень $N-1$.

Б.2.4. Решетки и торы

В решетке размера $n \times n$ имеется $N = n^2$ вершин, размещенных в n рядах по n вершин в каждом ряду. Каждая вершина соединена с вершинами, расположенными над ней, справа от нее, слева от нее и под ней (см. рис. Б.2). Тор размера $n \times n$ устроен сходным образом, но, кроме того, в каждом ряду самая правая и самая левая вершины соединены друг с другом; также в каждом ряду соединены друг с другом самая верхняя и самая нижняя вершины (см. рис. Б.2). Решетки и торы — наиболее распространенные топологии при проектировании многопроцессорных компьютеров. Ввиду того что степень каждой вершины в этих графах не превосходит 4, эти топологии можно физически реализовать, используя микропроцессоры Transputer. Указанные топологии очень удобны для выполнения вычислений над матрицами.

Чтобы привести формальное определение этих топологий, напомним, что символом $\mathbb{Z}_n$ мы обозначили множество натуральных чисел с операциями по модулю n .

Определение Б.11. Решеткой размера $n \times n$ называется граф, имеющий $N = n^2$ вершин, которые взаимно однозначно сопоставлены элементам из множества $\{(i, j) : 0 \leq i, j < n\}$ так, что вершины, помеченные парами (i, j) и (i', j') , являются смежными в том и только том случае, когда $(i = i') \wedge (j = j' + 1)$, или $(i = i') \wedge (j = j' - 1)$, или $(i = i' + 1) \wedge (j = j')$, или $(i = i' - 1) \wedge (j = j')$.

Тором размера $n \times n$ называется граф, имеющий $N = n^2$ вершин, которые взаимно однозначно сопоставлены элементам из множества $\{(i, j) : 0 \leq i, j < n\}$ так, что вершины, помеченные парами (i, j) и (i', j') , являются смежными в том и только том случае, когда $(i = i') \wedge (j = j' + 1)$, или $(i = i') \wedge (j = j' - 1)$, или $(i = i' + 1) \wedge (j = j')$, $(i = i' - 1) \wedge (j = j')$, где все арифметические операции выполняются по модулю n .

Решетка размера $n \times n$ имеет $2n(n-1)$ ребер, содержит вершины степени два, три и четыре; диаметр этой решетки равен $2(n-1)$. Тор размера $n \times n$ имеет $2n^2$ ребер, является регулярным графом степени 4, и его диаметр равен $2\lfloor \frac{n}{2} \rfloor$.

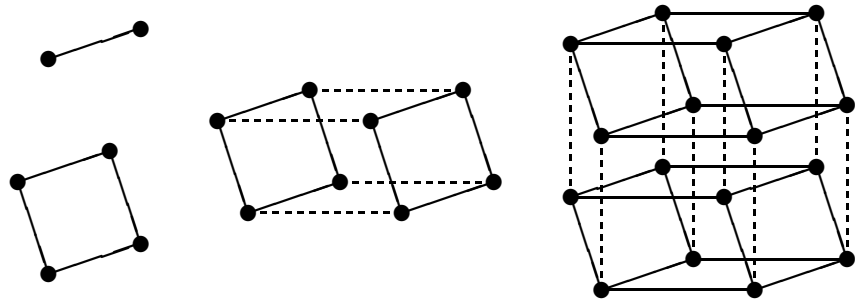


Рис. Б.3. Гиперкубы размерности 1, 2, 3 и 4

Б.2.5. Гиперкубы

Наряду с решетками и торами при проектировании многопроцессорных компьютеров используются гиперкубы. В них достаточно небольшой диаметр сочетается со сравнительно небольшой степенью: и диаметр, и степень гиперкуба равны $\log N$, где N — число вершин. Топологические свойства этого класса графов были изучены в работе Саада и Шульца [167].

Определение Б.12. Гиперкубом размерности n называется граф, имеющий $N = 2^n$ вершин, которые взаимно однозначно соответствуют двоичным наборам $\{(b_0, \dots, b_{n-1}) : b_i = 0, 1\}$ таким образом, что

$E = \{uv : \text{наборы, соответствующие вершинам } u \text{ и } v, \text{ отличаются ровно в одном разряде}\}.$

Гиперкуб является регулярным графом степени n , и его диаметр равен n . Существуют разные способы определения гиперкуба; некоторые интересные теоретико-графовые свойства гиперкубов обсуждаются в работе [167]. Прежде всего, разметку вершин гиперкуба двоичными наборами, о которой говорится в определении Б.12, можно задавать по-разному. Если граф G является n -мерным гиперкубом, то для него существует в точности $2^n n!$ таких разметок.

Чтобы построить $(n+1)$ -мерный гиперкуб, можно взять два n -мерных гиперкуба и соединить их соответствующие³⁾ вершины. На рис. Б.3 ребра, соединяющие соответствующие вершины, выделены пунктирными линиями.

Если разметка вершин гиперкуба уже задана, то нетрудно убедиться в справедливости следующего утверждения. Если u и v — смежные вершины, то соседи вершины u и соседи вершины v соединены друг с другом взаимно однозначно. Под этим подразумевается следующее: каждый сосед вершины v является смежным в точности с одним соседом вершины u , и наоборот. Гораздо труднее показать, что взаимно однозначного соответствия между соседними вершинами (наряду с некоторыми дополнительными достаточно простыми свойствами) достаточно для того, чтобы граф был гиперкубом.

³⁾т. е. помеченные одинаковыми наборами. — Прим. перев.

Теорема Б.13. Для всякого неориентированного графа G следующие утверждения равносильны.

1. Граф G — гиперкуб размерности n .
2. Справедливо равенство $N = 2^n$, и G — регулярный связный граф степени n , у которого для любой пары соседних вершин u, v соседи вершины v взаимно однозначно соединены с соседями вершины u .

Итоги. В таблице Б.4 собраны воедино основные характеристики тех классов графов, которые были рассмотрены в этом параграфе.

Графы	степени	$ E $	D	Комментарии
Кольца	2	N	$\left\lfloor \frac{N}{2} \right\rfloor$	$D = 1$ лишь при $N = 2$
Деревья	переменная	$N - 1$	$< N$	
Леса	переменная	$< N$		
Звезды	1 или $N - 1$	$N - 1$	2 или 1	
Клики	$N - 1$	$\frac{1}{2}N(N - 1)$	1	$N = n^2$
Решетки $(n \times n)$	2, 3, 4	$2n(n - 1)$	$2(n - 1)$	
Торы $(n \times n)$	4	$2n^2$	$2 \left\lfloor \frac{n}{2} \right\rfloor$	
Гиперкубы (n мерный)	n	$2^{n-1}n$	n	

Таблица Б.4. Классы графов

Б.3. Восприятие направления

Санторо в работе [168] удалось обнаружить, что коммуникационная сложность распределенных вычислений зависит не только от топологии как таковой, но и от того, насколько процессы системы осведомлены о ней.

1. *Топологическая осведомленность.* Иногда, чтобы воспользоваться теми преимуществами, которые предоставляют топологии специального вида, необходимо, чтобы процессы «знали» о том, что они связаны друг с другом сетью определенного вида.

Например, если сеть образует клику, то задачу избрания лидера (гл. 7) можно решить, совершив всего лишь $O(N \log N)$ обменов сообщениями. Однако, если для клики применить алгоритм избрания лидера, который является оптимальным для сетей произвольного вида, то потребуется совершить $\Omega(N^2)$ обменов сообщениями. Таким образом, чтобы воспользоваться преимуществами полного соединения, процессы должны быть осведомлены о том, что они образуют клику.

2. *Восприятие направления.* Распространение информации по маршрутам в сети можно сделать более эффективным, если ребра, инцидентные каждой вершине, пометить так, чтобы в этой пометке каким-либо образом было то «направление» в сети, по которому направлено это ребро. Подобная информация позволяет нам отыскать короткий (обычно оптимальный) путь к вершине, о ко-

торой было известно только то, что она соединена с нашей исходной вершиной какими-то длинными путями.

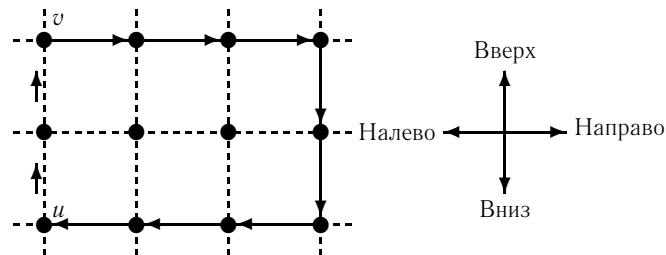


Рис. Б.5. Как воспользоваться восприятием направления

В качестве иллюстрации тех преимуществ, которые открываются с восприятием направления, рассмотрим рис Б.5, на котором изображен фрагмент решетчатой сети. В узле v был сформирован запрос на предоставление некоторого ресурса, который был передан последовательно по цепочке узлов (не располагающих требуемым ресурсом), и, наконец, достиг узла u , обладающего этим ресурсом. Если узел u будет знать о том, что сообщение с этим запросом было передано три раза направо, два раза вниз и три раза налево, то u отправляет сообщение вверх. Вместо того чтобы следовать в обратном направлении по пути (длины 8), пройденному запросом, отклик передается по пути длины 2. Ясно, что кратчайший обратный путь может быть найден только тогда, когда узлы «знают», в каком направлении ведет каждое инцидентное ребро.

Конечно, смысл, который придается слову «направление», зависит от топологии сети. Однако в любом случае предполагается, что для каждого узла все ребра, инцидентные этому узлу, помечены так, чтобы в этих пометках были представлены только «разумные направления» для рассматриваемого класса графов. Для заданной разметки $\mathcal{S}$ будем использовать запись $\mathcal{S}_v(u)$ для обозначения той пометки, которая приписана в узле v ребру uv ; то же самое ребро в узле u будет иметь пометку $\mathcal{S}_u(v)$. Множество допустимых пометок должно быть одним и тем же для каждого узла; мощность этого множества $\mathcal{L}$ равна степени сети, поскольку в каждом узле различные ребра имеют разные пометки. Чтобы возникло восприятие направления, необходимо проверить дополнительные условия глобальной непротиворечивости. Далее мы рассмотрим четыре класса графов и расскажем о том, каким будет восприятие направления для каждого из этих классов.

Кольца. Так как кольцо — это регулярный граф степени 2, в кольце есть только два направления; мы будем называть эти направления «+» (положительное, по часовой стрелке) и «-» (отрицательное, против часовой стрелки). Естественно, если вершина u следует по часовой стрелке вслед за вершиной v , то вершина v следует против часовой стрелки за вершиной u . Если это отношение

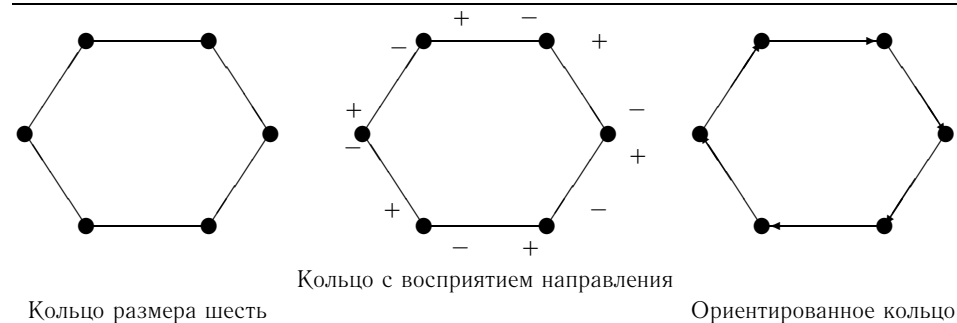


Рис. Б.6. Восприятие направления в кольцах

выполняется для каждой пары смежных вершин, то ориентация будет глобально непротиворечивой (см. рис Б.6).

Определение Б.14. Разметкой кольца называется отображение, которое для каждой вершины кольца приписывает инцидентным ей ребрам разные пометки из множества $\{-, +\}$.

Разметка кольца $\mathcal{S}$ задает восприятие направления, если для любой пары смежных вершин u и v выполняется соотношение

$$\mathcal{S}_u(v) = - \iff \mathcal{S}_v(u) = +.$$

Восприятие направления в кольце не уникально: есть две разметки ребер, которые удовлетворяют указанному выше условию. Всегда предполагается, что у колец есть восприятие направления. Ясно, что если некоторый узел отправит сообщение по часовой стрелке и каждый узел, получивший сообщение, снова передаст его по часовой стрелке, то все узлы кольца получат это сообщение, перед тем как оно вернется в исходный узел. Обычно каналы связи используются для передачи сообщений только в одном направлении, и чаще всего предполагается, что кольца ориентированы так, как это показано на рис. Б.6.

Клики. Клика, состоящая из N вершин, имеет степень $N - 1$; ее направления занумерованы числами от 1 до $N - 1$, т. е. $\mathcal{L} = \{1, \dots, N - 1\}$. Разметка задает восприятие направления, если в клике можно выделить такой ориентированный гамильтонов цикл, что для каждой вершины u ребро uv помечено числом, которое равно расстоянию от вершины u до вершины v в указанном ориентированном цикле.

Определение Б.15. Разметкой клики, состоящей из N вершин, называется отображение, которое для каждой вершины клики приписывает инцидентным ей ребрам разные пометки из множества $\{1, \dots, N - 1\}$.

Разметка клики $\mathcal{S}$ задает восприятие направления, если вершины клики можно так занумеровать числами от 0 до $N - 1$, что для каждой пары чисел i и j выполняется соотношение $\mathcal{S}_i(j) = (j - i) \bmod N$.

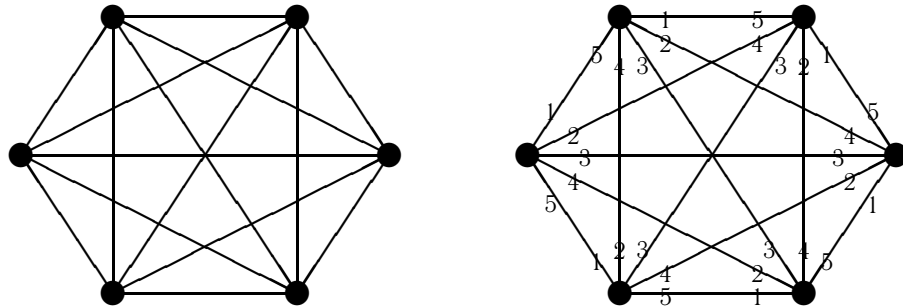


Рис. Б.7. Восприятие направления в клике

Одна из клик с восприятием направления изображена на рис. Б.7. Восприятие направления в клике не уникально: существует в точности $(N - 1)!$ различных разметок, задающих восприятие направления в клике. Лоуи и др. в работе [134] показали, что при наличии восприятия направления в клике задача избрания лидера может быть решена с использованием $O(N)$ сообщений.

Торы. Четыре направления в торе — вверх, вниз, направо и налево — мы условно обозначим буквами U, D, R, L .

Определение Б.16. Разметкой тора называется отображение, которое для каждой вершины тора приписывает инцидентным ей ребрам разные пометки из множества $\{U, D, R, L\}$.

Разметка тора $\mathcal{S}$ задает восприятие направления, если вершины тора можно так пометить парами натуральных чисел $\{(i, j) : i, j \in \mathbb{Z}_n\}$, что будут выполняться соотношения

- 1) $\mathcal{S}_{(i,j)}((i', j')) = U \implies (i' = i) \wedge (j' = j + 1)$;
- 2) $\mathcal{S}_{(i,j)}((i', j')) = D \implies (i' = i) \wedge (j' = j - 1)$;
- 3) $\mathcal{S}_{(i,j)}((i', j')) = L \implies (i' = i - 1) \wedge (j' = j)$;
- 4) $\mathcal{S}_{(i,j)}((i', j')) = R \implies (i' = i + 1) \wedge (j' = j)$.

Восприятие направления в торе не уникально: если $N > 4$, то существует в точности восемь различных разметок, задающих восприятие направления в торе. Один из торов с восприятием направления изображена на рис. Б.8.

Гиперкубы. В n -мерном гиперкубе все n измерений занумерованы целыми числами от 0 до $n - 1$. Считается, что в гиперкубе есть восприятие направления, если все параллельные ребра помечены одним и тем же числом.

Определение Б.17. Разметкой n -мерного гиперкуба называется отображение, которое для каждой вершины тора приписывает инцидентным ей ребрам разные пометки из множества $\{0, \dots, n - 1\}$.

Разметка гиперкуба $\mathcal{S}$ задает восприятие направления, если вершины гиперкуба можно так пометить двоичными наборами из множества $\{(b_0, \dots, b_{n-1}) : b_i =$

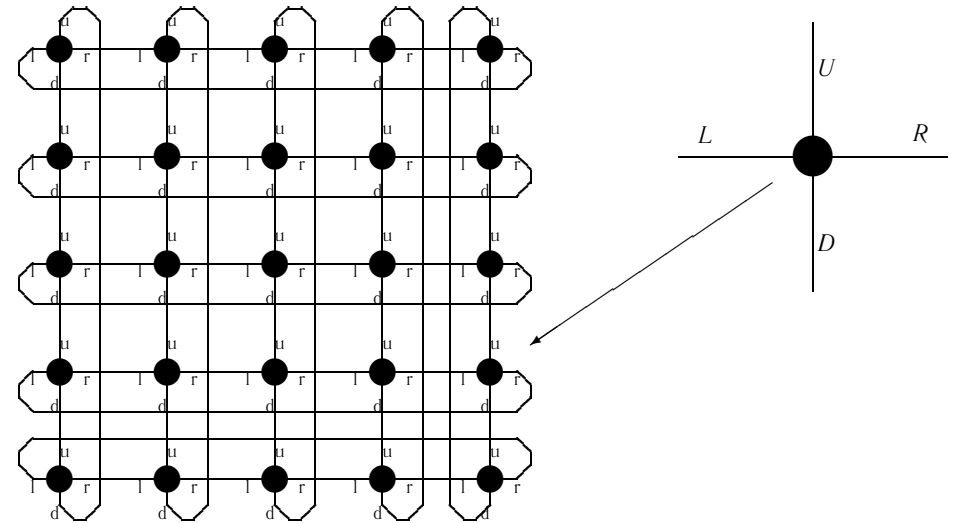


Рис. Б.8. Восприятие направления в торе

$= 0, 1\}$, что всякий раз, когда ребру приписана пометка $\mathcal{S}_{(b_0, \dots, b_{n-1})}((c_0, \dots, c_{n-1})) = i$, двоичные наборы $(b_0, \dots, b_{n-1})$ и $(c_0, \dots, c_{n-1})$, которые соответствуют вершинам, связанным этим ребром, отличаются в точности в одном разряде, а именно, в разряде i .

Восприятие направления в гиперкубе не уникально: в n -мерном гиперкубе имеется в точности $n!$ различных разметок, задающих восприятие направления. Из приведенного определения следует, что для смежных вершин u и v справедливо равенство $\mathcal{S}_u(v) = \mathcal{S}_v(u)$. Поэтому на рис. Б.9 каждое ребро помечено только одним числом.

В таблице Б.10 собраны характеристики восприятия направления для четырех рассмотренных классов графов. Общее определение восприятия направления и формальное описание его свойств можно найти в работе Флоччини и др. [86]. Как показано в гл. 11, для всех графов, рассмотренных в этом параграфе, а также для ряда других графов восприятие направления наиболее удобно описывать при помощи групп. В этой главе, а также в работе [190] приведены распределенные алгоритмы для вычисления восприятия направления в гиперкубе.

Б.4. Упражнения к главе Б

Б.4.1

Упражнение Б.1. Докажите, что для неориентированных графов справедливы следующие утверждения:

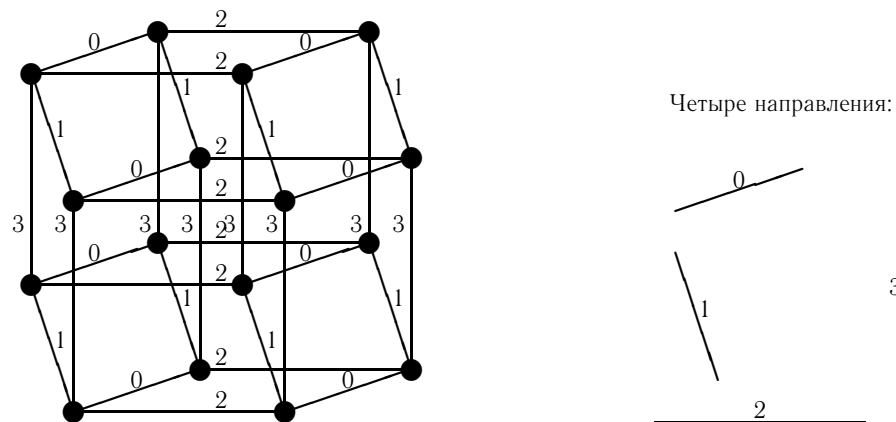


Рис. Б.9. Восприятие направления в гиперкубе

Связный граф с n вершинами имеет не менее $n - 1$ ребер.
 Ациклический граф с n вершинами имеет не более $n - 1$ ребер.

Неориентированный граф называется *двудольным*, если его вершины можно так раскрасить в два цвета (красный и черный), что никакие две вершины одного цвета не являются смежными.

Упражнение Б.2. Покажите, что неориентированный граф является двудольным тогда и только тогда, когда в нем нет циклов нечетной длины.

Б.4.2

Упражнение Б.3. Докажите, что 1) кольца с четным числом вершин, 2) деревья и 3) гиперкубы являются двудольными графами.

Упражнение Б.4. Докажите теоремы Б.4 и Б.6. (Подсказка: воспользуйтесь упражнением Б.1.)

Курсовая работа Б.5. Останется ли верным утверждение теоремы Б.13, если не выполняется хотя бы одно из условий пункта (2)?

Класс графов	Степень	Мощность множества $\mathcal{L}$	Число восприятий направления	Определение совместности приведено в
Кольца	2	$\{-, +\}$	2	Определение Б.14
Клики	$N - 1$	$\{1, \dots, N - 1\}$	$(N - 1)!$	Определение Б.15
Торы	4	$\{U, D, L, R\}$	8 (если $n > 4$)	Определение Б.16
Гиперкубы	n	$\{0, \dots, n - 1\}$	$n!$	Определение Б.17

Таблица Б.10. Восприятие направления в графах

Б.4.3

Упражнение Б.6. Докажите, что существует в точности $(N - 1)!$ способов задания восприятия направления в клике.

Докажите, что существует в точности $n!$ способов задания восприятия направления в n -мерном гиперкубе.

Упражнение Б.7. Пусть задана разметка клики $\mathcal{S}$. Для всякого пути $P = \langle v_0, \dots, v_k \rangle$ будем полагать $Sum(P) = \mathcal{S}_{v_0}(v_1) + \dots + \mathcal{S}_{v_{k-1}}(v_k)$. Докажите, что следующие утверждения равносильны.

1. Разметка $\mathcal{S}$ задает восприятие направления.
2. Всякий путь P является циклом тогда и только тогда, когда $Sum(P) \equiv 0 \pmod{N}$.

ЛИТЕРАТУРА

- [1] Afek Y., Landau G. M., Schieber B., Yung M. The power of multimedia: combining point-to-point and multicast networks. *Inf. Comput.*, **34**, N 1, 1990, p. 97-118.
- [2] Afek Y., Kutten S., Yung M. Memory-efficient self stabilizing protocols for general networks. In Proc. *4th Int. Workshop on Distributed Algorithms* (Bari, 1990), J. van Leeuwen, N. Santoro (eds.), Lecture Notes in Computer Science, vol. 486, p. 15-28.
- [3] Afek Y., Gafni E. Time and message bounds for election in synchronous and asynchronous complete networks. *SIAM J. Comput.*, **20**, N 2, 1991, p. 376-394.
- [4] Aguilera M. K., Chen W., Toueg S. Heartbeat: a timeout-free failure detector for quiescent reliable communication. In Proc. *11th Int. Workshop on Distributed Algorithms* (1997), Mavronicolas M., Tsigas P. (eds.), Lecture Notes in Computer Science, vol. 1320, p. 126-140.
- [5] Ahuja M. Flush primitives for asynchronous distributed systems. *Inf. Proc. Lett.*, **34**, 1990, p. 5-12.
- [6] Alpern B., Schneider F. B. Defining liveness. *Inf. Proc. Lett.*, **21**, 1985, p. 181-185.
- [7] Anagnostou E., Hadzilacos V. Tolerating transient and permanent failure. In Proc. *Int. Workshop on Distributed Algorithms* (Lausanne, 1992), A. Schiper (eds.), Lecture Notes in Computer Science, vol. 725, p. 144-188.
- [8] Angluin D. Local and global properties in networks of processes. In Proc. *Symp. on Theory of Computing*, 1980, p. 82-93.
- [9] Attiya H. Constructing efficient election algorithms from efficient traversal algorithms. In Proc. *2nd Int. Workshop on Distributed Algorithms* (Amsterdam, 1987), J. van Leeuwen (ed.), Lecture Notes in Computer Science, vol. 312, p. 377-384.
- [10] Attiya H., van Leeuwen J., Santoro N., Zaks S. Efficient elections in chordal ring networks. *Algorithmica*, **4**, 1989, p. 437-466.
- [11] Attiya H., Bar-Noy A., Dolev D., Peleg D., Reischuk R. Renaming in an asynchronous environment. *J. ACM*, **37**, 1990, p. 524-548.
- [12] Attiya H., Welch J. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*. McGraw-Hill, 1998.
- [13] Awerbuch B. Optimal distributed algorithm for minimum weight spanning tree, counting, leader election and related problems. *Inf. Proc. Lett.*, **20**, 1985, p. 147-150.
- [14] Awerbuch B. Complexity of network synchronization. *J. ACM*, **32**, 1985, p. 804-823.
- [15] Awerbuch B., Gallager R. G. Distributed DFS algorithms. In Proc. *Foundations of Computation Theory*, 1985, p. 250-256.
- [16] Awerbuch B. A new distributed depth first search algorithm. In Proc. *Symp. on Theory of Computing*, 1987, p. 230-240.
- [17] Awerbuch B., Bar-Noy A., Linial N., Peleg D. Improved routing strategies with succinct tables. *J. Algorithms*, **11**, 1990, p. 307-341.
- [18] Bakker E. M., Leeuwen J. van, Tan R. B. Linear interval routing. *Algorithmic Review*, **2**, N 2, 1991, 45-61.
- [19] Bakker E. M., Leeuwen J. van, Tan R. B. Prefix routing schemes in dynamic networks. *Computer Networks and ISDN Systems*, **26**, 1993, 403-421.
- [20] Bakker E. M., Leeuwen J. van. Uniform d -emulations of rings, with an application to distributed virtual ring construction. *Networks*, **23**, 1993, p. 237-248.
- [21] Bal H. *Programming Distributed Systems*. Prentice Hall, 1996.
- [22] Bar-Ilan J., Zernik D. Random leaders and random spanning trees. In Proc. *3rd Int. Workshop on Distributed Algorithms* (Nice, 1989), J.-C. Bermond and M. Raynal (eds.), Lecture Notes in Computer Science, vol. 392, p. 1-12.
- [23] Baratz A., Gopal L., Segal A. Fault tolerant queries in computer networks. In Proc. *2nd Int. Workshop on Distributed Algorithms* (Amsterdam, 1987), J. van Leeuwen (ed.), Lecture Notes in Computer Science, vol. 312, p. 30-40.
- [24] Barbosa V. C. *An Introduction to Distributed Algorithms*. MIT Press, 1996, 365 p.
- [25] Beame P. W., Bodlaender H. L. Distributed computing on transitive networks: the tours. In Proc. *Symp. on Theoretical Aspects of Computer Science* (1987), B. Monien, R. Cori (eds.), Lecture Notes in Computer Science, vol. 349, p. 294-303.
- [26] Belsnes D. Single-message communication. *IEEE Trans. Comp.*, **COM-24**, N 2, 1976, p. 190-194.

- [27] Ben-Ari M. Algorithms for on-the-fly garbage collection. *ACM Trans. Program. Lang. Syst.*, **6**, N 3, 1984, p. 333-344.
- [28] Berman P., Garay J. A., Perry K. J. Optimal early stopping in distributed consensus. In Proc. *6nd Int. Workshop on Distributed Algorithms* (Haifa, 1992), S. Zaks, A. Segall (eds.), Lecture Notes in Computer Science, vol. 647, p. 221-237.
- [29] Biran O., Moran S., Zaks S. A combinatorial characterization of the distributed tasks which are solvable in the presence of one faulty processor. *J. Algorithms*, **11**, N 1990, p. 420-440.
- [30] Bodlaender H. L. Distributed computing: Structure and complexity. Ph.D. thesis. Dept. Computer Science, Utrecht University, The Netherlands, 1986, 297 p.
- [31] Bodlaender H. L. A better lower bound for distributed leader finding in bidirectional asynchronous rings of processors. *Inf. Proc. Lett.*, **27**, 1988, p. 287-290.
- [32] Bodlaender H. L. New lower bound techniques for distributed leader finding and other problems on rings of processors. *Theor. Comput. Sci.*, **81**, 1991, p. 237-256.
- [33] Bodlaender H. L. Some lower bound results for decentralized extrema-finding in rings of processors. *J. Comput. Syst. Sci.*, **42**, N 1, 1991, p. 97-118.
- [34] Bourne S. R. *The Unix System*. Addison-Wesley, 1983, 351 p.
- [35] Bracha G., Toueg S. Asynchronous consensus and broadcast protocols. *J. ACM*, **32**, 1985, p. 824-840.
- [36] Brassrd G., Bratley P. *Algorithms: Theory and Practice*. Prentice-Hall, 1988.
- [37] Brzezinski J., Helary J.-M., Raynal M. Deadlock models and general algorithm for distributed deadlock detection. Research Report, IRISA, Rennes, 1992.
- [38] Burns J. E., Pachl J. Uniform self-stabilizing rings. In Proc. *Aegean Workshop on Computing* (1988), J. H. Reif (ed.), Lecture Notes in Computer Science, vol. 319, p. 391-400.
- [39] Burns J. E., Gouda M. G., Miller M. E. Stabilization and pseudo-stabilization. *Distributed Comput.*, **7**, N 1, 1993, p. 35-42.
- [40] Chang E. J.-H., Roberts R. An improved algorithm for decentralized extrema finding in circular arrangements of processes. *Commun. ACM*, **22**, 1979, p. 281-283.
- [41] Chang E. J.-H. Echo algorithms: Depth parallel operations on general graphs. *IEEE Trans. Soft. Eng.*, **SE-8**, 1983, p. 504-512.

- [42] Chandra T. D., Toueg S. Unreliable failure detectors for reliable distributed systems. *J. ACM*, **43**, N 2, 1996, p. 225-267.
- [43] Chandrasekarnan K. M., Venkatesan S. A message-optimal algorithm for distributed termination detection. *J. Parallel and Distributed Computing*, **8**, N 3, 1990, p. 245-252.
- [44] Chandy K. M., Misra J. Distributed computation on graphs: shortest path algorithms. *Commun. ACM*, **25**, N 11, 1982, p. 833-838.
- [45] Chandy K. M., Lamport L. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.*, **3**, N 1, p. 63-75.
- [46] Chandy K. M., Misra J. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988, 516 p.
- [47] Charron-Bost B. Mesures de la concurrence et du parallelisme des calculs repartis. Ph.D. thesis, Universite Paris VII, 1989, 129 p.
- [48] Charron-Bost B., Mattern F., Tel G. Synchronous, asynchronous, and casually ordered communication. *Distributed Comput.*, **9**, 1996, p. 173-191.
- [49] Charron-Bost B., Mattern F., Tel G. Calculus approches de la borne inferieure de valeurs reparties. *Inf. Theor. appl.*, **31**, N 4, 1997, p. 305-330.
- [50] Chendy K. M., Misra J., Haas L. M. Distributed deadlock detection. *ACM Trans. Comput. Syst.*, **1**, N 2, 1983, p. 144-156.
- [51] Cheung T.-Y. Graph traversal techniques and the maximum flow problem in distributed computation *IEEE Trans. Soft. Eng.*, **SE-9**, 1983, p. 504-512.
- [52] Chou C. T., Cidon I., Gopal I. S., Zaks S. Synchronizing asynchronous bounded delay networks. *IEEE Trans. Commun.*, **38**, N 2, 1990, p. 144-147.
- [53] Cidon I. Yet another distributed depth-first search algorithm. *Inf. Proc. Lett.*, **26**, 1988, p. 301-305.
- [54] Coan B. A., Dwork C. Simultaneity is harder than agreement. *Information and Control*, **91**, N 2, 1991, p. 205-231.
- [55] Cormen T. H., Leiserson C. E., Rivest R. L. *Introduction to Algorithms*. McGraw Hills/MIT Press, 1990, 1028 p.
- [56] Coulouris G. F., Dollimore J. *Distributed Systems: Concepts and Design*. Addison-Wesley, 1988, 366 p.
- [57] Cristchlow C., Taylor K. The inhibition spectrum and the achievement of causal consistency. Tech. Rep. 90-1101, Dep. Computer Science, Cornell Univ., 1990.

- [58] Cristian F. Probabilistic clock synchronization. *Distributed Computing*, **3**, 1989, p. 146-158.
- [59] Davidson J. *An Introduction to TCP/IP*. Springer-Verlag, 1988.
- [60] Dijkstra E. W. Co-operating sequential processes. In *Programming Languages*, F. Genuys (ed.), Academic Press, 1968, p. 43-112.
- [61] Dijkstra E. W. Self-stabilization in spite of distributed control. *Commun. ACM*, **17**, 1974, p.643-644.
- [62] Dijkstra E. W. Self-stabilization in spite of distributed control. In *Selected Writing on Computing: A Personal Perspective*, Springer-Verlag, 1982, p. 41-46.
- [63] Dijkstra E. W. Shmuel Safra's version of termination detection. EWD-Note 998, 1987.
- [64] Dijkstra E. W., Lamport L., Martin A. J., Scholten C. S., Steffens E. F. M. On-the-fly garbage collection: An exercise in cooperation. *Commun. ACM*, **21**, N 11, 1978, p. 966-975.
- [65] Dijkstra E. W., Scholten C. S. Termination detection for diffusing computations. *Inf. Proc. Lett.*, **11**, N 1, 1980, p. 1-4.
- [66] Dijkstra E. W., Feijen M. J., van Gasteren A. J. Derivation of a termination detection algorithm for distributed computations. *Inf. Proc. Lett.*, **16**, N 5, 1983, p. 217-219.
- [67] Dobrev S., Ruzicka P., Tel J. Time and bit optimal broadcasting on anonymous unoriented hypercubes. In *Proc. 5th Colloq. on Structural Information and Communication Complexity* (1998), L. Gargano, D. Peled (eds.), Carleton Univ. Press, p. 173-187.
- [68] Dobrev S. Linear election on unoriented hypercubes and graphs of constant diameter with any sense of direction. Tech. Rep. Institute of Informatics, Comenius Univ., Bratislava, 1999.
- [69] Dolev D., Klawe M., Rodeh M. An $O(N \log N)$ unidirectional distributed algorithm for extrema-finding in a circle. *J. Algorithms*, **3**, 1982, p. 245-260.
- [70] Dolev D., Reischuk R., Strong H. R. Eventual is earlier than immediate. In *Proc. 23rd Foundations of Computation Theory*, 1982, p. 196-203.
- [71] Dolev D., Strong H. R. Authenticated algorithms for Byzantine agreement. *SIAM J. Comput.*, **12**, N 4, 1983, p. 656-666.
- [72] Dolev D., Fischer M. J., Fowler R., Lynch N. A., Strong H.R. An efficient algorithm for Byzantine agreement without authentication. *Information and Control*, **52**, 1982, p. 257-274.

- [73] Dolev D., Halpern J. Y., Strong H.R. On the possibility and impossibility of achieving clock synchronization. *Symp. on Theory of Computing*, 1984, p. 504-511.
- [74] Dolev D., Dwork C., Sockmeyer L. On the minimal synchronism needed for distributed consensus. *J. ACM*, **34**, 1987, p. 77-97.
- [75] Dwork C., Lynch N. A., Sockmeyer L. Consensus in the presence of partial synchrony. *J. ACM*, **35**, N 2, 1988, p. 288-323.
- [76] Eilam T., Moran S., Zaks S. A lower bound for linear interval routing. In *Proc. 10th Int. Workshop on Distributed Algorithms* (1996), O. Babaoglu, K. Marzullo (eds.), Lecture Notes in Computer Science, vol. 1151, p. 191-205.
- [77] ElGamal T. A public key cryptosystem and a signature scheme based on discrete logarithm. *IEEE Trans. Inf. Theory*, **IT-31**, N 2, 1985, p. 269-472.
- [78] Fiat A., Shamir A. How to prove yourself: practical solutions to identification and signature problems. In *Proc. Advances in Cryptology* (1986), Odlyzko A. M. (ed.), Lecture Notes in Computer Science, vol. 263, p. 186-194.
- [79] Finn S.G. Resynch procedures and fail-safe network protocol. *IEEE Trans. Commun.*, **COM-27**, 1979, p. 840-845.
- [80] Fisher M. J., Lynch N. A. A lower bound for the time to assure interactive consistency. *Inf. Proc. Lett.*, **14**, 1982, p. 183-186.
- [81] Fisher M. J., Lynch N. A., Paterson M. S. Impossibility of distributed consensus with one faulty process. *J. ACM*, **32**, 1985, p. 374-382.
- [82] Flammini M., Gambosi G., Salomone S. Boolean routing. In *Proc. Int. Workshop on Distributed Algorithms* (Lausanne, 1993), A. Schipper (ed.), Lecture Notes in Computer Science, vol. 725, p. 219-233.
- [83] Flammini M., Leeuwen J. van, Marchetti-Spaccamela A. The complexity of interval routing on random graphs. Tech. Rep. UU-CS-1995-16, Dep. Computer Science, Utrecht University, The Netherlands, 1995.
- [84] Flammini M., Gambosi G., Nanni U., Tan R. B. Multidimensional interval routing schemes. *Theor. Comput. Sci.*, **205**, 1998, p. 115-133.
- [85] Fletcher J. G., Watson R. W. Mechanism for reliable timer-based protocol. *Computer Networks*, **2**, 1978, p. 271-290.
- [86] Flocchini P., Mans B., Santoro N. Sense of direction: definition, properties, and classes. *Networks*, **32**, 1998, p. 165-180.
- [87] Francez N. Distributed termination. *ACM Trans. Program. Lang. Syst.*, **2**, N 1, 1980, p. 42-55.

- [88] Francez N., Roden M. Achieving distributed termination without freezing. *IEEE Trans. Software Eng.*, **SE-8**, N 3, 1982, p. 287-292.
- [89] Francez N. *Fairness*. Springer-Verlag, 1986, 295 p.
- [90] Franklin W. R. On an improved algorithm for decentralized extrema finding in circular configurations of processes. *Commun. ACM*, **25**, 1982, p. 336-337.
- [91] Fredericson G. N., Lynch N. A. A lower for the time to assure interactive consistency. In Proc. *Symp. on Theory of Computing*, 1994, p. 493-503.
- [92] Fredericson G. N. A single source shortest path algorithm for a planar distributed network. In Proc. *Symp. on Theoretical Aspects of Computer Science* (Saarbrücken, 1985), K. Mehlhorn (ed.), Lecture Notes in Computer Science, vol. 182, p. 143-150.
- [93] Fredericson G. N., Janardan R. Designing networks with compact routing tables. *Algorithmica*, **3**, 1988, p. 171-190.
- [94] Freigniaud P., Gavoile C. Interval routing schemes. Research Report 94-01, LIPS-ENS, Lyon, 1994.
- [95] Gafni E. Generalized scheme for topology-update in dynamic networks. In Proc. *2nd Int. Workshop on Distributed Algorithms* (Amsterdam, 1987), J. van Leeuwen (ed.), Lecture Notes in Computer Science, vol. 312, p. 187-196.
- [96] Gallager R. G., Humblet P. A., Spira P. M. A distributed algorithm for minimum weight spanning tree. *ACM Trans. Program. Lang. Syst.*, **5**, 1983, p. 67-77.
- [97] Garay J. A., Perry K. J. A continuum of failure models for distributed computing. In Proc. *6th Int. Workshop on Distributed Algorithms* (Haifa, 1992), S. Zaks, A. Segall (eds.), Lecture Notes in Computer Science, vol. 647, p. 153-165.
- [98] Garay M. R., Johnson D. S. *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, 1979.
- [99] Gasteren A. J., M. van, Tel G. Comments on «On the proof of a distributed algorithm»: Always true is not invariant. *Inf. Proc. Lett.*, **35**, 1990, p. 277-279.
- [100] Gopal A. S., Perry K. J. Unifying self-stabilization and fault-tolerance. In Proc. *12th Symposium on Principles of Distributed Computing* (Ithaca, 1993), p. 195-206.
- [101] Goscinski A. *Distributed Operating Systems: The Logical Design*. Addison-Wesley, 1991, 913 p.

- [102] Harget A. J., Johnson I. D. Load balancing algorithms in loosely coupled distributed systems: a survey. In *Distributed Computer Systems*, H. S. M. Zadan (ed.), Butterworths, 1990, p. 85-108.
- [103] Herman T. Adaptivity through distributed convergence. Ph.D. Thesis, Dept. Computer Science, University of Texas at Austin, 1991.
- [104] Higham L., Przytycka T. A simple efficient algorithm algorithm for maximum finding on rings. In Proc. *7th Int. Workshop on Distributed Algorithms* (1993), A. Schiper (ed.), Lecture Notes in Computer Science, vol. 725, p. 249-263.
- [105] Hirschberg D. S., Sinclair J. B. Decentralized extrema-finding in circular configuration of processes. *Commun. ACM*, **23**, 1980, p. 627-628.
- [106] Hoepman J.-H. Self-stabilizing mutual exclusion on a ring, even if $K = N$. Submitted for publication, 1999.
- [107] Hoare C. A. R. Monitors: an operating system structuring concept. *Commun. ACM*, **17**, 1974, p. 549-557.
- [108] Hsu S.-C., Huang S.-T. A self-stabilizing algorithm for maximal matching. *Inf. Proc. Lett.*, **43**, N 2, 1992, p. 77-81.
- [109] Huang S.-T. A fully distributed termination detection scheme. *Inf. Proc. Lett.*, **29**, N 1, 1988, p. 13-18.
- [110] Israeli A., Jalfon M. Self-stabilizing ring orientation. In Proc. *4th Int. Workshop on Distributed Algorithms* (Bari, 1990), J. van Leeuwen, N. Santoro (eds.), Lecture Notes in Computer Science, vol. 486, p. 1-14.
- [111] Itai A., Rodeh M. Symmetry breaking in distributed networks. *Symp. on Theory of Computing*, 1980, p. 150-158.
- [112] Itai A., Kutten S., Wolfstahl Y., Zaks S. Optimal distributed t -resilient election in complete networks. *IEEE Trans. Soft. Eng.*, **SE-8**, N 4, 1990, p. 415-420.
- [113] Johansen K. E., Jorgensen U. L., Nielsen S. H., Nielsen S. E., Skyum S. A distributed spanning tree algorithm. In Proc. *2nd Int. Workshop on Distributed Algorithms* (Amsterdam, 1987), J. van Leeuwen (ed.), Lecture Notes in Computer Science, vol. 312, p. 1-12.
- [114] Katz S., Perry K. J. Self-stabilizing extensions for message-passing systems. *Distributed Comput.*, **7**, N 1, 1993, p. 17-26.
- [115] Keller R. Formal verification of parallel programs. *Commun. ACM*, **19**, N 7, 1976, p. 371-384.

- [116] Krousis L. M., Kranakis E. A brief survey of concurrent readers and writers. *CWI Quarterly*, **2**, N 4, 1989, p. 307-330.
- [117] Knapp E. Deadlock detection in distributed databases. *Computing surveys*, **19**, N 4, 1987, p. 303-328.
- [118] Korach E., Kutten S., Moran S. A modular technique for the design of efficient leader finding algorithm. *ACM Trans. Program. Lang. Syst*, **12**, 1990, p. 84-101.
- [119] Korach E., Moran S., Zaks S. Tight upper and lower bounds for some distributed algorithms for a complete network of processes. In *Proc. Symp. on Principles of Distributed Computing*, 1984, p. 199-207.
- [120] Korach E., Moran S., Zaks S. The optimality of distributive constructions of minimum weight and degree restricted spanning trees in a complete network of processes. In *Proc. 4th Symp. on Principles of Distributed Computing*, 1985.
- [121] Kranakis E., Krizanc D. Computing boolean functions on anonymous hypercube networks. Report CS-R9040, Center for Mathematics and Computer Science, Amsterdam, 1990.
- [122] Lai T. H., Yang T. H. On distributed snapshots. *Inf. Proc. Lett.*, **25**, 1987, p. 153-158.
- [123] Lakshmanan K. B., Meenakshi N., Thulasiraman K. A time optimal message-efficient distributed algorithm for depth-first search. *Inf. Proc. Lett.*, **25**, 1987, p. 103-109.
- [124] Lamport L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, **21**, 1978, p. 558-564.
- [125] Lamport L. An assertional correctness proof of a distributed algorithm. *Sci. Computer Programming*, **2**, 1982, p. 175-206.
- [126] Lamport L., Shostak R., Pease M. The Byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, **4**, 1982, p. 382-401.
- [127] Leeuwen J. van, Tan R. B. Computer networks with compact routing tables. In *The Book of L*, G. Rozenberg, A. Salomaa (eds.), Springer-Verlag, 1986, p. 259-273.
- [128] Leeuwen J. van, Tan R. B. Interval routing. *Computer J.*, **30**, 1987, p. 298-307.
- [129] Leeuwen J. van, Santoro N., Urrutia J., Zaks S. Guessing games and distributed computations in synchronous networks. In *Proc. Int. Colloq. Automata, Languages, and Programming* (Karlsruhe, 1987), Lecture Notes in Computer Science, vol. 267, p. 347-356.

- [130] Leiserson C. E., Abuhamden Z. S., Douglas D. C., Feynman C. R., Ganmukhi M. N., Hill J. V., Hillis W. D., Kuszmaul B. C., St. Pierre M. A., Wells D. S., Wong M. C., Yang S.-W., Zak R. The network architecture of the Connection Machine CM-5. In *Proc. of 4th Symp. on Parallel Algorithms and Architectures* (San Diego, 1992), L. Snyder (ed.), p. 272-285.
- [131] LeLann G. Distributed systems: toward a formal approach. In *Proc. Information Processing'77* (1977) B. Gilchrist (ed.), North-Holland, p. 155-160.
- [132] Lentfert P. J. A., Uittenbogaard A. H., Swierstra S. D., Tel J. Distributed hierarchical routing. Tech. Rep. RUU-CS-89-5, Dept. Computer Science, Utrecht University, Netherlands, 1989.
- [133] Liskov B., Ladin R. Highly-available distributed services and fault-tolerant distributed garbage collection. In *Proc. 5th Symp. on Principles of Distributed Computing* (Vancouver, 1986), p. 29-39.
- [134] Loui M. C., Matsushita T. A., West D. B. election in a complete network with a sense of direction. *Inf. Proc. Lett.*, **22**, 1986, p. 185-187. Addendum: *Inf. Proc. Lett.*, **28**, 1988, p. 327.
- [135] Lynch W. C. Reliable full-duplex file transmission over half-duplex telephone lines. *Commun. ACM*, **11**, N 6, 1968, p. 407-410.
- [136] Lynch N. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.
- [137] Mahany S. R., Schneider F. B. Inexact agreement: accuracy, precision and graceful degradation. In *Proc. 4th Symp. on Principles of Distributed Computing*, 1985, p. 237-249.
- [138] Manna Z., Pnueli A. The anchored version of the temporal framework. In *Proc. Symp. on Linear Time, Branching Time, and Partial Order in Logics and Model for Concurrency* (Noordwijkerhout, 1988), J. W. de Bakker, W.-P. de Roever and G. Rozenberg (eds.), Lecture Notes in Computer Science, vol. 354, Springer-Verlag, p. 201-284.
- [139] Masuzawa T., Nishikawa N., Hagihara K., Tokura N. Optimal fault tolerant distributed algorithms for election in complete networks with a global sense of direction. In *Proc. 3rd Int. Workshop on Distributed Algorithms* (Nice, 1989), J.-C. Bermond and M. Raynal (eds.), Lecture Notes in Computer Science, vol. 392, p. 171-182.
- [140] Mattias Y., Afek Y. Simple and efficient election algorithms for anonymous networks. In *Proc. 3rd Int. Workshop on Distributed Algorithms* (Nice, 1989), J.-C. Bermond and M. Raynal (eds.), Lecture Notes in Computer Science, vol. 392, p. 183-194.
- [141] Mattern F. Algorithms for distributed termination detection. *Distributed Computing*, **2**, N 3, 1987, p. 161-175.

- [142] Mattern F. Global quiescence detection based on credit distribution and recovery. *Inf. Proc. Lett.*, **30**, N 4, 1989, p. 195-200.
- [143] Mattern F. Message complexity of simple ring-based election algorithm: an empirical analysis. In *Proc. 9th Int. Conf. on Distributed Computer Systems*, 1989, p. 94-100.
- [144] Mattern F. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms* (1989), M. Cosnard et al. (eds.), Elsevier Science Publishers, p. 215-226.
- [145] McHugh J. A. *Algorithmic Graph Theory*, Prentice-Hall, 1990.
- [146] Menasce D., Muntz R. Locking and deadlock detection in distributed databases. *Commun. ACM*, **21**, 1979.
- [147] Merlin P. M., Segall A. A failsafe distributed routing protocol. *IEEE Trans. Comp.*, **COM-27**, N 9, 1979, p. 1280-1287.
- [148] Merlin P. M., Schweitzer P.J. Deadlock avoidance in store-and-forward networks I: Store-and-forward deadlock. *IEEE Trans. Commun.*, **COM-28**, N 3, 1980, p. 345-354.
- [149] Merlin P. M., Schweitzer P.J. Deadlock avoidance in store-and-forward networks II: Other deadlock types. *IEEE Trans. Commun.*, **COM-28**, N 3, 1980, p. 355-360.
- [150] Mitchell D. P., Merrit M. J. A distributed algorithm for deadlock detection and resolution. In *Proc. 3rd Symp. on Principles of Distributed Computing*, 1984, p. 282-284.
- [151] Moran S., Wolfstahl Y. Extended impossibility results for asynchronous complete networks. *Inf. Proc. Lett.*, **26**, 1987, p. 145-151.
- [152] Naimi N. Global stability detection in the asynchronous distributed computations. In *Proc. Future Trends of distributed Computing Systems in the Nineties* (Hong Kong, 1988), p. 87-92.
- [153] Natarajan N. A distributed scheme for detecting communication deadlocks. *IEEE Trans. Soft. Eng.*, **SEM-12**, N 4, 1986, p. 531-537.
- [154] Odlyzko A. M. Discrete logarithms in finite fields and their cryptographic significance. In *Proc. Advances in Cryptology* (Paris, 1984), Lecture Notes in Computer Science, vol. 209, p. 224-314.
- [155] Owicki S., Gries D. Verifying properties of parallel programs: An axiomatic approach. *Commun. ACM*, **19**, N 5, 1976, p. 279-285.
- [156] Pachl J., Korach E., Rotem D. Lower bounds for distributed maximum algorithms. *J. ACM*, **31**, 1984, p. 905-918.

- [157] Pease M., Shostak R., Lamport L. Reaching agreement in the presence of faults. *J. ACM*, **27**, 1984, p. 228-234.
- [158] Peleg D. Time-optimal leader election in general networks. *J. Parallel and Distributed Computing*, **8**, N 1, 1990, p. 96-99.
- [159] Peterson G. L. An $O(n \log n)$ unidirectional algorithm for circular extrema problem. *ACM Trans. Program. Lang. Syst.*, **4**, 1982, p. 758-762.
- [160] Peterson G. L. Efficient algorithms for elections in meshes and complete networks. Tech. Rep. TR 140, Dep. Computer Science, University of Rochester, Rochester, NY, 14627, 1985.
- [161] Rana S. P. A distributed solution of the distributed termination problem. *Inf. Proc. Lett.*, **17**, 1983, p. 43-46.
- [162] Riccardi A., Birman K., Stephenson P. The cost of order in asynchronous systems. In *Proc. 6nd Int. Workshop on Distributed Algorithms* (Haifa, 1992), S. Zaks, A. Segall (eds.), Lecture Notes in Computer Science, vol. 647, p. 329-345.
- [163] Rivest R., Shamir A., Adleman L. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, **21**, 1978, p. 120-126.
- [164] Rudolph E., Graubmann P., Grabowski J. Tutorial on message charts. *Comput. Networks ISDN Syst.*, **28**, 1996, p. 1629-1641.
- [165] Ruzicka P. On efficiency of interval routing algorithms. In *Proc. Mathematical Foundations of Computer Science* (1988), M. P. Chitill, L. Janiga, V. Koubek (eds.), Lecture Notes in Computer Science, vol. 324, Springer-Verlag, p. 492-500.
- [166] Ruzicka P. Efficient communication schemes. In *Proc. SOFSEM'98 Theory and Practice of Informatics* (1998), B. Rován (ed.), Lecture Notes in Computer Science, vol. 1521, Springer-Verlag, p. 244-263.
- [167] Saad Y., Schultz M. H. Topological properties of hypercubes. *IEEE Trans. Comput.*, **C-37**, N 7, 1988, p. 867-872.
- [168] Santoro N. Sense of direction, topological awareness, and communication complexity. *ACM SIGAT News*, **16**, 1984, p. 50-56.
- [169] Santoro N., Khatib R. Labelling and implicit routing in networks. *Computer J.*, **28**, 1984, p. 5-8.
- [170] Santoro N., Rotem D. On the complexity of distributed elections in synchronous graphs. In *Proc. Int. Workshop on Graph-Theoretic Concepts in Computer Science* (1985), H. Noltemeier (ed.), Trauner Verlag, p. 337-346.

- [171] Schieber R., Snir M. Calling names on nameless networks. In Proc. 8-th *Symp. on Principles of Distributed Computing* (Edmonton, 1989), p. 319-329.
- [172] Schiper A., Eggli J., Sandoz A. A new algorithm to implement causal ordering. In Proc. 3rd *Int. Workshop on Distributed Algorithms* (Nice, 1989), J.-C. Bermond, M. Raynal (ed.), Lecture Notes in Computer Science, vol. 392, p. 219-232.
- [173] Scoone A. A. Assertion verification in distributed computing. Ph.D. thesis, Dept. Computer Science, Utrecht University, The Netherlands, 1991, 191 p.
- [174] Segall A. Distributed network protocols. *IEEE Trans. Inf. Theory*, **IT-29**, 1983, p. 23-35.
- [175] Shavit N., Francez N. A new approach to the detection of locally indicative stability. In Proc. *Int. Colloq. Automata, Languages, and Programming* (1986), L. Kott (ed.), Lecture Notes in Computer Science, vol. 226, p. 344-358.
- [176] Sloman M., Kramer J. *Distributed Systems and Computer Networks*. Prentice Hall International, 1987, 336 p.
- [177] Spector A., Gifford D. The Space Shuttle computer system. *Commun. ACM*, **27**, 1984, p. 874-900.
- [178] Spirakis P., Tampakas B. Efficient distributed algorithms by using the Archimedean time assumption. *Informatique Theorique et Applications*, **23**, N 1, 1989, p. 113-128.
- [179] Stefankovic D. Acyclic orientations do not lead to optimal deadlock-free packet routing algorithms. *Inf. Proc. Lett.*, 2000.
- [180] Tajibnapis W. D. A correctness proof of topology information maintenance protocol for distributed computer networks. *Commun. ACM*, **20**, N 7, 1977, p. 477-485.
- [181] Tan R. B., van Leeuwen J. General symmetric distributed termination detection. Tech. Rep. RUU-CS-86-2, Dep. Computer Science, Utrecht University, Netherlands, 1986.
- [182] Tanenbaum A.S. *Computer Networks*, 3rd edition. Prentice Hall, 1996.
- [183] Tarry G. Le probleme des labyrinthes. *Nouvelles Annales de Mathematiques*, **14**, 1895.
- [184] Tarjan R. E. Depth-first search and linear graph algorithms. *SIAM Computer J.*, **1**, 1972, p. 146-160.

- [185] Taubenfeld G. Moran S. Possibility and impossibility results in a shared memory environment. In Proc. 3rd *Int. Workshop on Distributed Algorithms* (Nice, 1989), J.-C. Bermond and M. Raynal (eds.), Lecture Notes in Computer Science, vol. 392, p. 254-267.
- [186] Taylor K. The role of inhibition in asynchronous consistent-cut protocols. In Proc. 3rd *Int. Workshop on Distributed Algorithms* (Nice, 1989), J.-C. Bermond and M. Raynal (eds.), Lecture Notes in Computer Science, vol. 392, p. 280-291.
- [187] Tel G. *Topics on Distributed Algorithms*, vol. 1 of *Cambridge Int. Series on Parallel Computation*. Cambridge University Press, 1991, 240 p.
- [188] Tel G. Fouttolerantie in gedistribueerde algoritmen. Lecture Note INF/DOC-91-02, Dept. Computer Science, Utrecht University, Netherlands, 1991.
- [189] Tel G., Mattern F. The derivation of termination detection algorithms from garbage collection schemes. *ACM Trans. Program. Lang.*, **15**, N 1, 1993, p. 1-35.
- [190] Tel G. Network orientation. *Int. J. Foundations Comput. Sci.*, **5**, N 1, 1994, p. 23-57.
- [191] Tel G., Korach E., Zaks S. Synchronizing ABD-networks. *IEEE Trans. Networking*, **2**, N 1, 1994, p. 66-69.
- [192] Tel G. Sense of direction in processor networks. In Proc. *SOFSEM'95: Theory and Practice of Informatics* (Milovy, Czech Rep., 1995), M. Bartosek, J. Staudek, J. Wiedermann (eds.), Lecture Notes in Computer Science, vol. 1012, p. 50-82.
- [193] Topor R. W. Termination detection for distributed computation. *Inf. Proc. Lett.*, **18**, 1984, p. 33-36.
- [194] Toueg S. An all-pairs shortest-path distributed algorithm. Tech. Rep. RC 8237, IBM T. J. Watson Research Center, Yorktown Heights, NY 10598, 1980.
- [195] Toueg S. Deadlock- and livelock-free packet switching networks. In *Proc. of Symp. on Theory of Computing* (1980), p. 94-99.
- [196] Toueg S., Ullman J. D. Deadlock-free packet switching networks. *SIAM J. Comput.*, **10**, N 3, 1981, p. 594-611.
- [197] Verweij A. M., Tel J. A Monte-Carlo algorithm for election. In Proc. 2nd *Colloq. on Structural Information and Communication Complexity* (Olympia, Greece, 1995), L. M. Kirousis, E. Kranakis (eds.), Carleton Univ. Press, p. 77-88.

- [198] Vitanyi P. M. Time-driven algorithms for distributed control. Tech. Rep. CS-R8510, Center for Mathematics and Computer Science. Amsterdam, 1985.
- [199] Watson R. W. Timer-based mechanisms unreliable transport protocol connection management. *Computer Networks*, **5**, 1981, p. 47-56.
- [200] van Wezel M. C., Tel J. An assertional proof of Rana's algorithm. *Inf. Proc. Lett.*, **49**, N 5, 1994, p. 227-233.

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

- || (concurrent), 69
- ~ (equivalent executions), 71
- $\mathbb{M}A$ (multisets), 57
- $\prec$, 68, 194
- H_n (harmonic number), 249
- $\rightarrow$ (transition relation), 56
- $\triangleleft$ (prefix), 161
- 0/1-decided configuration, 456
- 0/1-valent configuration, 456
- algorithm
 - Chang, echo, 206
- ABD network, 419
- accuracy (failure det.), 533
- acknowledgement, 89, 98
- action, 582
- acyclic orientation cover, 178
- addressing, 82, 580
- adjacent nodes, 585
- agreement
 - for broadcast algorithm, 492
 - for consensus algorithm, 456
 - for interactive consistency, 516
- algorithm
 - Attiya (renaming), 466
 - Awerbuch, DFS, 224
 - Awerbuch—Gallager, 439
 - Bracha—Toueg (Byzantine broadcast), 485
 - Bracha—Toueg (consensus), 474, 479
 - Chandy—Lamport, 357
 - Chandy—Misra, 134, 168
 - Chang—Roberts, 246
 - Chang—Roberts, 339
 - Cidon, depth-first search, 226
 - credit-recovery, 314
 - deterministic, 324
 - Dijkstra—Feijen—Van Gasteren, 300
 - Dijkstra—Scholten, 289
 - distributed (definition), 57
 - Dolev—Klawe—Rodeh, 250
 - fast-convergence, 521
 - Finn, 212, 244
 - Fischer (knot agreement), 462
 - Floyd—Warshall, 123
 - Gallager—Humblet—Spira, 261
 - global-marking, 368
 - Itai—Rodeh (election), 339
 - Itai—Rodeh (ring size), 345
 - Korach—Kutten—Moran, 273
 - Lai—Yang, 359
 - LeLann, 245
 - local (definition), 57
 - Merlin—Segall, 133
 - Netchange, 137, 168, 550, 575
 - Peleg (election), 244
 - Peterson, 250, 278
 - Rana, 317
 - Safra, 304
 - Santoro—Khatib, 148, 434
 - Shavit—Francez, 294
 - Tarry, traversal, 218
 - Toueg, 126, 419
 - Update, 573
 - vector-counting, 308
- alive (of process), 368
- alpha synchronizer, 430
- alpha-time complexity, 235
- alternating-bit protocol, 98
- ancestor, 588
- AND model (deadlock), 372
- anonymous network, 82, 322, 323, 553
- applicable action, 582
- application layer, 35
- approximate agreement, 450, 465
- architecture, 32
- ARPANET network, 19
- assertion, 62

asynchronous bounded-delay (ABD) network, 419, 517
 authentication (of a message), 504
 Awerbuch's DFS algorithm, 224
 Awerbuch–Gallager BFS algorithm, 439

 backward-count controller, 183
 backward-state controller, 185
 balanced sliding-window protocol, 91
 basic algorithm, 284, 366
 basic computation, 284, 366
 basic message, 284, 357
 benign failure, 448
 beta synchronizer, 430
 BFS tree, 433
 bifurcated routing, 123
 bipartite, 598
 bivalent configuration, 456
 body (of an action), 582
 boolean routing, 160
 bottom of a partial order, 236
 bounded drift, 518
 breadth-first search tree, 433
 broadcast message, 39
 broadcasting, 23
 browser, 20
 buffer graph, 173
 buffer-graph controller, 174
 bus, 22
 Byzantine broadcast, 483, 492
 Byzantine process, 447
 Byzantine-robust consensus, 477

 capacity (of a channel), 80
 causal order, 68
 causal order ($\prec$, $\preceq$), 68
 — local ($\preceq_\rho$), 352
 causality chain, 69
 centralized algorithm, 195, 285
 chain-time complexity, 235
 Chandy–Lamport algorithm, 357
 Chandy–Misra algorithm, 134, 168
 Chang's echo algorithm, 206
 Chang–Roberts algorithm, 246, 339
 channel, 77
 child, 588
 chordal ring, 378
 Cidon's DFS algorithm, 226
 clique, 78, 590
 — election in, 278, 442
 — sense of direction in, 595
 — wave algorithm for, 208
 clock, 74, 317, 517
 clock synchronization, 518
 — deterministic, 519
 — probabilistic, 520
 clustering, 164
 coding in time, 418
 collateral composition, 564
 collect . . . from, 535
 coloring (of a planar graph), 566
 commit–abort, 449
 communication deadlock, 372
 communication failure, 79
 compact routing, 147
 — interval routing, 151
 — prefix routing, 160
 — tree-labeling scheme, 148
 comparison algorithm, 241, 426
 complete graph, 590
 completeness (failure det.), 532
 computation, 72
 — synchronous, 418
 computation forest, 294
 computation graph, 289
 computation tree, 289
 concurrent events ($\parallel$), 69
 configuration, 56
 — 0/1-decided, 456
 — 0/1-valent, 456
 — bivalent, 456
 — deadlock, 173, 368
 — decided, 456
 — initial, 56
 — legitimate, 547
 — livelock, 190
 — of a probabilistic algorithm, 325
 — possible, 355
 — symmetric, 332
 — synchronous system, 417
 — terminal, 56
 — univalent, 457
 congestion, 21
 connected component, 585
 connected graph, 585

connection management, 86, 103
 consensus, 449
 consensus algorithm, 456
 — agreement, 456
 — convergence, 472
 — dependence, 483
 — non-triviality, 456
 — termination, 456
 consensus problem, 449
 consistency for decision task, 448, 464
 consistent cut, 355
 consumption of a packet, 172
 control algorithm, 284
 control message, 284
 control-oriented notation, 581
 controller, 172
 — acyclic orientation cover, 179
 — backward-count, 183
 — backward-state, 185
 — buffer-graph, 174
 — deadlock-free, 173
 — destination scheme, 176
 — forward-count, 182
 — forward-state, 183
 — hops-so-far scheme, 177
 — hops-to-go scheme, 177, 192
 — livelock-free, 190
 convergence (consensus alg.), 472
 copy-release deadlock, 188
 crash model, 447
 crash-robust consensus, 472
 creation of a message, 79
 credit-recovery algorithm, 314
 critical section, 550
 CSMA/CD, 37
 cut, 354
 — consistent, 355
 cycle, 585
 cycle-free routing tables, 121
 cycle-increase, 571
 cyclic function, 333
 cyclic interval, 148
 cyclic wait, 24

 data-link layer, 33, 87
 deadlock, 172, 281, 368
 — AND model, 372
 — communication deadlock, 372
 — copy-release, 188
 — OR model, 372
 — pacing, 189
 — progeny, 188
 — reassembly, 189
 — resource, 371
 — store-and-forward, 170
 deadlock detection, 24, 363, 366
 deadlock freeness, 64
 debugging, 352
 decentralized algorithm, 195, 285
 decided configuration, 456
 decision event, 194
 decision graph, 469
 decision problem, 448
 decision task, 464
 degree, 585
 dependence
 — for broadcast algorithm, 492
 — for consensus algorithm, 483
 — for interactive consistency, 516
 depth, 588
 depth-first search, 221
 — Awerbuch's algorithm, 224
 — Cidon's algorithm, 226
 — DFS tree, 221, 589
 — interval labeling scheme (ILS), 152
 — Update algorithm, 576
 — with sense of direction, 380
 derivate, 63, 84
 descendant, 588
 destination, 58
 deterministic algorithm, 324
 diffusing computation, 285, 289
 digital signature, 504
 Dijkstra–Feijen–Van Gasteren algorithm, 300
 Dijkstra–Scholten algorithm, 289
 direct addressing, 82, 580
 directed graph, 586
 directed network
 — wave algorithm for, 209
 direction, sense of, 593
 discrete logarithm, 509
 — ElGamal's scheme, 510
 distance, 585

distributed algorithm (definition), 57
distributed database, 20, 371
— commit-abort, 449
— deadlock, 372
distributed system, 15
Dolev–Klawe–Rodeh algorithm, 250
drift, 112, 423, 518
duplication of a message, 79, 237
dynamic topology, 78, 548

early stopping, 516
echo algorithm, 237, 261
edge, 584
edge function, 571
election, 23, 239, 450
— $[k, l]$ -election, 488
— Chang–Roberts alg., 246
— fault-tolerant, 463
— Gallager–Humblet–Spira alg., 261
— in hypercube, 280
— in planar graph, 280
— in torus, 280
— Korach–Kutten–Moran alg., 273
— LeLann’s alg., 245
— match-making (hypercube), 394
— Peterson’s alg., 250
— using traversal, 273
ElGamal’s signature scheme, 509
empty path, 571
emulation (failure det.), 543
enabled action, 582
environment, 289
equivalence of executions ($\sim$), 71
Euler’s phi function, 511
event, 56, 58
— corresponding, 66
— decision, 194
— internal, 57
— postshot, 353
— preshot, 353
— receive, 57
— send, 57
event-driven notation, 582
eventual strong accuracy, 533
eventual weak accuracy, 533
eventually perfect failure det., 533
eventually strong failure det., 533

execution, 56
exhaustive set, 255
explicit termination, 281
extinction, 259
extrema-finding problem, 241

failure detector, 530
— history, 532
failure pattern, 531
fair collateral composition, 565
fair execution (t -crash), 456
fair execution (t -initially-dead), 461
fair scheduling, 472
fairness assumption, 60, 93
fast-convergence algorithm, 521
father, 588
fault model, 446
— Byzantine, 447
— crash, 447
— hierarchy, 447
— initially dead, 447
— omission, 447
— timing, 448
fault tolerance, 455
— stabilizing algorithm, 548
feasible snapshot, 354
Fiat–Shamir signature scheme, 512
fifo channel, 79
file-transfer protocol (FTP), 33
Finn’s algorithm, 237, 244
firing-squad problem, 515
Floyd–Warshall algorithm, 123
flush channel, 80
follower, 195
forall statement, 579
forest, 589
fork, 457
forward-count controller, 182
forward-state controller, 183
forwarding of a packet, 171
fragment of an MST, 263
frame, 34
fromthis (in receive statement), 580
frond edge, 152, 588
full channel, 80

Gallager–Humblet–Spira MST algorithm, 261

gamma synchronizer, 431
garbage collection, 29, 364
garbling of a message, 79, 87
gateway, 19
general, 483, 492
generation of a packet, 171
generator (of a group), 509
global state, 56
— lack of, 40
global time, 415
— lack of, 41
global virtual time, 366
global-marking algorithm, 368
graceful degradation, 445
graph, 77, 584
greedy routing, 385
grid, 591
— routing, 158
group (algebra), 375
group membership, 452
group sense of direction, 376
guaranteed path, 174
guard (of an action), 582

Hamiltonian cycle, 587
harmonic number, 249, 279
heartbeat failure detector, 543
hierarchical routing, 164
hops-so-far scheme, 177
hops-to-go scheme, 177, 192
hypercube, 78, 592
— anonymous, 338
— controller for, 192
— election in, 280, 442
— interval routing in, 159
— match-making, 394
— sense of direction in, 217, 596
— traversal algorithm for, 217

identity, 81
IEEE standards, 32, 36
ILS, interval labeling scheme, 151
image, 174
IMP (interface message processor), 19
implicit message, 418, 424
implicit termination, 281
in-neighbor, 586
incident edge, 585

incoming edge, 586
index, 509
indirect addressing, 82, 580
induced subgraph, 585
inexact agreement, 521
infimum computation, 200
infimum theorem, 202
initial configuration, 56, 57
initially dead process, 447
initiator, 195
input collection, 334
interactive consistency, 516
interface message processor (IMP), 19
internal events, 57
Internet, 19
interval labeling scheme, 151, 237
— depth-first search, 152
— for a grid, 158
— for a hypercube, 159
— for a ring, 158
— for an outerplanar network, 159
— linear, 157
— valid, 151
interval routing, 151
— efficiency of, 155
— linear, 157
— multi-dimensional, 160
— neighborly, 156
— optimal, 157
invariant, 62, 84
ISO standard, 32
Itai–Rodeh algorithm (election), 339
Itai–Rodeh algorithm (ring size), 345

key (for signatures), 508
Korach–Kutten–Moran algorithm, 273
Kruskal’s MST algorithm, 263

Lai–Yang algorithm, 359
Lamport’s logical clock, 75, 319
Las Vegas algorithm, 328, 472
layer, 31
leader, 240
legitimate configuration, 547
LeLann’s algorithm, 245
length, 585
length-increase, 571
lexicographic order, 84

lieutenant, 492
 line of processors, 324
 linear ILS, 155
 linear-interval routing, 157
 link register, 550
 livelock, 190
 liveness, 61, 64
 load balancing, 27
 local algorithm, 57
 local causal order, 352
 local snapshot, 352
 local-area network, 18, 21
 local-area network (LAN), 16
 locking (in database), 371
 lockstep operation, 415
 logical link control sublayer, 37
 long-haul network, 16
 loss of a message, 79
 low-diameter spanning tree, 589

 malign failure, 448
 mask, 401
 match-making algorithm, 394
 matching (in graph), 557
 Mattern's vector clock, 76
 Mattern, credit-recovery algorithm, 314
 maximal matching, 557
 maximum packet lifetime, 99, 188
 meaningful snapshot, 355
 medium-access-control sublayer, 36
 Merlin–Segall algorithm, 133
 message, 57, 579
 — broadcast, 39
 — implicit, 418
 — multicast, 39
 — point-to-point, 39
 — postshot, 354
 — preshot, 354
 message chain, 235
 message passing, 55, 549
 message termination, 281
 message-generation function, 417
 MIMD, 24
 minimal spanning tree (MST), 262
 minimal-path forest, 571
 minimal-path problem, 570
 minimal-weight spanning tree, 262, 589

 minimum-delay routing, 117
 minimum-hop routing, 117
 mixed-failure model, 448
 monitor, 30
 monotonicity, 571
 Monte Carlo algorithm, 328
 MST (minimal spanning tree), 262
 multi-dimensional interval routing, 160
 multicast message, 39
 multiple interval routing, 156
 multiple-instruction, multiple-data (MIMD), 24
 multiple-path routing, 123
 multiprocessor computer, 24
 multisets (MA), 57
 multisource algorithm, 195
 mutual exclusion, 24, 550

 named network, 82
 neigh, 579
 neighbor, 77, 579, 585
 neighbor knowledge, 82
 — depth-first search using, 228
 neighborly ILS, 155
 neighborly interval routing, 156
 Netchange algorithm, 137, 168, 550, 575
 network, 77
 — anonymous, 82, 322
 — asynchronous bounded-delay (ABD), 419
 — dynamic, 548
 — named, 82
 — synchronous, 417
 network layer, 34
 node, 15, 584
 non-determinism, 41, 324
 non-triviality, 448
 — for consensus algorithm, 456
 — for decision task, 469
 norm function, 64, 144, 154

 omission failure, 447
 one-time complexity, 233
 open-systems interconnection (OSI), 33
 optimal ILS, 155
 optimal sink tree, 121, 150, 589
 optimistic fault tolerance, 546
 OR model (deadlock), 372

order-preserving renaming, 465, 469
 orientation, 178
 OSI, 33
 out-neighbor, 586
 outerplanar graph, 585
 — interval routing in, 159
 outgoing edge, 586

 pacing deadlock, 189
 packet, 89
 parallel composition, 84
 parallel random-access memory (PRAM), 27
 parent, 588
 partial failure, 16, 445
 path, 585
 Peleg's algorithm (election), 244
 perfect failure detector, 533
 permanent failure, 454
 pessimistic fault tolerance, 546
 Peterson's election algorithm, 250, 278
 physical layer, 33, 36
 piggybacking, 317, 360
 pipe, 30
 planar graph, 585
 — coloring of, 566
 — election in, 280
 point-to-point message, 39
 point-to-point network, 19
 polling algorithm, 208
 possible configuration, 355
 postshot event, 353
 postshot message, 354
 PRAM, 27
 precision, 518
 prefix ($\prec$), 161
 prefix labeling scheme, 161, 236
 — tree, 162
 — valid, 161
 prefix routing, 160
 presentation layer, 35
 preshot event, 353
 Prim's MST algorithm, 263
 primitive root, 509
 private key, 508
 probabilistic algorithm, 323
 probabilistic consensus algorithm, 474, 479

 process, 57
 — deterministic, 324
 — probabilistic, 324
 — synchronous, 417
 process identity, 81
 process termination, 281
 producer–consumer problem, 29
 progeny deadlock, 188
 propagation of information with feedback, 198
 protocol, 32
 pseudo-anonymous network, 349
 pseudo-stabilization, 549
 pseudocode, 579
 public key, 509
 pulse, 416, 491
 pulse complexity, 515
 pulse time, 423

 Rana's algorithm, 317
 randomization, 324
 reachable, 56
 — in Byzantine model, 477
 read-all, 550
 read-one, 550
 real time, 517
 real-time clock, 74
 reassembly deadlock, 189
 receive events, 57
 receive statement, 580
 regular graph, 585
 reliable channel, 79
 remote procedure call, 39
 renaming problem, 465
 — order-preserving, 465, 469
 replay argument, 335, 338
 resilience, 456
 resource allocation, 23
 resource deadlock, 371
 restricted-degree spanning tree, 589
 ring, 77, 587
 — controller for, 179
 — election in, 244, 278
 — orientation, 554
 — routing for, 158
 — sense of direction in, 594
 — termination detection in, 300

— wave algorithm for, 202
 Rivest–Shamir–Adleman scheme, 511
 robust algorithm, 446
 rooted tree, 588
 rotating coordinator, 535
 round complexity, 515
 round-robin, 36
 routing, 20, 117
 — compact, 147
 — hierarchical, 164
 — minimum-delay, 117
 — minimum-hop, 117
 — shortest-path, 117
 — Update algorithm, 575
 routing tables, 116
 RSA signature scheme, 511
 rubber band transformation, 71

safety, 61
 Safrá's algorithm, 304
 Santoro–Khatib tree-labeling, 148, 434
 scalable organization, 22
 secret key, 508
 selective delay, 419, 425
 self-stabilization, 547
 self-stabilizing algorithm, 453
 semaphore, 30
 send event, 57
 send statement, 580
 sending window, 95
 sense of direction, 81, 593
 — group, 376
 — in clique, 595
 — in hypercube, 596
 — in ring, 594
 — in torus, 596
 — ring, 554
 sequence number, 89
 session layer, 35
 shared variables, 55, 550
 Shavit–Francez algorithm, 294
 Sherwood algorithm, 327
 shortest-path routing, 117
 — with Netchange algorithm, 146
 signature scheme, 504
 — ElGamal, 509
 — Fiat–Shamir, 512
 — Rivest–Shamir–Adleman, 511
 signing (of a message), 504
 SIMD, 24
 simple cycle, 585
 simple mail-transfer protocol (SMTP), 33
 simple path, 585
 simultaneity
 — for broadcast algorithm, 492
 — for firing-squad problem, 515
 single-instruction, multiple-data (SIMD), 24
 singlesource algorithm, 195
 sink tree, 121
 — optimal, 121, 150, 589
 six-color theorem, 567
 size of routing table, 164
 SMTP, 33
 snapshot, 351
 — feasible, 354
 — meaningful, 355
 snapshot state, 353
 son, 588
 source of a packet, 171
 Space Shuttle, 445
 spanning subgraph, 585
 spanning tree, 588
 — low diameter, 589
 square root, 512
 stabilization, 547
 stabilizing algorithm, 453
 stable property, 351, 363, 368
 star, 78, 590
 starter, 195
 state model, 550
 state of a process, 57
 static topology, 78
 step, 459
 store-and-forward deadlock, 170
 store-and-forward network, 21
 strong accuracy, 533
 strong completeness, 532
 strong failure detector, 533
 strong fairness, 60
 strongly connected graph, 586
 subgraph, 585
 subtree, 588
 suitable (buffer), 174

superimposed algorithm, 284
 symmetric configuration, 332
 symmetric system, 324
 synchronization, 23, 199, 427
 — ABD network, 420
 — for fault tolerance, 452
 synchronizer, 416, 420, 451
 synchronous computation, 418
 synchronous message passing, 58
 synchronous process, 417
 synchronous system, 416

Tarry's traversal алгоритм, 218
 task, 464
 — connected, 469
 — decision graph of, 469
 TCP/IP protocols, 33
 terminal configuration, 56, 281
 termination, 448
 — for broadcast algorithm, 492
 — for consensus algorithm, 456
 — for decision task, 464
 — for firing-squad problem, 515
 — for interactive consistency, 516
 termination detection, 23, 281, 363
 — with token algorithm, 299
 — with wave algorithm, 286
 time complexity, 221
 — alpha-time, 235
 — chain-time, 235
 — one-time, 233
 timer, 99
 timing error, 448
 token algorithm
 — for election, 245, 273
 — for ring, 202
 — for termination detection, 299
 — traversal, 215
 token bus, 36
 token ring, 22, 36
 topological awareness, 593
 topological information, 81
 topology, 77, 584
 torus
 — anonymous, 338
 — election in, 278, 442
 — sense of direction in, 216, 596
 — traversal algorithm for, 216
 Toueg's algorithm, 126, 419
 trace, 255, 334
 transaction, database, 371
 transient failures, 546
 transition, 56, 57
 — for probabilistic algorithm, 325
 — for stabilizing system, 547
 — for synchronous system, 418
 transport layer, 34, 87
 Transputer, 25, 160, 323
 traversal algorithm, 193, 196, 215
 — election with, 273
 tree, 77, 588
 — controller for, 181
 — election for, 242
 — routing for, 148
 — wave algorithm for, 203
 tree edge, 588
 tree-labeling scheme, 148, 434

underlying computation, 284
 undirected graph, 584
 uniform stabilizing algorithm, 552
 univalent configuration, 457
 UNIX system, 19
 Update algorithm, 573
 UUCP network, 19

validity for firing-squad problem, 515
 vector clock, 76
 vector order, 76
 vector-counting algorithm, 308
 virtual ring, 587
 visit (of a wave), 312

wait-free atomic variables, 29
 wave algorithm, 193
 — centralized versus decentralized, 231
 — definition, 194
 — echo algorithm, 206
 — for clique, 208
 — for directed network, 209
 — for hypercube, 217
 — for ring, 202, 258
 — for torus, 216
 — for tree, 203
 — overview, 230

— phase algorithm, 209	weak fairness, 60
— polling algorithm, 208	weighted graph, 586
— use in election, 242	well-founded set, 64, 84
— use in termination detection, 286, 295, 312	wide-area network, 18, 19
weak accuracy, 533	wide-area network (WAN), 16
weak completeness (failure det.), 543	window, 95
	witness, 474