Распределенные системы

(Крюков В.А., Бахтин В.А.)

Распределенная система - совокупность независимых компьютеров, которая представляется пользователю единым компьютером (metacomputer).

Примеры: сеть рабочих станций, банк с множеством филиалов, система резервирования авиабилетов, распределенная операционная система.

Введение в ОС однопроцессорных ЭВМ.

Два взгляда:

- менеджер ресурсов;
- один слой в множестве слоев абстрактных машин.

Представление ОС как менеджера ресурсов

Управление процессами Управление процессорами	Управление файлами Управление памятью	 Управление устройствами
Процессоры	Память	Устройства

Представление ОС как абстрактной машины

Абстрактная машина		
Интерфейс пользователя Языки управления заданиями Командные языки Окна, меню, пиктограммы	Интерфейс программы Система команд	Системные вызовы Процессы Память Файлы Информационные функции

Место ОС среди ПО

Прикладное ПО (отдельные приложения, пакеты прикладных программ, информационные системы, САПР)
Системное ПО (**ОС** + системы программирования, СУБД, графические библиотеки, сервисные программы)

История ОС.

1940-е и 1950-е

"Персональные ЭВМ" - "пультовый режим"

Библиотека программ ввода-вывода, служебная программа.

Инструкция оператору.

Середина 1950-х

Пакетная обработка. Однопрограммный и мультипрограммный режимы.

Инструкция оператору -> паспорт задачи (простейший язык управления заданиями).

Требования к аппаратуре:

- защита памяти;
- прерывания;
- привилегированный режим;
- таймер.

Как обеспечить мультипрограммный режим без таких механизмов.

Середина 1960-х

Режим разделения времени.

Терминалы, квантование, свопинг, страничная и сегментная организация (общие сегменты памяти).

1970-е

Многопроцессорные ЭВМ, многомашинные комплексы, сети ЭВМ.

1980-е

Персональные ЭВМ.

1990-е

MPP, открытые системы, Internet.

2000-е

Кластеры, распределенные системы, GRID, многоядерные и многопоточные процессоры.

2010-е

Гетерогенность – универсальные ЦПУ + ускорители

***********Лекция 2**

1 Введение в параллельные и распределенные системы

1.1 Достоинства многопроцессорных систем с общей памятью (мультипроцессоров)

(1) Производительность

(2) Надежность

1.2. Недостатки

(1) ПО (приложения, языки, ОС) сложнее, чем для однопроцессорных ЭВМ

(2) Ограниченность при наращивании (физ. размеры - близость к памяти, когерентность КЭШей, 64 процессора - максимально достигнутое).

1.2 Достоинства распределенных систем

Почему создаются распределенные системы? В чем их преимущества перед централизованными ЭВМ?

1-ая причина - экономическая.

Закон Гроша (Herb Grosh) - быстродействие процессора пропорционально квадрату его стоимости. С появлением микропроцессоров закон перестал действовать - за двойную цену можно получить тот же процессор с несколько большей частотой.

2-ая причина - можно достичь такой высокой производительности путем объединения микропроцессоров, которая недостижима в централизованном компьютере.

3-я причина - естественная распределенность (банк, поддержка совместной работы группы пользователей).

4-ая причина - надежность (выход из строя нескольких узлов незначительно снизит производительность).

5-я причина - наращиваемость производительности.

Главная причина - наличие огромного количества персональных компьютеров и необходимость совместной работы без ощущения неудобства от географического и физического распределения людей, данных и машин.

Почему нужно объединять РС в сети?

1. Необходимость разделять данные.

2. Преимущество разделения дорогих периферийных устройств, уникальных информационных и программных ресурсов.

3. Достижение развитых коммуникаций между людьми. Электронная почта во многих случаях удобнее писем, телефонов и факсов.

4. Гибкость использования различных ЭВМ, распределение нагрузки.

5. Упрощение постепенной модернизации посредством замены компьютеров.

Недостатки распределенных систем:

1. Проблемы ПО (приложения, языки, ОС).

2. Проблемы коммуникационной сети (потери информации, перегрузка, развитие и замена).

3. Секретность.

1.3 Виды операционных систем (сетевые ОС, распределенные ОС, ОС мультипроцессоров).

Сетевые ОС - машины обладают высокой степенью автономности, общесистемных требований мало. Можно вести диалог с другой ЭВМ, вводить задания в ее очередь пакетных заданий, иметь доступ к удаленным файлам, хотя иерархия директорий может быть разной для разных клиентов. Пример - серверы файлов (многие WS могут не иметь дисков вообще).

Распределенные ОС - единый глобальный межпроцессный коммуникационный механизм, глобальная схема контроля доступа, одинаковое видение файловой системы. Вообще - иллюзия единой ЭВМ.

ОС мультипроцессоров - единая очередь процессов, ожидающих выполнения, одна файловая система.

	Сетевая ОС	Распределенная ОС	ОС мульти процессора
Компьютерная система выглядит как виртуальный мультипроцессор	НЕТ	ДА	ДА
Имеется ли единая очередь выполняющихся процессов	НЕТ	НЕТ	ДА
Имеется хорошо определенная семантика разделения файлов	Обычно НЕТ	ДА	ДА

1.4. Принципы построения распределенных ОС (прозрачность, гибкость, надежность, эффективность, масштабируемость).

(1) Прозрачность (для пользователя и программы).

Прозрачность расположения	Пользователь не должен знать, где расположены ресурсы
Прозрачность миграции	Ресурсы могут перемещаться без изменения их имен
Прозрачность размножения	Пользователь не должен знать, сколько копий существует
Прозрачность конкуренции	Множество пользователей разделяет ресурсы автоматически
Прозрачность параллелизма	Работа может выполняться параллельно без участия пользователя

(2) Гибкость (не все еще ясно - потребуется менять решения).

Использование монолитного ядра ОС или микроядра.

(3) Надежность.

Доступность, устойчивость к ошибкам (fault tolerance).

Секретность.

(4) Производительность.

Грануллированность. Мелкозернистый и крупнозернистый параллелизм (fine-grained parallelism, coarse-grained parallelism).

Устойчивость к ошибкам требует дополнительных накладных расходов.

(*5) Масштабируемость.*

Плохие решения:

- централизованные компоненты (один почтовый-сервер);
- централизованные таблицы (один телефонный справочник);
- централизованные алгоритмы (маршрутизатор на основе полной информации).

Только децентрализованные алгоритмы со следующими чертами:

- ни одна машина не имеет полной информации о состоянии системы;
- машины принимают решения на основе только локальной информации;
- выход из строя одной машины не должен приводить к отказу алгоритма;
- не должно быть неявного предположения о существовании глобальных часов.

Лекция 3

2 Операционные системы мультипроцессорных ЭВМ

Организация ОС:

- главный-подчиненный (master-slave, выделение одного процессора для ОС упрощает ее, но этот процессор становится узким местом с точки зрения загруженности и надежности);
- симметричная (наиболее эффективная и сложная).

2.1 Процессы и нити

Процесс - это выполнение программы. Компоненты процесса - выполняющаяся программа, ее данные, ее ресурсы (например, память), и состояние выполнения.

Традиционно, процесс имеет собственное адресное пространство и его состояние характеризуется следующей информацией:

- таблицы страниц (или сегментов);
- дескрипторы файлов;
- заказы на ввод-вывод;
- регистры;
- и т.п.

Большой объем этой информации делает дорогими операции создания процессов, их переключение.

Потребность в легковесных процессах, нитях (threads), которые совместно используют единое адресное пространство процесса, возникла еще на однопроцессорных ЭВМ (физические процессы или их моделирование, совмещение обменов и счета), но для использования достоинств многопроцессорных ЭВМ с общей памятью они просто необходимы.

Процессы могут быть независимыми, которые не требуют какой-либо синхронизации и обмена информацией (но могут конкурировать за ресурсы), либо взаимодействующими.

2.2. Взаимодействие процессов

Если приложение реализовано в виде множества процессов (или нитей), то эти процессы (нити) могут взаимодействовать двумя основными способами:

- посредством разделения памяти (оперативной или внешней)
- посредством передачи сообщений

При взаимодействии через общую память процессы должны синхронизовать свое выполнение.
Различают два вида синхронизации - взаимное исключение критических интервалов и координация процессов.
Критические секции – частный случай критического интервала, когда критический интервал оформлен в виде конструкции языка программирования. Если не обеспечено взаимное исключение критических интервалов, то недетерминизм, race condition (условия гонок).

Процесс p1 выполняет оператор X = X+1,
а процесс p2 - оператор X = X-1
машинные коды выглядят так:

```
Load  R1,X          Load  R1,X
Add   R1,=”1”  Sub   R1, =”1”
Store R1,X          Store R1,X
```

Результат зависит от порядка выполнения этих команд.

Требуется **взаимное исключение критических интервалов.** Решение проблемы взаимного исключения должно удовлетворять требованиям:

- в любой момент времени только один процесс может находиться внутри критического интервала;
- если ни один процесс не находится в критическом интервале, то любой процесс, желающий войти в критический интервал, должен получить разрешение без какой либо задержки;
- ни один процесс не должен бесконечно долго ждать разрешения на вход в критический интервал (если ни один процесс не будет находиться внутри критического интервала бесконечно долго);
- не должно существовать никаких предположений о скоростях процессоров.

Взаимное исключение критических интервалов в однопроцессорной ЭВМ.

1. Блокировка внешних прерываний (может нарушаться управление внешними устройствами, возможны внутренние прерывания при работе с виртуальной памятью).
2. Блокировка переключения на другие процессы (MONO, MULTI).

Взаимное исключение критических интервалов в многопроцессорной ЭВМ.

Программные решения на основе неделимости операций записи и чтения из памяти (при чтении переменной всегда читаем последнее присвоенное ей значение) .

Алгоритм Деккера (1968).

Два процесса (нити) с номерами i=0 и i=1 в цикле регулярно входят в критический интервал и выходят из него.

```
int turn;
boolean  flag[2 ];
proc( int i )
{
   while (TRUE)
        {
         <вычисления>;
         enter_region( i );
         <критический интервал>;
         leave_region( i );
        }
}

void enter_region( int i )
{
     try:  flag[ i ]=TRUE;
          while (flag [( i+1 ) % 2])
               {
                  if ( turn = = i ) continue;
                  flag[ i ] = FALSE;
                  while ( turn != i );
                  goto try;
                 }

}

void leave_region( int i )
{
          turn = ( i +1 )  % 2;
          flag[ i ] = FALSE;
}

   turn = 0;
```

```
flag[ 0 ] = FALSE;
flag[ 1 ] = FALSE;
proc( 0 ) AND proc( 1 )  /* запустили 2 процесса */
```

Алгоритм Петерсона (1981)

```
int turn;
int flag[ 2 ];

void enter_region( int i )
{
    int other;                    /* номер другого процесса */

    other = 1 - i;
    flag[ i ] = TRUE;
   turn = i;
   while (turn = = i  &&  flag[ other ] = =  TRUE)   /* пустой оператор
     */;
}

void leave_region( int i )
{
   flag[ i ] =  FALSE;
}
```

Использование неделимой операции TEST_and_SET_LOCK.

Операция TSL(r,s): [r = s; s = 1]

Квадратные скобки - используются для спецификации неделимости операций.

```
enter_region:
        tsl  reg, flag
        cmp reg, #0  /* сравниваем с нулем */
        jnz  enter_region   /* если не нуль - повторяем попытку */
        ret
leave_region:
        mov  flag, #0       /* присваиваем нуль*/
        ret
```

Недостатки такого "активного ожидания" при использовании команды TSL или алгоритмов Деккера или Петерсона -

бесполезная трата времени, нагрузка на общую память, и возможность фактически заблокировать работу процесса, находящегося в критическом интервале. Избежать активного ожидания помогают семафоры.

Семафоры Дейкстры (1965).

Семафор - неотрицательная целая переменная, которая может изменяться и проверяться только посредством двух функций:

Функция запроса семафора **P(s):**

[if (s == 0) <заблокировать текущий процесс>; else s = s-1;]

Замечание. Неделимость этой операции означает, что после разблокирования процесса он начнет ее выполнять заново.

Функция освобождения семафора **V(s):**

[if (s == 0) <разблокировать один из заблокированных процессов>;
s = s+1;]

Двоичные семафоры как частный случай общих (считающих).

Использование семафоров для взаимного исключения критических интервалов и для **координации** в задаче производитель-потребитель.

Задача производитель-потребитель (поставщик-потребитель, проблема ограниченного буфера).

```
semaphore  s = 1;
semaphore  full = 0;
semaphore  empty = N;
```

```
producer()
{

    int item;
    while (TRUE)
    {
      produce_item(&item);
      P(empty);
      P(s);
      enter_item(item);
      V(s);
      V(full);
```

```
consumer()
{

    int item;
    while (TRUE)
    {
      P(full);
      P(s);
      remove_item(&item);
      V(s);
      V(empty);
```

```
                                        |           consume_item(item);
    }                                   |       }
}                                       |   }
                                        |
producer() AND consumer()  /* запустили 2 процесса */
```

Реализация семафоров.

Мультипрограммный режим.

- блокировка внешних прерываний;
- запрет переключения на другие процессы;
- переменная и очереди ожидающих процессов в ОС.

Для многопроцессорной ЭВМ первые два способа не годятся. Для реализации третьего способа достаточно команды TSL и возможности объявлять прерывание указанному процессору (чтобы сообщить другим процессорам, что разблокирован один из процессов).

Блокирование процесса и переключение на другой - не эффективно, если семафор захватывается на очень короткое время. Можно ввести специальные семафоры, которые захватываются на короткое время. Ожидание освобождения таких семафоров может быть реализовано в ОС посредством циклического опроса значения семафора.

************Лекция 4**

Если произведенный объект используется многими, то семафоры не годятся.

События.

Это переменные, показывающие, что произошли определенные события.

Для объявления события служит оператор POST(имя переменной), для ожидания события - WAIT (имя переменной). Для чистки (присваивания нулевого значения) - оператор CLEAR(имя переменной).

Варианты реализации - не хранящие информацию (по оператору POST из ожидания выводятся только те процессы, которые уже выдали WAIT), однократно объявляемые (нет оператора чистки).

Метод последовательной верхней релаксации (SOR) с использованием массива событий. В нем для вычисления

элемента матрицы A[i][j] требуется предварительно вычислить элементы A[i-1][j] и A[i][j-1]. Нулевая строка и нулевой столбец матрицы заданы изначально и не вычисляются. Заголовок цикла parfor означает, что витки цикла можно выполнять параллельно.

```
float  A[ L1 ][ L2 ];
struct event  s[ L1 ][ L2 ];
for ( i = 0; i < L1; i++)
     for ( j = 0; j < L2; j++)   { clear( s[ i ][ j ]) };
for ( j = 0; j < L2; j++)   { post( s[ 0 ][ j ]) };
for ( i = 0; i < L1; i++)   { post( s[ i ][ 0 ]) };
.............
.............
parfor ( i = 1; i < L1-1; i++)
     parfor ( j = 1; j < L2-1; j++)
          {  wait( s[ i-1 ][ j ]);
             wait( s[ i ][ j-1 ]);
             A[ i ][ j ] = (A[ i-1 ][ j ] + A[ i+1][ j ] + A[ i ][ j-1 ] + A[ i ][
     j+1 ]) / 4;
             post( s[ i ][ j ]);
          }
```

Обмен сообщениями (message passing)

Хоар (Xoare) 1978 год, "Взаимодействующие последовательные процессы". Цели - избавиться от проблем разделения памяти и предложить модель взаимодействия процессов для распределенных систем.
send (destination, &message, msize);
receive ([source], &message, msize);
Адресат - процесс. Отправитель - может не специфицироваться (любой).
С буферизацией (почтовые ящики) или нет (рандеву - Ада, Оккам).
Пайпы ОС UNIX - почтовые ящики, заменяют файлы и не хранят границы сообщений (все сообщения объединяются в одно большое, которое можно читать произвольными порциями).

Пример использования буферизуемых сообщений.

```
#define  N   100                    /* максимальное число сообщений */
                                    /* в буфере*/
#define  msize  4                   /* размер сообщения*/
typedef  int message[msize];

producer()
 {
     message m;
     int item;

     while (TRUE)
     {
       produce_item(&item);
       receive(consumer, &m, msize); /* получает пустой */
                                     /* "контейнер" */
       build_message(&m, item);      /* формирует сообщение */
       send(consumer, &m, msize);
      }
 }

consumer()
 {
     message m;
     int item, i;

     for (i = 0; i < N; i ++)
       send (producer, &m, msize);   /* посылает все пустые *.
                                     /* "контейнеры" */
     while (TRUE)
     {
       receive(producer, &m, msize);
       extract_item(&m, item);
       send(producer, &m, msize);    /* возвращает "контейнер" */
       consume_item(item);
      }
 }

producer() AND consumer()  /* запустили 2 процесса */
```

Механизмы семафоров и обмена сообщениями взаимозаменяемы семантически и на мультипроцессорах могут быть

реализованы один через другой. Другие классические задачи взаимодействия процессов – “проблема обедающих философов” и “читатели-писатели”.

2.3 Планирование процессоров

Планирование процессоров очень сильно влияет на производительность мультипроцессорной системы. Можно выделить следующие главные причины деградации производительности:

1) Накладные расходы на переключение процессора. Они определяются не только переключениями контекстов процессов, но и (при переключении на процессы другого приложения) перемещениями страниц виртуальной памяти, а также порчей кэша (информация в кэше другому приложению не нужна и будет заменена).

2) Переключение на другой процесс в тот момент, когда текущий процесс выполнял критическую секцию, а другие процессы активно ожидают входа в критическую секцию. В этом случае потери будут велики (хотя вероятность прерывания выполнения коротких критических секций мала).

Применяются следующие стратегии борьбы с деградацией производительности.

1) Совместное планирование, при котором все процессы одного приложения (неблокированные) одновременно выбираются на процессоры и одновременно снимаются с них (для сокращения переключений контекста).
2) Планирование, при котором находящиеся в критической секции процессы не прерываются, а активно ожидающие входа в критическую секцию процессы не выбираются до тех пор, пока вход в секцию не освободится.
3) Процессы планируются на те процессоры, на которых они выполнялись в момент их снятия (для борьбы с порчей кэша). При этом может нарушаться балансировка загрузки процессоров.
4) Планирование с учетом "советов" программы (во время ее выполнения). В ОС Mach имеется два класса таких советов (hints) - указания (разной степени категоричности) о снятии текущего процесса с процессора, а также указания о том процессе, который должен быть выбран взамен текущего.

Обзор технологии параллельного программирования MPI

План лекции

- Стандарт MPI
- Основные понятия
- Блокирующие двухточечные обмены
- Двухточечные обмены с буферизацией, другие типы двухточечных обменов

MPI

- MPI 1.1 Standard разрабатывался 92-94
- MPI 2.0 - 95-97
- MPI 2.1 - 2008 сентябрь 2008 г.
- MPI 3.0 - сентябрь 2012 г.

- Стандарты
  http://www.mcs.anl.gov/mpi
  http://www.mpi-forum.org/docs/docs.html
  https://computing.llnl.gov/tutorials/mpi/

- Описание функций
  http://www-unix.mcs.anl.gov/mpi/www/

Цель MPI

- Основная цель:
  - Обеспечение переносимости исходных кодов
  - Эффективная реализация

- Кроме того:
  - Большая функциональность
  - Поддержка неоднородных параллельных архитектур

Реализации MPI

- MPICH
- LAM/MPI
- Mvapich
- OpenMPI
- Коммерческие реализации Intel, IBM и др.

Модель MPI

- Параллельная программа состоит из процессов, процессы могут быть многопоточными.
- MPI реализует передачу сообщений между процессами.
- Межпроцессное взаимодействие предполагает:
  - синхронизацию
  - перемещение данных из адресного пространства одного процесса в адресное пространство другого процесса.

Основные понятия

- Процессы объединяются в группы.
- Каждое сообщение посылается в рамках некоторого контекста и должно быть получено в том же контексте.
- Группа и контекст вместе определяют коммуникатор.
- Процесс идентифицируется своим номером в группе, ассоциированной с коммуникатором.
- Коммуникатор, содержащий все начальные процессы, называется MPI_COMM_WORLD.

Понятие коммуникатора MPI

- Управляющий объект, представляющий группу процессов, которые могут взаимодействовать друг с другом
- Все обращения к MPI функциям содержат коммуникатор, как параметр
- Наиболее часто используемый коммуникатор MPI_COMM_WORLD
- Определяется при вызове MPI_Init
- Содержит ВСЕ процессы программы

Типы данных MPI

- Данные в сообщении описываются тройкой: (address, count, datatype), где datatype определяется рекурсивно как:
  - Предопределенный базовый тип, соответствующий типу данных в базовом языке (например, MPI_INT, MPI_DOUBLE_PRECISION)
  - Непрерывный массив MPI типов
  - Векторный тип
  - Индексированый тип
  - Произвольные структуры
- MPI включает функции для построения пользовательских типов данных, например, типа данных, описывающих пары (int, float).

Базовые типы данных

MPI datatype	C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double

Понятие тега

- Сообщение сопровождается определяемым пользователем признаком для идентификации принимаемого сообщения
- Теги сообщений у отправителя и получателя должны быть согласованы.
- Можно указать в качестве значения тэга константу MPI_ANY_TAG.
- Некоторые не-MPI системы передачи сообщений называют тэг типом сообщения.
- MPI вводит понятие тэга, чтобы не путать это понятие с типом данных MPI.

Формат MPI-функций

```
error = MPI_Xxxxx(parameter,...);
MPI_Xxxxx(parameter,...);
```

- Возвращаемое значение – код ошибки. Определяется константой MPI_SUCCESS

```
int error;
.....
error = MPI_Init(&argc, &argv));
if (error != MPI_SUCCESS)
{
        fprintf (stderr, " MPI_Init error \n");
        return 1;

}
```

Выполнение MPI-программы

- При запуске указываем число требуемых процессоров np и название программы
  mpirun -np 3 prog
- На выделенных для расчета узлах запускается np копий указанной программы
- Каждая копия программы получает два значения:
  - **np**
  - **rank** из диапазона [0 ... np-1]
- Любые две копии программы могут непосредственно обмениваться данными с помощью функций передачи сообщений

C: MPI helloworld.c

```
#include <mpi.h>
main(int argc, char **argv)
{
  int numtasks, rank;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &
  numtasks);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  printf("Hello World from process %d of %d\n“,
  rank, numtasks);
  MPI_Finalize();
}
```

Функции определения среды

- **int MPI_Init(int *argc, char ***argv)**
  должна первым вызовом, вызывается только один раз
- **int MPI_Comm_size(MPI_Comm comm, int *size)**
  число процессов в коммуникаторе
- **int MPI_Comm_rank(MPI_Comm comm, int *rank)**
  номер процесса в коммуникаторе (нумерация с 0)
- **int MPI_Finalize()**
  завершает работу процесса
- **int MPI_Abort (MPI_Comm_size(MPI_Comm comm, int*errorcode)**
  завершает работу программы

Информация о статусе сообщения

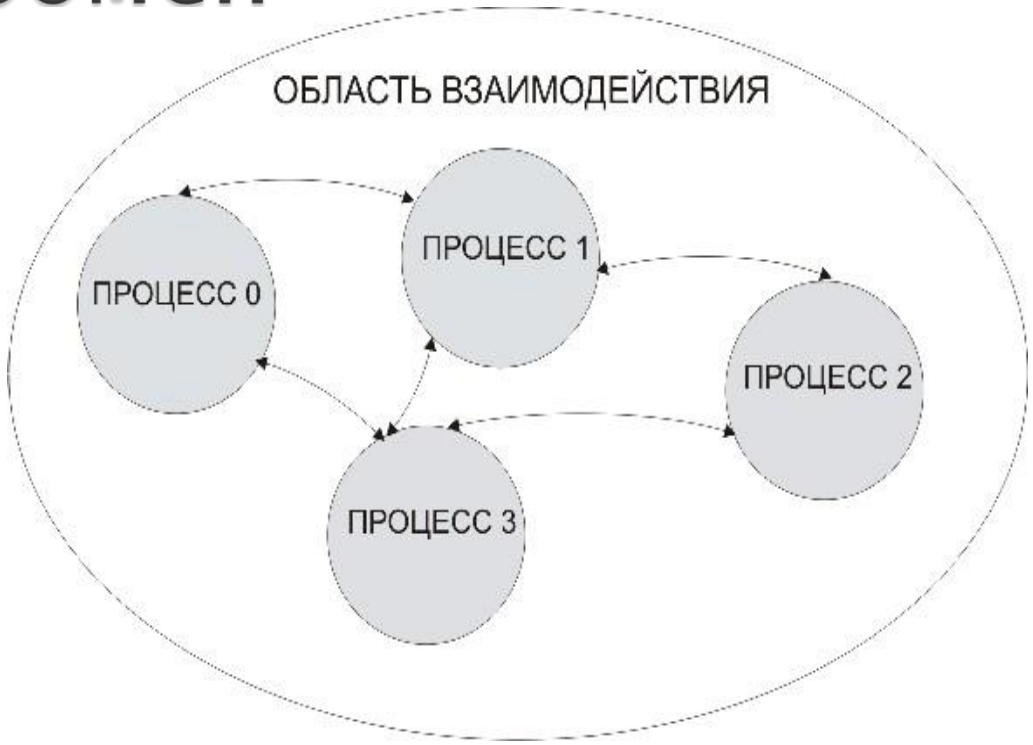
Содержит:

- Source: status.MPI_SOURCE
- Tag: status.MPI_TAG
- Count: MPI_Get_count

Двухточечный (point-to-point, p2p) обмен

- В двухточечном обмене участвуют только два процесса, процесс-отправитель и процесс-получатель (источник сообщения и адресат).
- Двухточечные обмены используются для организации локальных и неструктурированных коммуникаций.

Двухточечный (point-to-point, p2p) обмен



- Двухточечный обмен возможен только между процессами, принадлежащими одной области взаимодействия (одному коммуникатору).

Условия успешного взаимодействия точка-точка

- Отправитель должен указать правильный rank получателя
- Получатель должен указать верный rank отправителя
- Одинаковый коммуникатор
- Тэги должны соответствовать друг другу
- Буфер у процесса-получателя должен быть достаточного объема

Разновидности двухточечного обмена

- *блокирующие* прием/передача, которые приостанавливают выполнение процесса на время приема или передачи сообщения;
- *неблокирующие* прием/передача, при которых выполнение процесса продолжается в фоновом режиме, а программа в нужный момент может запросить подтверждение завершения приема сообщения;
- *синхронный* обмен, который сопровождается уведомлением об окончании приема сообщения;
- *асинхронный* обмен, который таким уведомлением не сопровождается.

Двухточечный (point-to-point, p2p) обмен

- Правильно организованный двухточечный обмен сообщениями должен исключать возможность блокировки или некорректной работы параллельной MPI-программы.

- Примеры ошибок в организации двухточечных обменов:
  - выполняется передача сообщения, но не выполняется его прием;
  - процесс-источник и процесс-получатель одновременно пытаются
    выполнить блокирующие передачу или прием сообщения.

Двухточечный (point-to-point, p2p) обмен

В MPI приняты следующие соглашения об именах подпрограмм двухточечного обмена:

`MPI_[I][R, S, B]Send`

здесь префикс [`I`] (Immediate) обозначает неблокирующий режим.

Один из префиксов [`R, S, B`] обозначает режим обмена: по готовности, синхронный и буферизованный.

Отсутствие префикса обозначает подпрограмму стандартного обмена.

Имеется 8 разновидностей операции передачи сообщений.

Для подпрограмм приема:

`MPI_[I]Recv`

то есть всего 2 разновидности приема.

Подпрограмма `MPI_Irsend`, например, выполняет передачу «по готовности» в неблокирующем режиме, `MPI_Bsend` буферизованную передачу с блокировкой, а `MPI_Recv` выполняет блокирующий прием сообщений.

Подпрограмма приема любого типа может принять сообщения от любой подпрограммы передачи.

Стандартная блокирующая передача

int MPI_Send(void *buf, int count, MPI_Datatype datatype,int dest, int tag, MPI_Comm comm)

MPI_Send(buf, count, datatype, dest, tag, comm,ierr)

- buf – адрес первого элемента в буфере передачи;
- count – количество элементов в буфере передачи (допускается count = 0);
- datatype – тип MPI каждого пересылаемого элемента;
- dest – ранг процесса-получателя сообщения (целое число от 0 до n – 1, где n число процессов в области взаимодействия);
- tag – тег сообщения;
- comm – коммуникатор;
- ierr – код завершения.

Стандартная блокирующая передача

- При стандартной блокирующей передаче после завершения вызова (после возврата из функции/процедуры передачи) можно использовать любые переменные, использовавшиеся в списке параметров. Такое использование не повлияет на корректность обмена.
- Дальнейшая «судьба» сообщения зависит от реализации MPI. Сообщение может быть сразу передано процессу-получателю или может быть скопировано в буфер передачи.
- Завершение вызова не гарантирует доставки сообщения по назначению. Такая гарантия предоставляется при использовании других разновидностей двухточечного обмена.

Стандартный блокирующий прием

int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)

MPI_Recv(buf, count, datatype, dest, tag, comm, status, ierr)

- buf – адрес первого элемента в буфере приёма;
- count – количество элементов в буфере приёма;
- datatype – тип MPI каждого пересылаемого элемента;
- source – ранг процесса-отправителя сообщения (целое число от 0 до n – 1, где n число процессов в области взаимодействия);
- tag – тег сообщения;
- comm – коммуникатор;
- status – статус обмена;
- ierr – код завершения.

Стандартный блокирующий прием

- Значение параметра `count` может оказаться больше, чем количество элементов в принятом сообщении. В этом случае после выполнения приёма в буфере изменится значение только тех элементов, которые соответствуют элементам фактически принятого сообщения.
- Для функции `MPI_Recv` гарантируется, что после завершения вызова сообщение принято и размещено в буфере приема.

Коды завершения

- MPI_ERR_COMM – неправильно указан коммуникатор. Часто возникает при использовании «пустого» коммуникатора;
- MPI_ERR_COUNT – неправильное значение аргумента count (количество пересылаемых значений);
- MPI_ERR_TYPE – неправильное значение аргумента, задающего тип данных;
- MPI_ERR_TAG – неправильно указан тег сообщения;
- MPI_ERR_RANK – неправильно указан ранг источника или адресата сообщения;
- MPI_ERR_ARG – неправильный аргумент, ошибочное задание которого не попадает ни в один класс ошибок;
- MPI_ERR_REQUEST – неправильный запрос на выполнение операции.

Джокеры

- В качестве ранга источника сообщения и в качестве тега сообщения можно использовать «джокеры» :
  - MPI_ANY_SOURCE – любой источник;
  - MPI_ANY_TAG – любой тег.

- При использовании «джокеров» есть опасность приема сообщения, не предназначенного данному процессу

Двухточечные обмены

Подпрограмма `MPI_Recv` может принимать сообщения, отправленные в любом режиме.

Прием может выполняться от произвольного процесса, а в операции передачи должен быть указан вполне определенный адрес.

Приемник может использовать «джокеры» для источника и для тега. Процесс может отправить сообщение и самому себе, но следует учитывать, что использование в этом случае блокирующих операций может привести к «тупику».

Двухточечные обмены

Размер полученного сообщения (count) можно определить с помощью вызова подпрограммы

int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count)

MPI_Get_count(status, datatype, count, ierr)

- count – количество элементов в буфере передачи;
- datatype – тип MPI каждого пересылаемого элемента;
- status – статус обмена;
- ierr – код завершения.

Аргумент datatype должен соответствовать типу данных, указанному в операции обмена

Двухточечный обмен с буферизацией

- Передача сообщения в буферизованном режиме может быть начата независимо от того, зарегистрирован ли соответствующий прием. Источник копирует сообщение в буфер, а затем передает его в неблокирующем режиме, так же как в стандартном режиме.
- Эта операция локальна, поскольку ее выполнение не зависит от наличия соответствующего приема.
- Если объем буфера недостаточен, возникает ошибка. Выделение буфера и его размер контролируются программистом.

Двухточечный обмен с буферизацией

- Размер буфера должен превосходить размер сообщения на величину `MPI_BSEND_OVERHEAD`. Это дополнительное пространство используется подпрограммой буферизованной передачи для своих целей.

- Если перед выполнением операции буферизованного обмена не выделен буфер, MPI ведет себя так, как если бы с процессом был связан буфер нулевого размера. Работа с таким буфером обычно завершается сбоем программы.

- Буферизованный обмен рекомендуется использовать в тех ситуациях, когда программисту требуется больший контроль над распределением памяти. Этот режим удобен и для отладки, поскольку причину переполнения буфера определить легче, чем причину тупика.

Двухточечный обмен с буферизацией

- При выполнении буферизованного обмена программист должен заранее создать буфер достаточного размера:

```
int MPI_Buffer_attach(void *buf, size)
```

```
MPI_Buffer_attach(buf, size, ierr)
```

- В результате вызова создается буфер `buf` размером `size` байтов. В программах на языке Fortran роль буфера может играть массив. За один раз к процессу может быть подключен только один буфер.

Двухточечный обмен с буферизацией

- Буферизованная передача завершается сразу, поскольку сообщение немедленно копируется в буфер для последующей передачи. В отличие от стандартного обмена, в этом случае работа источника и адресата не синхронизована:

```
int MPI_Bsend(void *buf, int count,
  MPI_Datatype datatype, int    dest, int tag,
  MPI_Comm comm)

MPI_Bsend(buf, count, datatype, dest, tag,
  comm, ierr)
```

Двухточечный обмен с буферизацией

- После завершения работы с буфером его необходимо отключить:

```
int MPI_Buffer_detach(void *buf, int *size)

MPI_Buffer_detach(buf, size, ierr)
```

- Возвращается адрес (`buf`) и размер отключаемого буфера (`size`). Эта операция блокирует работу процесса до тех пор, пока все сообщения, находящиеся в буфере, не будут обработаны. Вызов данной подпрограммы можно использовать для форсированной передачи сообщений. После завершения вызова можно вновь использовать память, которую занимал буфер. В языке С данный вызов не освобождает автоматически память, отведенную для буфера.

Двухточечный обмен с буферизацией

```
#include "mpi.h"
#include <stdio.h>
int main(int argc,char *argv[]) {
 int *buffer;
 int myrank;
 MPI_Status status;
 int buffsize = 1;
 int TAG = 0;
 MPI_Init(&argc, &argv);
 MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
 if (myrank == 0) {
    buffer = (int *) malloc(buffsize + MPI_BSEND_OVERHEAD);
    MPI_Buffer_attach(buffer, buffsize + MPI_BSEND_OVERHEAD);
    buffer = (int *) 10;
    MPI_Bsend(&buffer, buffsize, MPI_INT, 1, TAG, MPI_COMM_WORLD);
    MPI_Buffer_detach(&buffer, &buffsize);
 } else {
    MPI_Recv(&buffer, buffsize, MPI_INT, 0, TAG, MPI_COMM_WORLD, &status);
    printf("received: %i\n", buffer);
 }
  MPI_Finalize();
  return 0;
}
```

Синхронный режим

- Завершение передачи происходит только после того, как прием сообщения инициализирован другим процессом.
- Адресат посылает источнику «квитанцию» – уведомление о завершении приема. После получения этого уведомления обмен считается завершенным и источник "знает", что его сообщение получено:

```
int MPI_Ssend(void *buf, int count,
MPI_Datatype datatype, int dest, int tag,
MPI_Comm comm)
```

Режим «по готовности»

Передача «по готовности» выполняется с помощью подпрограммы

```
int MPI_Rsend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
```

```
MPI_Rsend(buf, count, datatype, dest, tag, comm, ierr)
```

Передача «по готовности» должна начинаться, если уже зарегистрирован соответствующий прием. При несоблюдении этого условия результат выполнения операции не определен.

Режим «по готовности»

Завершается она сразу же. Если прием не зарегистрирован, результат выполнения операции не определен.

Завершение передачи не зависит от того, вызвана ли другим процессом подпрограмма приема данного сообщения или нет, оно означает только, что буфер передачи можно использовать вновь.

Сообщение просто выбрасывается в коммуникационную сеть в надежде, что адресат его получит. Эта надежда может и не сбыться.

Режим «по готовности»

Обмен «по готовности» может увеличить производительность программы, поскольку здесь не используются этапы установки межпроцессных связей, а также буферизация.

Все это — операции, требующие времени. С другой стороны, обмен «по готовности» потенциально опасен, кроме того, он усложняет отладку, поэтому его рекомендуется использовать только в том случае, когда правильная работа программы гарантируется ее логической структурой, а выигрыша в быстродействии надо добиться любой ценой.

Совместные прием и передача

- Подпрограмма `MPI_Sendrecv` выполняет прием и передачу данных с блокировкой:

```
int MPI_Sendrecv(void *sendbuf, int sendcount,
MPI_Datatype sendtype, int dest, int sendtag, void
*recvbuf, int recvcount, MPI_Datatype recvtype,
int source, int recvtag, MPI_Comm comm, MPI_Status
*status)
```

- Подпрограмма `MPI_Sendrecv_replace` выполняет прием и передачу данных, используя общий буфер для передачи и приёма:

```
int MPI_Sendrecv_replace(void *buf, int count,
MPI_Datatype datatype, int dest, int sendtag, int
source,int recvtag, MPI_Comm comm, MPI_Status
*status)
```

Неблокирующие обмены

- Вызов подпрограммы неблокирующей передачи инициирует, но не завершает ее. Завершиться выполнение подпрограммы может еще до того, как сообщение будет скопировано в буфер передачи.

- Применение неблокирующих операций улучшает производительность программы, поскольку в этом случае допускается перекрытие (то есть одновременное выполнение) вычислений и обменов. Передача данных из буфера или их считывание может происходить одновременно с выполнением процессом другой работы.

Неблокирующие обмены

- Для завершения неблокирующего обмена требуется вызов дополнительной процедуры, которая проверяет, скопированы ли данные в буфер передачи.

- **ВНИМАНИЕ!**
  При неблокирующем обмене возвращение из подпрограммы обмена происходит сразу, но запись в буфер или считывание из него после этого производить **нельзя** – сообщение может быть еще не отправлено или не получено и работа с буфером может «испортить» его содержимое.

Неблокирующие обмены

Неблокирующий обмен выполняется в два этапа:

1. **инициализация** обмена;
2. **проверка завершения** обмена.

Разделение этих шагов делает необходимым *маркировку* каждой операции обмена, которая позволяет целенаправленно выполнять проверки завершения соответствующих операций.

Для маркировки в неблокирующих операциях используются *идентификаторы операций обмена*

Неблокирующие обмены

- Инициализация неблокирующей стандартной передачи выполняется подпрограммами `MPI_I[S, B, R]send`.
  Стандартная неблокирующая передача выполняется подпрограммой:

```
int MPI_Isend(void *buf, int count, MPI_Datatype
datatype, int dest, int tag, MPI_Comm comm,
MPI_Request *request)
```

```
MPI_Isend(buf, count, datatype, dest, tag, comm,
request, ierr)
```

- Входные параметры этой подпрограммы аналогичны аргументам подпрограммы `MPI_Send`.
- Выходной параметр `request` – идентификатор операции.

Неблокирующие обмены

- Инициализация неблокирующего приема выполняется при вызове подпрограммы:

```
int MPI_Irecv(void *buf, int count,
MPI_Datatype datatype, int source, int tag,
MPI_Comm comm, MPI_Request *request)
```

```
MPI_Irecv(buf, count, datatype, source, tag,
comm, request, ierr)
```

- Назначение аргументов здесь такое же, как и в ранее рассмотренных подпрограммах, за исключением того, что указывается ранг не адресата, а источника сообщения (`source`).

Неблокирующие обмены

- Вызовы подпрограмм неблокирующего обмена формируют *запрос* на выполнение операции обмена и связывают его с идентификатором операции `request`.

- Запрос идентифицирует свойства операции обмена:
  - режим;
  - характеристики буфера обмена;
  - контекст;
  - тег и ранг.

- Запрос содержит информацию о состоянии ожидающих обработки операций обмена и может быть использован для получения информации о состоянии обмена или для ожидания его завершения.

Проверка выполнения обмена

- Проверка фактического выполнения передачи или приема в неблокирующем режиме осуществляется с помощью вызова *подпрограмм ожидания*, блокирующих работу процесса до завершения операции или неблокирующих подпрограмм проверки, возвращающих логическое значение «истина», если операция выполнена

Проверка выполнения обмена

- В том случае, когда одновременно несколько процессов обмениваются сообщениями, можно использовать проверки, которые применяются одновременно к нескольким обменам.
- Есть три типа таких проверок:
  1. проверка завершения всех обменов;
  2. проверка завершения любого обмена из нескольких;
  3. проверка завершения заданного обмена из нескольких.
- Каждая из этих проверок имеет две разновидности:
  1. «ожидание»;
  2. «проверка».

Блокирующие операции проверки

- Подпрограмма `MPI_Wait` блокирует работу процесса до завершения приема или передачи сообщения:

```
int MPI_Wait(MPI_Request *request, MPI_Status *status)
MPI_Wait(request, status, ierr)
```

- Входной параметр `request` — идентификатор операции обмена, выходной — статус (`status`).

Блокирующие операции проверки

- Успешное выполнение подпрограммы `MPI_Wait` после вызова `MPI_Ibsend` подразумевает, что буфер передачи можно использовать вновь, то есть пересылаемые данные отправлены или скопированы в буфер, выделенный при вызове подпрограммы `MPI_Buffer_attach`.
- В этот момент уже нельзя отменить передачу. Если не будет зарегистрирован соответствующий прием, буфер нельзя будет освободить. В этом случае можно применить подпрограмму `MPI_Cancel`, которая освобождает память, выделенную подсистеме коммуникаций.

Проверка завершения всех обменов

- Проверка завершения всех обменов выполняется подпрограммой:

```
int MPI_Waitall(int count, MPI_Request
requests[], MPI_Status statuses[])

MPI_Waitall(count, requests, statuses, ierr)
```

- При вызове этой подпрограммы выполнение процесса блокируется до тех пор, пока **все** операции обмена, связанные с активными запросами в массиве `requests`, не будут выполнены. Возвращается статус этих операций. Статус обменов содержится в массиве `statuses`. `count` – количество запросов на обмен (размер массивов `requests` и `statuses`).

Проверка завершения всех обменов

- В результате выполнения подпрограммы `MPI_Waitall` запросы, сформированные неблокирующими операциями обмена, аннулируются, а соответствующим элементам массива присваивается значение `MPI_REQUEST_NULL`.

- В случае неуспешного выполнения одной или более операций обмена подпрограмма `MPI_Waitall` возвращает код ошибки `MPI_ERR_IN_STATUS` и присваивает полю ошибки статуса значение кода ошибки соответствующей операции.

- Если операция выполнена успешно, полю присваивается значение `MPI_SUCCESS`, а если не выполнена, но и не было ошибки – значение `MPI_ERR_PENDING`. Это соответствует наличию запросов на выполнение операции обмена, ожидающих обработки.

Проверка завершения любого числа обменов

- Проверка завершения любого числа обменов выполняется подпрограммой:

  `int MPI_Waitany(int count, MPI_Request requests[], int *index, MPI_Status *status)`

- Выполнение процесса блокируется до тех пор, пока, по крайней мере, один обмен из массива запросов (`requests`) не будет завершен.
- Входные параметры:
  - `requests` – запрос;
  - `count` – количество элементов в массиве `requests`.
- Выходные параметры:
  - `index` – индекс запроса (в языке С это целое число от 0 до `count` – 1) в массиве `requests`;
  - `status` – статус.

Неблокирующие процедуры проверки

- Подпрограмма `MPI_Test` выполняет неблокирующую проверку завершения приема или передачи сообщения:

```
int MPI_Test(MPI_Request *request, int *flag,
MPI_Status *status)
```

- Входной параметр: идентификатор операции обмена `request`.
- Выходные параметры:
  - `flag` — «истина», если операция, заданная идентификатором `request`, выполнена;
  - `status` — статус выполненной операции.

Неблокирующая проверка завершения всех обменов

- Подпрограмма `MPI_Testall` выполняет неблокирующую проверку завершения приема или передачи всех сообщений:

```
int MPI_Testall(int count, MPI_Request requests[],
int *flag, MPI_Status statuses[])
MPI_Testall(count, requests, flag, statuses, ierr)
```

- При вызове возвращается значение флага (`flag`) «истина», если все обмены, связанные с активными запросами в массиве `requests`, выполнены. Если завершены не все обмены, флагу присваивается значение «ложь», а массив `statuses` не определен.
- Параметр `count` – количество запросов.
- Каждому статусу, соответствующему активному запросу, присваивается значение статуса соответствующего обмена.

Неблокирующая проверка любого числа обменов

- Подпрограмма `MPI_Testany` выполняет неблокирующую проверку завершения приема или передачи сообщения:

```
int MPI_Testany(int count, MPI_Request requests[], int *index, int *flag, MPI_Status *status)
```

- Смысл и назначение параметров этой подпрограммы те же, что и для подпрограммы `MPI_Waitany`. Дополнительный аргумент `flag`, принимает значение «истина», если одна из операций завершена.
- Блокирующая подпрограмма `MPI_Waitany` и неблокирующая `MPI_Testany` взаимозаменяемы, как и другие аналогичные пары.

Другие операции проверки

- Подпрограммы `MPI_Waitsome` и `MPI_Testsome` действуют аналогично подпрограммам `MPI_Waitany` и `MPI_Testany`, кроме случая, когда завершается более одного обмена.
- В подпрограммах `MPI_Waitany` и `MPI_Testany` обмен из числа завершенных выбирается произвольно, именно для него и возвращается статус, а для `MPI_Waitsome` и `MPI_Testsome` статус возвращается для всех завершенных обменов.
- Эти подпрограммы можно использовать для определения, сколько обменов завершено.

Другие операции проверки

- Неблокирующая проверка выполнения обменов:

  int MPI_Waitsome(int incount, MPI_Request requests[], int *outcount, int indices[], MPI_Status statuses[])

- Здесь `incount` – количество запросов. В `outcount` возвращается количество выполненных запросов из массива `requests`, а в первых `outcount` элементах массива `indices` возвращаются индексы этих операций. В первых `outcount` элементах массива `statuses` возвращается статус завершенных операций. Если выполненный запрос был сформирован неблокирующей операцией обмена, он аннулируется. Если в списке нет активных запросов, выполнение подпрограммы завершается сразу, а параметру `outcount` присваивается значение `MPI_UNDEFINED`.

Другие операции проверки

- Неблокирующая проверка выполнения обменов:

```
int MPI_Testsome(int incount, MPI_Request
requests[], int *outcount, int indices[],
MPI_Status statuses[])
```

- Параметры такие же, как и у подпрограммы `MPI_Waitsome`. Эффективность подпрограммы `MPI_Testsome` выше, чем у `MPI_Testany`, поскольку первая возвращает информацию обо всех операциях, а для второй требуется новый вызов для каждой выполненной операции.

Неблокирующие сообщения (пример)

```
MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /*
find rank */
if (myrank == 0) {
    int x;
    MPI_Isend(&x,1,MPI_INT, 1, msgtag,
                        MPI_COMM_WORLD,&req1);
    compute();
    MPI_Wait(&req1, &status);
} else if (myrank == 1) {
    int x;
    MPI_Recv(&x,1,MPI_INT,0,msgtag,
                        MPI_COMM_WORLD, &status);
}
```

Проверка статуса операции приема сообщения

- Блокирующая проверка:

```
int MPI_Probe(int source, int tag, MPI_Comm comm,
  MPI_Status* status)
```

- Неблокирующая проверка:

```
int MPI_Iprobe(int source, int tag, MPI_Comm comm,
  int *flag, MPI_Status *status)
```

- Входные параметры этой подпрограммы те же, что и у подпрограммы `MPI_Probe`.
- Выходные параметры:
  - `flag` – флаг;
  - `status` – статус.

- Если сообщение уже поступило и может быть принято, возвращается значение флага «истина».

Проверка статуса операции приема сообщения (пример)

```
if (rank == 0) {
    MPI_Send(buf, size, MPI_INT, 1, 0, MPI_COMM_WORLD); // Send to process 1
    printf("0 sent %d numbers to 1\n", size);
} else if (rank == 1) {
   MPI_Status status;
   MPI_Probe(0, 0, MPI_COMM_WORLD, &status); // Probe for an incoming  msg.
   // When probe returns, the status object has the size and other
   // attributes of the incoming message. Get the size of the message.
   MPI_Get_count(&status, MPI_INT, &size)
   // Allocate a buffer just big enough to hold the incoming numbers
   int* number_buf = (int*)malloc(sizeof(int) * size);
   // Now receive the message with the allocated buffer
    MPI_Recv(number_buf, size, MPI_INT, 0, 0,
     MPI_COMM_WORLD,MPI_STATUS_IGNORE);
    printf("1 dynamically received %d numbers from 0.\n",number_amount);
     free(number_buf);
 }
```

Коллективные операции

- Передача сообщений между группой процессов
- Вызываются ВСЕМИ процессами в коммуникаторе
- Примеры:
  - Broadcast, scatter, gather (рассылка данных)
  - Global sum, global maximum, и.т.д. (редукционные операции)
  - Барьерная синхронизаци

Характеристики коллективных передач

- Коллективные операции не являются помехой операциям типа точка-точка и наоборот
- Все процессы коммуникатора должны вызывать коллективную операцию
- Синхронизация не гарантируется (за исключением барьера)
- Нет неблокирующих коллективных операций
- Нет тэгов
- Принимающий буфер должен точно соответствовать размеру отсылаемого буфера

Широковещательная рассылка

- One-to-all передача: один и тот же буфер отсылается от процесса root всем остальным процессам в коммуникаторе

```
int MPI_Bcast (void *buffer, int, count,
MPI_Datatype datatype,int root, MPI_Comm
comm)
```

- Все процессы должны указать один тот же root и communicator

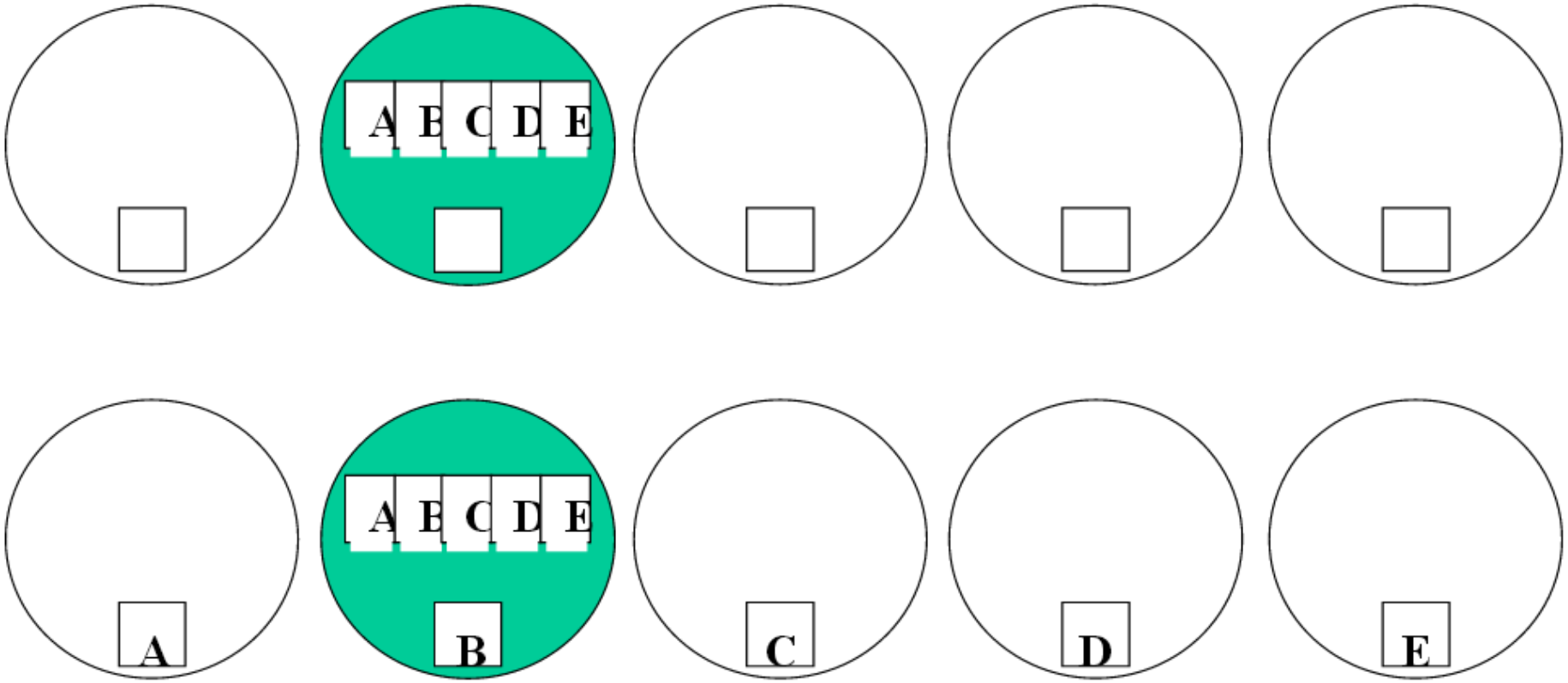
Scatter

- One-to-all communication: различные данные из одного процесса рассылаются всем процессам коммуникатора (в порядке их номеров)

```
int MPI_Scatter(void* sendbuf, int sendcount,
  MPI_Datatype sendtype, void* recvbuf,
  int recvcount,MPI_Datatype recvtype,
  int root, MPI_Comm comm)
```

- sendcount – число элементов, посланных каждому процессу, не общее число отосланных элементов;
- send параметры имеют смысл только для процесса root

Scatter

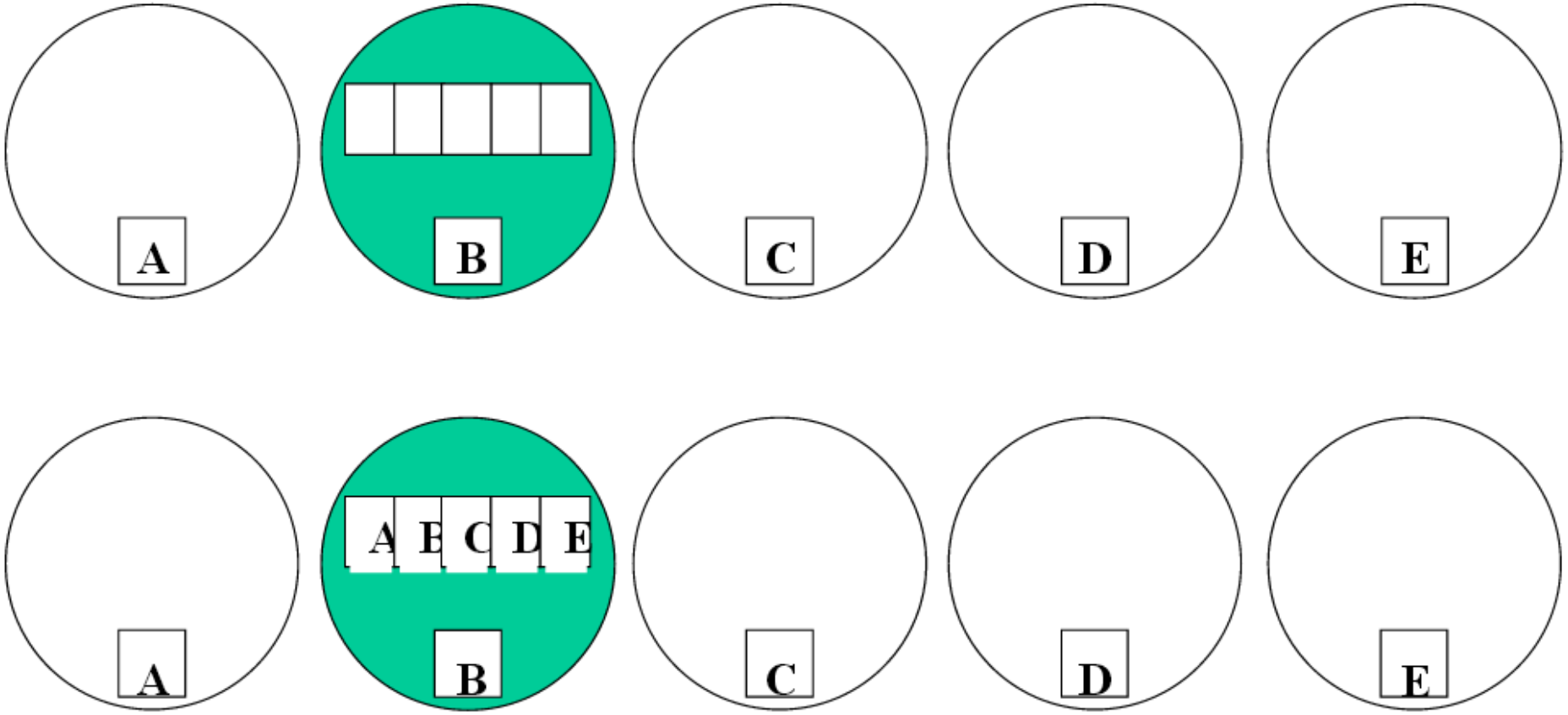


Gather

- All-to-one передачи: различные данные собираются процессом root
- Сбор данных выполняется в порядке номеров процессов.
- Длина блоков предполагается одинаковой, т.е. данные, посланные процессом i из своего буфера sendbuf, помещаются в i-ю порцию буфера recvbuf процесса root.
- Длина массива, в который собираются данные, должна быть достаточной для их размещения.

```
int MPI_Gather(void* sendbuf, int sendcount,
  MPI_Datatype sendtype, void* recvbuf, int
  recvcount,
  MPI_Datatype recvtype, int root, MPI_Comm comm)
```

Gather



Глобальные операции редукции

- Операции выполняются над данными, распределенными по процессам коммуникатора

- Примеры:
  - Глобальная сумма или произведение
  - Глобальный максимум (минимум)
  - Глобальная операция, определенная пользователем

Глобальные операции редукции

```
int MPI_Reduce(void* sendbuf, void* recvbuf, int
count, MPI_Datatype datatype, MPI_Op op, int root,
MPI_Comm comm)
```

- count число операций "op", выполняемых над последовательными элементами буфера
- sendbuf (также размер recvbuf)
- op является ассоциативной операцией, которая выполняется над парой операндов типа datatype и возвращает результат того же типа:
  MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD, MPI_LAND, MPI_BAND, MPI_LOR, MPI_BOR, MPI_LXOR, MPI_BXOR, MPI_MAXLOC, MPI_MINLOC

- MPI_ALLREDUCE – нет root процесса (все получают рез-т)

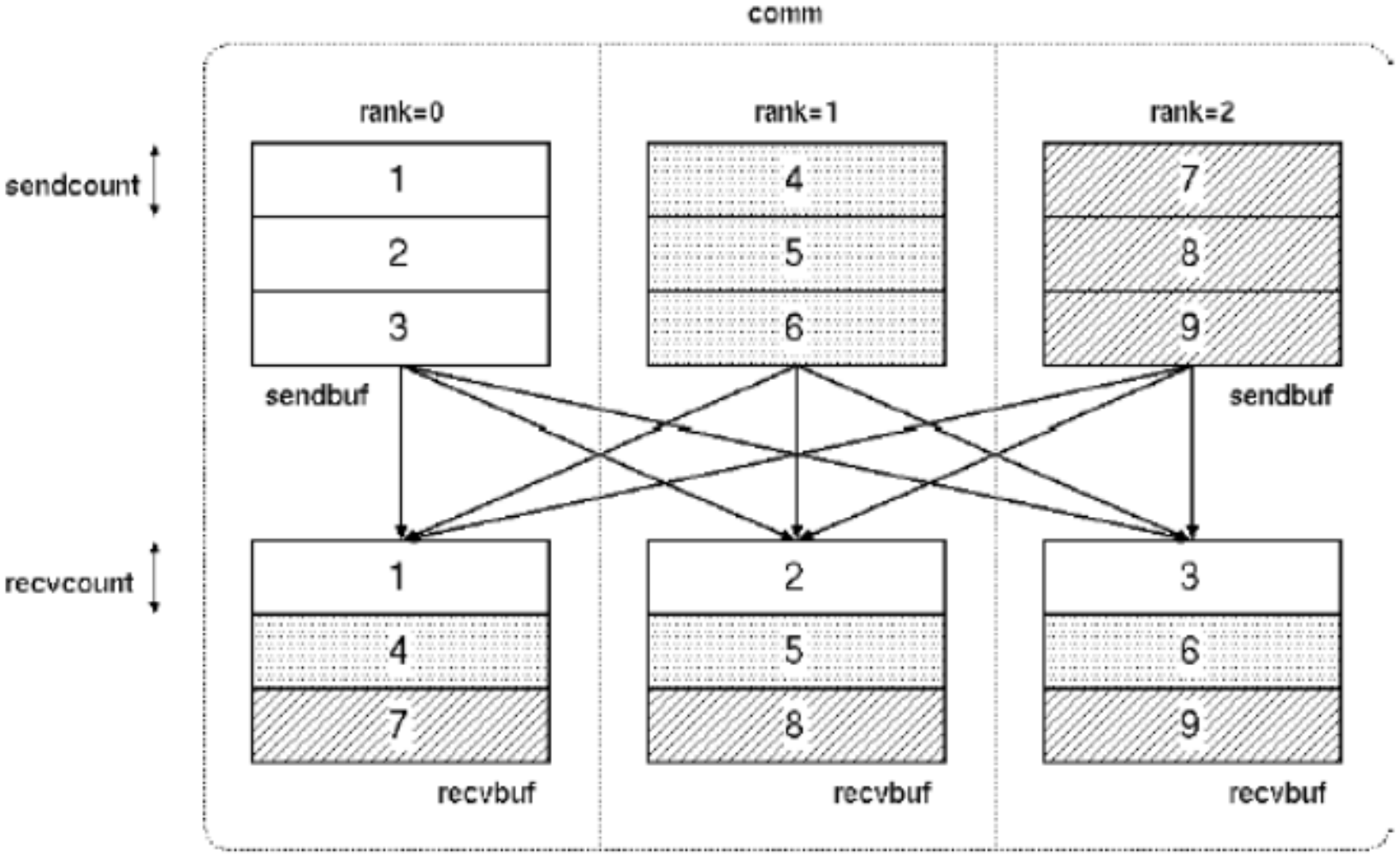
Рассылка MPI_Alltoall

```
int MPI_Alltoall (void* sendbuf,
  int sendcount, /* in */
  MPI_Datatype sendtype, /* in */
  void* recvbuf, /* out */
  int recvcount, /* in */
  MPI_Datatype recvtype, /* in */
  MPI_Comm comm);
```

Описание:

- Рассылка сообщений от каждого процесса каждому
- j-ый блок данных из процесса i принимается j-ым процессом и размещается в i-ом блоке буфера recvbuf

Рассылка MPI_Alltoall



Лекция 5

3 Коммуникации в распределенных системах

Все компьютеры в распределенной системе связаны между собой коммуникационной сетью. Коммуникационные сети подразделяются на широкомасштабные (Wide Area Networks, WANs) и локальные (Local Area Networks, LANs).

Широкомасштабные сети

WAN состоит из коммуникационных ЭВМ, связанных между собой коммуникационными линиями (телефонные линии, радиолинии, спутниковые каналы, оптоволокно) и обеспечивающих транспортировку сообщений.

Обычно используется техника store-and-forward, когда сообщения передаются из одного компьютера в следующий с промежуточной буферизацией.

Коммутация пакетов или коммутация линий.

Коммутация линий (телефонные разговоры) требует резервирования линий на время всего сеанса общения двух устройств.

Пакетная коммутация основана на разбиении сообщений в пункте отправления на порции (пакеты), посылке пакетов по адресу назначения, и сборке сообщения из пакетов в пункте назначения. При этом линии используются эффективнее, сообщения могут передаваться быстрее, но требуется работа по разбиению и сборке сообщений, а также возможны задержки (для передачи речи или музыки такой метод не годится).

Семиуровневая модель ISO

ISO OSI (International Standards Organizations»s Reference Model of Open Systems Interconnection) организует коммуникационные протоколы в виде семи уровней и специфицирует функции каждого уровня.

Локальные сети.

Особенности LAN:

- географическая область охвата невелика (здание или несколько зданий);
- высокая скорость передачи (100-1000 Mbps);
- малая вероятность ошибок передачи.

Свойственные многоуровневой модели ISO OSI накладные расходы являются причиной того, что в LAN применяются более простые протоколы.

Клиент-сервер

Можно избежать подтверждения получения сервером сообщения-запроса от клиента, если ответ сервера должен последовать скоро.

Удаленный вызов процедур

Send, receive - подход ввода/вывода

Более естественный подход, применяемый в централизованных ЭВМ - вызов процедур.

Birrell and Nelson (1984) (независимо и раньше - Илюшин А.И.,1978) предложили позволить вызывающей программе находиться на другой ЭВМ.

MPP с распределенной памятью может рассматриваться как частный случай локальной сети.

В середине 80-х годов английская фирма Inmos выпустила миникомпьютер, названный транспьютером. Его отличительной особенностью является наличие 8-ми каналов (4 входные и 4 выходные) для обмена информацией с другими транспьютерами или другими устройствами, такими как диски, терминалы и т.п.

Решетка транспьютеров, в которой каждый транспьютер параллельно с вычислениями может обмениваться одновременно по 8 каналам с 4 соседями, является хорошим примером, для которого будут формулироваться различные экзаменационные задачи.

Для передачи информации между двумя соседними (связанными одним каналом) узлами транспьютерной матрицы первый узел должен выдать операцию послать сообщение, а второй – операцию принять сообщение. При этом заданная в операции приема область памяти должна быть по размеру не меньше передаваемого сообщения. Операции посылки и приема сообщения завершаются только после полного завершения передачи сообщения.

Время передачи сообщения между двумя соседними узлами транспьютерной матрицы определяется двумя характеристиками аппаратуры - временем старта передачи Ts и временем передачи одного байта информации соседнему узлу Tb. При этом процессорные операции, включая чтение из памяти и запись в память, считаются бесконечно быстрыми. За время Ts+Tb транспьютер может передать 1 байт информации своим четырем соседям и принять от них 4 байта информации (по одному байту от каждого). Для этого необходимо запустить на нем 8 служебных процессов.

Конвейеризация (разбиение сообщения на кванты, чтобы воспользоваться параллельной работой каналов вдоль маршрута передачи) и параллельное использование нескольких маршрутов – два метода ускорения передачи сообщений.

3.1 Обмен сообщениями между прикладными процессами

SEND, RECEIVE (адресат/отправитель, [тэг,] адрес памяти, длина)

адресация - физический/логический номер процессора, уникальный идентификатор динамически создаваемого процесса, служба имен (сервер имен или широковещание - broadcasting).

Обычно пересылка в соседний компьютер требует три копирования - из памяти процесса-отправителя в буфер ОС на своем компьютере, пересылка между буферами ОС, копирование в память процесса-получателя.

Блокирующие операции send (до освобождения памяти с данными или до завершения фактической передачи) и неблокирующие.

Буферизуемые и небуферизуемые (rendezvous или с потерей информации при отсутствии receive).

Надежные и ненадежные.

MPI - Message-Passing Interface [4]

(Message Passing Interface Forum, May 5, 1994

http://www.mpi-forum.org)

(1) Цели**:**

- Создать интерфейс прикладного программирования (не только для компиляторов или библиотек реализации систем);
- Обеспечить возможность эффективных коммуникаций (избежать копирования из памяти в память, позволить совмещение вычислений и коммуникаций или разгрузку на коммуникационный процессор там, где он есть);
- Разрешить расширения для использования в гетерогенной среде;
- Исходить из надежности коммуникаций (пользователь не должен бороться с коммуникационными сбоями - это дело коммуникационных подсистем нижнего уровня);
- Определить интерфейс, который бы не слишком отличался от используемых в то время, таких как PVM, Express, P4, и пр.;
- Определить интерфейс, который мог бы быстро быть реализован на многих продаваемых платформах без серьезной переделки коммуникационного и системного ПО.

****

(2) Что включено в MPI ?

- Коммуникации точка-точка;
- Коллективные операции;
- Группы процессов;
- Коммуникационные контексты;
- Простой способ создания процессов для модели SPMD (одна программа используется для обработки разных данных на разных процессорах);
- Топология процессов.

****

(3) Что не включено в MPI ?

- Явные операции с разделяемой памятью и явная поддержка нитей (процессов с общей памятью);
- Операции, которые требуют больше поддержки от операционных систем, чем действующие в настоящее время стандарты на ОС (например, получение сообщений через механизм прерываний, активные сообщения);
- Вспомогательные функции, такие как таймеры.

****

(4) Некоторые понятия.

Коммуникационные операции могут быть:

неблокирущие - если возврат осуществляется до завершения операции;

блокируюшие - если возврат означает, что пользователь может использовать ресурсы (например, буфера), указанные в вызове;

Операция называется ***локальной***, если ее выполнение не требует коммуникаций; ***нелокальной***, если ее выполнение может требовать коммуникаций; ***коллективной***, если в ее выполнении должны участвовать все процессы группы.

(5) Группы, контексты, коммуникаторы.

Группа - упорядоченное (от 0 до ранга группы) множество идентификаторов процессов (т.е. процессов). Группы служат для указания адресата при посылке сообщений (процесс-адресат специфицируется своим номером в группе), определяют исполнителей коллективных операций.

Являются мощным средством функционального распараллеливания - позволяют разделить группу процессов на несколько подгрупп, каждая из которых должна выполнять свою параллельную процедуру. При этом существенно упрощается проблема адресации при использовании параллельных процедур.

Контекст - область «видимости» для сообщений, аналогичное области видимости переменных в случае вложенных вызовов процедур. Сообщения, посланные в некотором контексте, могут быть приняты только в этом же контексте. Контексты - также важные средства поддержки параллельных процедур.

Коммуникаторы - позволяют ограничить область видимости (жизни, определения) сообщений рамками некоторой группы процессов, т.е. могут рассматриваться как пара - группа и контекст. Кроме того, они служат и для целей оптимизации, храня необходимые для этого дополнительные объекты.

Имеются предопределенные коммуникаторы (точнее, создаваемые при инициализации MPI-системы):

- MPI_COMM_WORD - все процессы
- MPI_COMM_SELF - один текущий процесс

****

(6) Операции над группами (локальные, без обмена сообщениями).

Для поддержки пользовательских серверов имеется коллективная операция разбиения группы на подгруппы по ключам, которые указывает каждый процесс группы.

Для поддержки связывания с серверами, имеются средства построения коммуникатора по некоторому имени, известному и серверу и клиентам.

(7) Точечные коммуникации.

Основные операции - send, receive

Операции могут быть ***блокирующими*** и ***неблокирующими***.

В операции send задается:

- адрес буфера в памяти;
- количество посылаемых элементов;
- тип данных каждого элемента;
- номер процесса-адресата в его группе;
- тег сообщения;
- коммуникатор.

(последние 3 параметра - аналоги «почтового конверта»)

В операции receive задается:

- адрес буфера в памяти;
- количество принимаемых элементов;
- тип данных каждого элемента;
- номер процесса-отправителя в его группе (либо «любой»);
- тег сообщения (либо «любой»);
- коммуникатор;
- статус (источник и тег, необходимые в том случае, когда они неизвестны - при их задании с помощью шаблона «любой»).

Предусмотрена конвертация данных при работе в гетерогенной среде.

Имеется четыре режима коммуникаций - стандартный, буферизуемый, синхронный и режим готовности.

В стандартном режиме последовательность выдачи операций send и receive произвольна, операция send завершается тогда, когда сообщение изъято из памяти процесса (например, переписано в системный буфер) и эта память уже может использоваться процессом. При этом выполнение операции может осуществляться независимо от наличия receive, либо требовать наличие (вопрос реализации MPI). Поэтому операция считается нелокальной.

В буферизуемом режиме последовательность выдачи операций send и receive произвольна, операция send завершается тогда, когда сообщение изъято из памяти и помещено в буфер, предварительно заведенный в памяти процесса. Если места в буфере нет - ошибка программы (но есть возможность определить свой буфер). Операция локальная.

В синхронном режиме последовательность выдачи операций произвольна, но операция send завершается только после выдачи и начала выполнения операции receive. Для этого посылающая сторона запрашивает у принимающей стороны подтверждение выдачи операции receive. Операция нелокальная.

В режиме готовности операция send может быть выдана только после выдачи соответствующей операции receive, иначе программа считается ошибочной и результат ее работы не определен. Операция нелокальная.

Во всех четырех режимах операция receive завершается после получения сообщения в заданный пользователем буфер приема.

Неблокирующие операции не приостанавливают процесс до своего завершения, а возвращают ссылку на коммуникационный объект, позволяющий опрашивать состояние операции или дожидаться ее окончания.

Имеются операции проверки поступающих процессу сообщений, без чтения их в буфер (например, для определения длины сообщения и запроса затем памяти под него).

Имеется возможность аварийно завершать выданные неблокирующие операции, и поэтому предоставлены возможности проверки, хорошо ли завершились операции.

Имеется составная операция send-receive, позволяющая избежать трудностей с порядком выдачи отдельных операций в обменивающихся между собой процессах. Для частного случая обмена данными одного типа и длины предлагается специальная операция (send-receive-replace), в которой для посылки и приема сообщения используется один буфер.

(8) Коллективные коммуникации.

Для обеспечения коллективных коммуникаций введены следующие функции:

- барьер для всех членов группы (BARRIER);
- передача сообщения всем членам группы от одного (BROADCAST);
- сбор данных от всех членов группы для одного (GATHER);
- рассылка данных всем членам группы от одного (SCATTER);
- сбор данных от всех членов группы для всех (ALLGATHER);
- рассылка данных всем членам группы от всех (ALLTOALL);
- глобальные операции (сумма, максимум, и т.п.), когда результат сообщается всем членам группы или только одному. При этом пользователь может сам определить глобальную операцию - функцию;

Схема перемещения данных между 4 процессами

Данные

Pr-0	A0			
Pr-1				
Pr-2				
Pr-3				

BROADCAST →

A0			
A0			
A0			
A0			

Pr-0	A0	A1	A2	A3
Pr-1				
Pr-2				
Pr-3				

SCATTER →

GATHER ←

A0			
A1			
A2			
A3			

Pr-0	A0	A1	A2	A3
Pr-1	A0	A1	A2	A3
Pr-2	A0	A1	A2	A3
Pr-3	A0	A1	A2	A3

ALLGATHER ←

A0			
A1			
A2			
A3			

Pr-0	A0	A1	A2	A3
Pr-1	B0	B1	B2	B3
Pr-2	C0	C1	C2	C3

ALLTOALL →

A0	B0	C0	D0
A1	B1	C1	D1
A2	B2	C2	D2

Pr-3	D0	D1	D2	D3

A3	B3	C3	D3

Названия функций и параметры:

MPI_BARRIER(IN comm)

MPI_BCAST(IN/OUT buffer, IN cnt, IN type, IN root, IN comm)

MPI_GATHER(IN sendbuf, IN sendcnt, IN sendtype, OUT recvbuf,
IN recvcnt, IN recvtype, IN root, IN comm)

MPI_SCATTER(IN sendbuf, IN sendcnt, IN sendtype, OUT recvbuf,
IN recvcnt, IN recvtype, IN root, IN comm)

MPI_ALLGATHER(IN sendbuf, IN sendcnt, IN sendtype,
OUT recvbuf, IN recvcnt, IN recvtype, IN comm)

MPI_ALLTOALL(IN sendbuf, IN sendcnt, IN sendtype, OUT recvbuf,
IN recvcnt, IN recvtype, IN comm)

У последних 4-х функций есть векторные варианты, предназначенные для работы с неравномерным распределением данных по процессорам.

PVM (Parallel Virtual Machine).

Широко известная система PVM [5] была создана для объединения нескольких связанных сетью рабочих станций в единую виртуальную параллельную ЭВМ. Система представляет собой надстройку над операционной системой UNIX и используется в настоящее время на различных аппаратных платформах, включая и ЭВМ с массовым параллелизмом.

Задача пользователя представляет собой множество подзадач, которые динамически создаются на указанных процессорах распределенной системы и взаимодействуют между собой путем передачи и приема сообщений (а также посредством механизма сигналов).

Достоинства - простота, наличие наследованного от OS UNIX аппарата процессов и сигналов, а также возможность динамического добавления к группе вновь созданных процессов.

Недостатки - низкая производительность и функциональная ограниченность (например, имеется только один режим передачи сообщений - с буферизацией).

MPI-2 (1997 г.) http://www.mpi-forum.org

Развивает MPI в следующих направлениях:

- Динамическое создание и уничтожение процессов (важно для работы в сетях ЭВМ).
- Односторонние коммуникации и средства синхронизации для организации взаимодействия процессов через общую память (для эффективной работы на

системах с непосредственным доступом процессоров к памяти других процессоров).

- Параллельные операции ввода-вывода (для эффективного использования существующих возможностей параллельного доступа многих процессоров к различным дисковым устройствам).

4 Синхронизация в распределенных системах

Обычно децентрализованные алгоритмы имеют следующие свойства:

(1) Относящаяся к делу информация распределена среди множества ЭВМ.
(2) Процессы принимают решение на основе только локальной информации.

(3) Не должно быть единственной критической точки, выход из строя которой приводил бы к краху алгоритма.
(4) Не существует общих часов или другого источника точного глобального времени.

Первые три пункта все говорят о недопустимости сбора всей информации для принятия решения в одно место. Обеспечение синхронизации без централизации требует подходов, отличных от используемых в традиционных ОС.
Последний пункт также очень важен - в распределенных системах достигнуть согласия относительно времени совсем непросто. Важность наличия единого времени можно оценить на примере программы make в ОС UNIX.

4.1 Синхронизация времени.

Аппаратные часы (скорее таймер - счетчик временных сигналов и регистр с начальным значением счетчика) основаны на кварцевом генераторе и могут в разных ЭВМ различаться по частоте.

В 1978 году Lamport показал, что синхронизация времени возможна, и предложил алгоритм для такой синхронизации. При этом он указал, что абсолютной синхронизации не требуется. Если два процесса не взаимодействуют, то единого времени им не требуется. Кроме того, в большинстве случаев согласованное время может не иметь ничего общего с астрономическим временем, которое объявляется по радио. В таких случаях можно говорить о логических часах.

Для синхронизации логических часов Lamport определил отношение «произошло до». Выражение a-->b читается как «a произошло до b» и означает, что все процессы согласны, что сначала произошло событие «a», а затем «b». Это отношение может в двух случаях быть очевидным:

(1)Если оба события произошли в одном процессе.
(2)Если событие «a» есть операция SEND в одном процессе, а событие «b» - прием этого сообщения другим процессом.
Отношение --> является транзитивным.

Если два события «x» и «y» случились в различных процессах, которые не обмениваются сообщениями, то отношения x-->y и y-->x являются неверными, а эти события называют одновременными.

Введем логическое время C таким образом, что если a-->b, то C(a) < C(b)

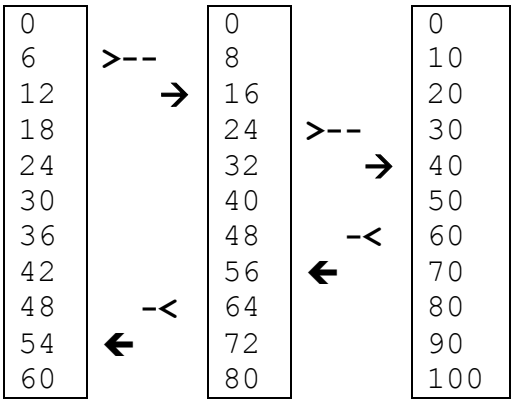
Алгоритм:

(1)Часы Ci увеличивают свое значение с каждым событием в процессе Pi:
Ci = Ci + d (d > 0, обычно равно 1)

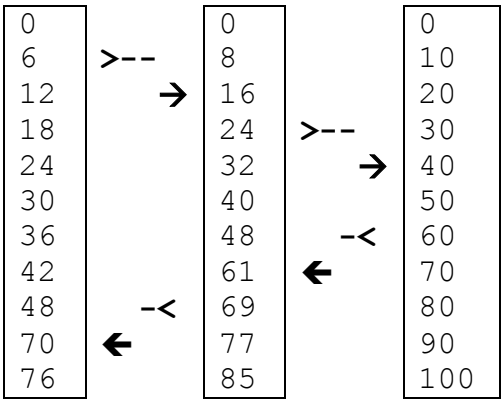
(2)Если событие «a» есть посылка сообщения «m» процессом Pi, тогда в это сообщение вписывается временная метка tm=Ci(a). В момент получения этого сообщения процессом Pj его время корректируется следующим образом:
Cj = max(Cj,tm+d)

Поясним на примере, как осуществляется эта коррекция.

Логическое время без коррекции.



Логическое время с коррекцией.



Для целей упорядочения всех событий удобно потребовать, чтобы их времена никогда не совпадали. Это можно сделать, добавляя в качестве дробной части к времени уникальный номер процесса (40.1, 40.2).

Однако логических часов недостаточно для многих применений (системы управления в реальном времени).

Физические часы.

После изобретения в 17 веке механических часов время измерялось астрономически. Интервал между двумя последовательными достижениями солнцем наивысшей точки на небе называется солнечным днем. Солнечная секунда равняется 1/86400(24*3600) части солнечного дня. В 1940-х годах было установлено, что период вращения земли не постоянен - земля замедляет вращение из-за приливов и атмосферы. Геологи считают, что 300 миллионов лет назад в году было 400 дней. Происходят и изменения длительности дня по другим причинам. Поэтому стали вычислять за длительный период среднюю солнечную секунду.

С изобретением в 1948 году атомных часов появилась возможность точно измерять время независимо от колебаний солнечного дня. В настоящее

время 50 лабораторий в разных точках земли имеют часы, базирующиеся на частоте излучения Цезия-133. Среднее значение является международным атомным временем (TAI), исчисляемым с 1 июля 1958 года.

Отставание TAI от солнечного времени компенсируется добавлением секунды тогда, когда разница становится больше 800 мксек. Это скорректированное время, называеемое UTC (Universal Coordinated Time), заменило прежний стандарт (Среднее время по Гринвичу - астрономическое время). При объявлении о добавлении секунды к UTC электрические компании меняют частоту с 60 Hz на 61 Hz (с 50 на 51) на период времени в 60 (50) секунд. Для обеспечения точного времени сигналы WWV передаются коротковолновым передатчиком (Fort Collins, Colorado) в начале каждой секунды UTC. Есть и другие службы времени.

Алгоритмы синхронизации времени.

Две проблемы - часы не должны ходить назад (надо ускорять или замедлять их для проведения коррекции) и ненулевое время прохождения сообщения о времени (можно многократно замерять время прохождения сообщений с показаниями часов туда и обратно, и брать самую удачную попытку – с минимальным временем прохождения).

4.2 Выбор координатора.

Многие распределенные алгоритмы требуют, чтобы один из процессов выполнял функции координатора, инициатора или некоторую другую специальную роль. Выбор такого специального процесса будем называть выбором координатора. При этом очень часто бывает не важно, какой именно процесс будет выбран. Можно считать, что обычно выбирается процесс с самым большим уникальным номером. Могут применяться разные алгоритмы, имеющие одну цель - если процедура выборов началась, то она должна закончиться согласием всех процессов относительно нового координатора.

Алгоритм «задиры».

Если процесс обнаружит, что координатор очень долго не отвечает, то инициирует выборы. Процесс P проводит выборы следующим образом:

1) P посылает сообщение «ВЫБОРЫ» всем процессам с большими чем у него номерами.
2) Если нет ни одного ответа, то P считается победителем и становится координатором.
3) Если один из процессов с большим номером ответит, то он берет на себя проведение выборов. Участие процесса P в выборах заканчивается.

В любой момент процесс может получить сообщение «ВЫБОРЫ» от одного из коллег с меньшим номером. В этом случае он посылает ответ «OK», чтобы сообщить, что он жив и берет проведение выборов на себя, а затем начинает выборы (если к этому моменту он уже их не вел). Следовательно, все процессы прекратят выборы, кроме одного - нового координатора. Он

извещает всех о своей победе и вступлении в должность сообщением «КООРДИНАТОР».

Если процесс выключился из работы, а затем захотел восстановить свое участие, то он проводит выборы (отсюда и название алгоритма).

Круговой алгоритм.

Алгоритм основан на использовании кольца (физического или логического). Каждый процесс знает следующего за ним в круговом списке. Когда процесс обнаруживает отсутствие координатора, он посылает следующему за ним процессу сообщение «ВЫБОРЫ» со своим номером. Если следующий процесс не отвечает, то сообщение посылается процессу, следующему за ним, и т.д., пока не найдется работающий процесс. Каждый работающий процесс добавляет в список работающих свой номер и переправляет сообщение дальше по кругу. Когда процесс обнаружит в списке свой собственный номер (круг пройден), он меняет тип сообщения на «КООРДИНАТОР» и оно проходит по кругу, извещая всех о списке работающих и координаторе (процессе с наибольшим номером в списке). После прохождения круга сообщение удаляется.

4.3 Взаимное исключение.

Ниже приводятся 5 различных алгоритмов.

***Централизованный алгоритм*.**

Все процессы запрашивают у координатора разрешение на вход в критическую секцию и ждут этого разрешения. Координатор обслуживает запросы в порядке поступления. Получив разрешение процесс входит в критическую секцию. При выходе из нее он сообщает об этом координатору. Количество сообщений на одно прохождение критической секции - 3.

Недостатки алгоритма - обычные недостатки централизованного алгоритма (крах координатора или его перегрузка сообщениями).

Децентрализованный алгоритм на основе временных меток.

Алгоритм носит имя Ricart-Agrawala и является улучшением алгоритма, который предложил Lamport.

Требуется глобальное упорядочение всех событий в системе по времени.

Вход в критическую секцию

Когда процесс желает войти в критическую секцию, он посылает всем процессам сообщение-запрос, содержащее имя критической секции, номер процесса и текущее время.

После посылки запроса процесс ждет, пока все дадут ему разрешение. После получения от всех разрешения, он входит в критическую секцию.

Поведение процесса при приеме запроса

Когда процесс получает сообщение-запрос, в зависимости от своего состояния по отношению к указанной критической секции он действует одним из следующих способов.

1) Если получатель не находится внутри критической секции и не запрашивал разрешение на вход в нее, то он посылает отправителю сообщение «OK».
2) Если получатель находится внутри критической секции, то он не отвечает, а запоминает запрос.
3) Если получатель выдал запрос на вхождение в эту секцию, но еще не вошел в нее, то он сравнивает временные метки своего запроса и чужого. Побеждает тот, чья метка меньше. Если чужой запрос победил, то процесс посылает сообщение «OK». Если у чужого запроса метка больше, то ответ не посылается, а чужой запрос запоминается.

Выход из критической секции

После выхода из секции он посылает сообщение «OK» всем процессам, запросы от которых он запомнил, а затем стирает все запомненные запросы.

Количество сообщений на одно прохождение секции - 2(n-1), где n - число процессов.

Кроме того, одна критическая точка заменилась на n точек (если какой-то процесс перестанет функционировать, то отсутствие разрешения от него всех остановит).

И, наконец, если в централизованном алгоритме есть опасность перегрузки координатора, то в этом алгоритме перегрузка любого процесса приведет к тем же последствиям.

Некоторые улучшения алгоритма (например, ждать разрешения не от всех, а от большинства) требуют наличия неделимых широковещательных рассылок сообщений.

Следующие три алгоритма – маркерные. Их основное отличие от двух первых заключается в том, что процессы получают разрешение на вход в критическую секцию только при наличии маркера. В каждом алгоритме используется свой метод получения маркера.

Алгоритм с круговым маркером.

Все процессы составляют логическое кольцо, когда каждый знает, кто следует за ним. По кольцу циркулирует маркер, дающий право на вход в критическую секцию. Получив маркер (посредством сообщения точка-точка) процесс либо входит в критическую секцию (если он ждал разрешения) либо переправляет маркер дальше. После выхода из критической секции маркер переправляется дальше, повторный вход в секцию при том же маркере не разрешается.

Алгоритм широковещательный маркерный (Suzuki-Kasami).

Маркер содержит:

- очередь запросов;

- массив LN[1...N] с номерами последних удовлетворенных запросов.

Вход в критическую секцию

1) Если процесс Pk, запрашивающий критическую секцию, не имеет маркера, то он увеличивает порядковый номер своих запросов RNk[k] и посылает широковещательно сообщение «ЗАПРОС», содержащее номер процесса (k) и номер запроса (Sn = RNk[k]).
2) Процесс Pk выполняет критическую секцию, если имеет (или когда получит) маркер.

Поведение процесса при приеме запроса

1) Когда процесс Pj получит сообщение-запрос от процесса Pk, он устанавливает RNj[k]=max(RNj[k],Sn). Если Pj имеет свободный маркер, то он его посылает Pk только в том случае, когда RNj[k]==LN[k]+1 (запрос не старый).

Выход из критической секции процесса Pk.

1) Устанавливает LN[k] в маркере равным RNk[k].
2) Для каждого Pj, для которого RNk[j]=LN[j]+1, он добавляет его идентификатор в маркерную очередь запросов (если там его еще нет).
3) Если маркерная очередь запросов не пуста, то из нее удаляется первый элемент, а маркер посылается соответствующему процессу (запрос которого был первым в очереди).

Алгоритм древовидный маркерный (Raymond).

Все процессы представлены в виде сбалансированного двоичного дерева. Каждый процесс имеет очередь запросов от себя и соседних процессов (1-го, 2-х или 3-х) и указатель в направлении владельца маркера.

Вход в критическую секцию

Если есть маркер, то процесс выполняет КС.
Если нет маркера, то процесс:
1) помещает свой запрос в очередь запросов
2) посылает сообщение «ЗАПРОС» в направлении владельца маркера и ждет сообщений.

Поведение процесса при приеме сообщений

Процесс, не находящийся внутри КС должен реагировать на сообщения двух видов -«МАРКЕР» и «ЗАПРОС».

А) Пришло сообщение «МАРКЕР»

М1. Взять 1-ый запрос из очереди и послать маркер его автору (концептуально, возможно себе);
М2. Поменять значение указателя в сторону маркера;
М3. Исключить запрос из очереди;
М4. Если в очереди остались запросы, то послать сообщение «ЗАПРОС» в сторону маркера.

Б) Пришло сообщение «ЗАПРОС».

1. Поместить запрос в очередь
2. Если нет маркера, то послать сообщение «ЗАПРОС» в сторону маркера, иначе (если есть маркер) - перейти на пункт М1.

Выход из критической секции

Если очередь запросов пуста, то при выходе ничего не делается, иначе - перейти к пункту М1.

Измерение производительности.

Введем следующие три метрики.

1) MS/CS - количество операций приема сообщений, требуемое для одного прохождения критической секции.
2) TR - время ответа, время от появления запроса до получения разрешения на вход.
3) SD - синхронизационная задержка, время от выхода из критической секции одного процесса до входа в нее следующего процесса (другого!).

При оценке производительности интересны две ситуации:

- низкая загрузка (LL), при которой вероятность запроса входа в занятую критическую секцию очень мала;
- высокая загрузка (HL), при которой всегда есть запросы на вход в занятую секцию.

Для некоторых метрик интересно оценить наилучшее и наихудшее значение (которые часто достигаются при низкой или высокой загрузке).

Сравнение алгоритмов.

При оценке времен исходим из коммуникационной среды, в которой время одного сообщения (Т) равно времени широковещательного сообщения.

Название алгоритма	***TR***	***SD***	***MS/CS (LL)***	***MS/CS (HL)***
Централизованный	2T	2T	3	3
Круговой маркерный				
Древовидный маркерный				
Децентрализованный с временными метками	**NT**	T	2(N-1)	2(N-1)
Широковещательный маркерный				

Все алгоритмы не устойчивы к крахам процессов (децентрализованные даже более чувствительны к ним, чем централизованный). Они в таком виде не годятся для отказоустойчивых систем.

4.4 Координация процессов

1) Сообщения точка-точка (если известно, кто потребитель).
2) Если неизвестно, кто потребитель, то:

- сообщения широковещательные;
- сообщения в ответ на запрос.

3) Если неизвестно, кто потребляет и кто производит, то:

- сообщения и запросы через координатора:
- широковещательный запрос.

Лекция 7

5 Распределенные файловые системы

Две главные цели.

Сетевая прозрачность.

Самая важная цель - обеспечить те же самые возможности доступа к файлам, распределенным по сети ЭВМ, которые обеспечиваются в системах разделения времени на централизованных ЭВМ.

Высокая доступность.

Другая важная цель - обеспечение высокой доступности. Ошибки систем или осуществление операций копирования и сопровождения не должны приводить к недоступности файлов.

Понятие файлового сервиса и файлового сервера.

Файловый сервис - это то, что файловая система предоставляет своим клиентам, т.е. интерфейс с файловой системой.

Файловый сервер - это процесс, который реализует файловый сервис.

Пользователь не должен знать, сколько файловых серверов имеется и где они расположены.

Так, как файловый сервер обычно является обычным пользовательским процессом, то в системе могут быть различные файловые серверы, предоставляющие различный сервис (например, UNIX файл сервис и MS-DOS файл сервис).

5.1 Архитектура распределенных файловых систем

Распределенная файловая система обычно имеет два существенно отличающихся компонента - непосредственно файловый сервер и сервер директорий.

5.1.1 Интерфейс файлового сервера

Для любой файловой системы первый фундаментальный вопрос - *что такое файл*. Во многих системах, таких как UNIX и MS-DOS, файл - не интерпретируемая последовательность байтов. На многих централизованных ЭВМ (IBM/370) файл представляется как последовательность записей, которую можно специфицировать ее номером или содержимым некоторого поля (ключом). Так, как большинство распределенных систем базируются на использовании среды UNIX и MS-DOS, то они используют первый вариант понятия файла.

Файл может иметь *атрибуты* (информация о файле, не являющаяся его частью). Типичные атрибуты - владелец, размер, дата создания и права доступа.

Важный аспект файловой модели - могут ли файлы *модифицироваться* после создания. Обычно могут, но есть системы с неизменяемыми файлами. Такие файлы освобождают разработчиков от многих проблем при кэшировании и размножении.

Защита обеспечивается теми же механизмами, что и в однопроцессорных ЭВМ - мандатами и списками прав доступа. Мандат - своего рода билет, выданный пользователю для каждого файла с указанием прав доступа. Список прав доступа задает для каждого файла список пользователей с их правами. Простейшая схема с правами доступа - UNIX схема, в которой различают три типа доступа (чтение, запись, выполнение), и три типа пользователей (владелец, члены его группы, и прочие).

Файловый сервис может базироваться на одной из двух моделей - модели *загрузки/разгрузки* и модели *удаленного доступа*. В первом случае файл передается между клиентом (памятью или дисками) и сервером целиком, а во втором файл сервис обеспечивает множество операций (открытие, закрытие, чтение и запись части файла, сдвиг указателя, проверку и изменение атрибутов, и т.п.). Первый подход требует большого объема памяти у клиента, затрат на перемещение ненужных частей файла. При втором подходе файловая система функционирует на сервере, клиент может не иметь дисков и большого объема памяти.

5.1.2 Интерфейс сервера директорий.

Обеспечивает операции создания и удаления директорий, именования и переименования файлов, перемещение файлов из одной директории в другую.

Определяет алфавит и синтаксис имен. Для спецификации ***типа*** информации в файле используется часть имени (расширение) либо явный атрибут.

Все распределенные системы позволяют директориям содержать поддиректории - такая файловая система называется ***иерархической***. Некоторые системы позволяют создавать указатели или ссылки на произвольные директории, которые можно помещать в директорию. При этом можно строить не только деревья, но и произвольные графы. Разница между ними очень важна для распределенных систем, поскольку в случае графа удаление связи может привести к появлению недостижимых поддеревьев, обнаруживать которые в распределенных системах очень трудно.

Ключевое решение при конструировании распределенной файловой системы - должны или не должны машины (или процессы) одинаково видеть иерархию директорий. Тесно связано с этим решением наличие единой корневой директории (можно иметь такую директорию с поддиректориями для каждого сервера).

Прозрачность именования.

Две формы прозрачности именования различают - прозрачность расположения (/server/d1/f1) и прозрачность миграции (когда изменение расположения файла не требует изменения имени).

Имеются три подхода к именованию:

- машина + путь;
- монтирование удаленных файловых систем в локальную иерархию файлов;
- единственное пространство имен, которое выглядит одинаково на всех машинах.

Последний подход необходим для достижения того, чтобы распределенная система выглядела как единый компьютер.

Двухуровневое именование.

Большинство систем используют ту или иную форму двухуровневого именования. Файлы (и другие объекты) имеют символические имена для пользователей, но могут также иметь внутренние двоичные имена для использования самой системой. Например, в операции открыть файл пользователь задает символическое имя, а в ответ получает двоичное имя, которое и использует во всех других операциях с данным файлом.

Способы формирования двоичных имен различаются в разных системах:

- имя может указывать на сервер и файл;
- в качестве двоичных имен при просмотре символьных имен возвращаются мандаты, содержащие помимо прав доступа либо физический номер машины с сервером, либо сетевой адрес сервера, а также номер файла.

В ответ на символьное имя некоторые системы могут возвращать несколько двоичных имен (для файла и его дублей), что позволяет повысить надежность работы с файлом.

5.1.3 Семантика разделения файлов.

UNIX-семантика.

Естественная семантика однопроцессорной ЭВМ - если за операцией записи следует чтение, то результат определяется последней из предшествующих операций записи. В распределенной системе такой семантики достичь легко только в том случае, когда имеется один файл-сервер, а клиенты не имеют кэшей. При наличии кэшей семантика нарушается. Надо либо сразу все изменения в кэшах отражать в файлах, либо менять семантику разделения файлов.

Еще одна проблема - трудно сохранить семантику общего указателя файла (в UNIX он общий для открывшего файл процесса и его дочерних процессов) - для процессов на разных ЭВМ трудно иметь общий указатель.

Неизменяемые файлы - очень радикальный подход к изменению семантики разделения файлов.

Только две операции - создать и читать. Можно заменить новым файлом старый - т.е. можно менять директории. Если один процесс читает файл, а другой его подменяет, то можно позволить первому процессу доработать со старым файлом, в то время как другие процессы могут уже работать с новым.

Семантика сессий.

Изменения открытого файла видны только тому процессу (или машине), который производит эти изменения, а лишь после закрытия файла становятся видны другим процессам (или машинам). Что происходит, если два процесса одновременно работали с одним файлом - либо результат будет определяться

процессом, последним закрывшим файл, либо можно только утверждать, что один из двух вариантов файла станет текущим.

Транзакции.

Процесс выдает операцию «НАЧАЛО ТРАНЗАКЦИИ», сообщая тем самым, что последующие операции должны выполняться без вмешательства других процессов. Затем выдает последовательность чтений и записей, заканчивающуюся операцией «КОНЕЦ ТРАНЗАКЦИИ». Если несколько транзакций стартуют в одно и то же время, то система гарантирует, что результат будет таким, каким бы он был в случае последовательного выполнения транзакций (в неопределенном порядке). Пример - банковские операции.

5.2 Реализация распределенных файловых систем.

Выше были рассмотрены аспекты распределенных файловых систем, которые видны пользователю. Ниже рассматриваются реализационные аспекты.

5.2.1 Использование файлов.

Приступая к реализации очень важно понимать, как система будет использоваться. Приведем результаты некоторых исследований использования файлов (статических и динамических) в университетах. Очень важно оценивать представительность исследуемых данных.

a) большинство файлов имеют размер менее 10К. (Следует перекачивать целиком).

b) чтение встречается гораздо чаще записи. (Кэширование).

c) чтение и запись последовательны, произвольный доступ редок.

(Упреждающее кэширование, чтение с запасом, выталкивание после записи следует группировать).

d) большинство файлов имеют короткое время жизни. (Создавать файл в клиенте и держать его там до уничтожения).

e) мало файлов разделяются (кэширование в клиенте и семантика сессий).

f) существуют различные классы файлов с разными свойствами.

(Следует иметь в системе разные механизмы для разных классов).

5.2.2 Структура системы.

Есть ли *разница между клиентами и серверами*? Имеются системы, где все машины имеют одно и то же ПО и любая машина может предоставлять файловый сервис. Есть системы, в которых серверы являются обычными пользовательскими процессами и могут быть сконфигурированы для работы на одной машине с клиентами или на разных. Есть системы, в которых клиенты и серверы являются фундаментально разными машинами с точки зрения аппаратуры или ПО (требуют различных ОС, например).

Второй вопрос - должны ли быть файловый сервер и сервер директорий отдельными серверами или быть объединенными в один сервер. Разделение

позволяет иметь разные серверы директорий (UNIX, MS-DOS) и один файловый сервер. Объединение позволяет сократить коммуникационные издержки.

В случае разделения серверов и при наличии разных серверов директорий для различных поддеревьев возникает следующая проблема. Если первый вызванный сервер будет поочередно обращаться ко всем следующим, то возникают большие коммуникационные расходы. Если же первый сервер передает остаток имени второму, а тот третьему, и т.д., то это не позволяет использовать RPC.

Возможный выход - использование кэша подсказок. Однако в этом случае при получении от сервера директорий устаревшего двоичного имени клиент должен быть готов получить отказ от файлового сервера и повторно обращаться к серверу директорий (клиент может не быть конечным пользователем!).

Последний важный вопрос - должны ли серверы хранить информацию о клиентах.

Серверы с состоянием. Достоинства.

a) Короче сообщения (двоичные имена используют таблицу открытых файлов).
b) выше эффективность (информация об открытых файлах может храниться в оперативной памяти).
c) блоки информации могут читаться с упреждением.
d) убедиться в достоверности запроса легче, если есть состояние (например, хранить номер последнего запроса).
e) возможна операция захвата файла.

Серверы без состояния. Достоинства.

a) устойчивость к ошибкам.
b) не требуется операций ОТКРЫТЬ/ЗАКРЫТЬ.
c) не требуется память для таблиц.
d) нет ограничений на число открытых файлов.
e) нет проблем при крахе клиента.

5.2.3 Кэширование.

В системе клиент-сервер с памятью и дисками есть четыре потенциальных места для хранения файлов или их частей.

Во-первых, хранение файлов **на дисках сервера**. Нет проблемы консистентности, так как только одна копия файла существует. Главная проблема - эффективность, поскольку для обмена с файлом требуется передача информации в обе стороны и обмен с диском.

Во-вторых, кэширование **в памяти сервера**. Две проблемы - помещать в кэш файлы целиком или блоки диска, и как осуществлять выталкивание из кэша.

Коммуникационные издержки остаются.

Избавиться от коммуникаций позволяет кэширование в машине клиента.

В третьих, кэширование **на диске клиента**. Оно может не дать преимуществ перед кэшированием в памяти сервера, а сложность повышается значительно.

Поэтому рассмотрим подробнее четвертый вариант - организацию кэширования **в памяти клиента**. При этом имеется три различных способа:

a) кэширование в каждом процессе. (Хорошо, если с файлом активно работает один процесс - многократно открывает и закрывает файл, читает и пишет, например в случае процесса базы данных).

b) кэширование в ядре. (Накладные расходы на обращение к ядру).

c) кэш-менеджер в виде отдельного процесса. (Ядро освобождается от функций файловой системы, но на пользовательском уровне трудно эффективно использовать память, особенно в случае виртуальной памяти. Возможна фиксация страниц, чтобы избежать обменов с диском).

Оценить выбор того или иного способа можно только при учете характера приложений и данных о быстродействии процессоров, памятей, дисков и сети.

Консистентность кэшей.

Кэширование в клиенте создает серьезную проблему - сложность поддержания кэшей в согласованном состоянии.

Алгоритм со сквозной записью.

Этот алгоритм, при котором модифицируемые данные пишутся в кэш и сразу же посылаются серверу, не является решением проблемы. При его использовании в мультипроцессорах все кэши “подслушивали” шину, через которую там осуществляются все “сквозные” записи в память, и сразу же обновляли находящиеся в них данные. В распределенной системе такое “подслушивание” невозможно, а требуется перед использованием данных из кэша проверять, не устарела ли информация в кэше. Кроме того, запись вызывает коммуникационные расходы.

Алгоритм с отложенной записью. Через регулярные промежутки времени все модифицированные блоки пишутся в файл (так на традиционных ЭВМ работает ОС UNIX). Эффективность выше, но семантика непонятна пользователю.

Алгоритм записи в файл при закрытии файла. Реализует семантику сессий. Такой алгоритм, на первый взгляд, кажется очень неудачным для ситуаций, когда несколько процессов одновременно открыли один файл и модифицировали его. Однако, аналогичная картина происходит и на традиционной ЭВМ, когда два процесса на одной ЭВМ открывают файл, читают его, модифицируют в своей памяти и пишут назад в файл.

Алгоритм централизованного управления. Можно выдержать семантику UNIX, но не эффективно, ненадежно, и плохо масштабируется.

5.2.4 *Размножение.*

Система может предоставлять такой сервис, как поддержание для указанных файлов нескольких копий на различных серверах. Главные цели:

1) Повысить надежность.

2) Повысить доступность (крах одного сервера не вызывает недоступность размноженных файлов).

3) Распределить нагрузку на несколько серверов.

Имеются три схемы реализации размножения:

a) Явное размножение (непрозрачно). В ответ на открытие файла пользователю выдаются несколько двоичных имен, которые он должен использовать для явного дублирования операций с файлами.
b) «Ленивое» размножение. Сначала копия создается на одном сервере, а затем он сам автоматически создает (в свободное время) дополнительные копии и обеспечивает их поддержание.
c) Симметричное размножение. Все операции одновременно вызываются в нескольких серверах и одновременно выполняются.

Протоколы коррекции.

Просто посылка сообщений с операцией коррекции каждой копии является не очень хорошим решением, поскольку в случае аварий некоторые копии могут остаться не скорректированными. Имеются два алгоритма, которые решают эту проблему.

(1) Метод размножения главной копии. Один сервер объявляется главным, а остальные - подчиненными. Все изменения файла посылаются главному серверу. Он сначала корректирует свою локальную копию, а затем рассылает подчиненным серверам указания о коррекции. Чтение файла может выполнять любой сервер. Для защиты от рассогласования копий в случае краха главного сервера до завершения им рассылки всех указаний о коррекции, главный сервер до выполнения коррекции своей копии запоминает в стабильной памяти задание на коррекцию. Слабость - выход из строя главного сервера не позволяет выполнять коррекции.

(2) Метод одновременной коррекции всех копий. Все изменения файла посылаются (используя надежные и неделимые широковещательные рассылки) всем серверам. Чтение файла может выполнять любой сервер.

(3) Метод голосования. Идея - запрашивать чтение и запись файла у многих серверов (запись - у всех!). Для успешного выполнения записи требуется, чтобы Nw серверов ее выполнили. При этом у всех этих серверов должно быть согласие относительно номера текущей версии файла. Этот номер увеличивается на единицу с каждой коррекцией файла. Для выполнения чтения достаточно обратиться к Nr серверам и воспользоваться одним из тех, кто имеет последнюю версию файла. Значения для кворума чтения (Nr) и кворума записи (Nw) должны удовлетворять соотношению Nr+Nw>N. Поскольку чтение является более частой операцией, то естественно взять Nr=1. Однако в этом случае для кворума записи потребуются все серверы.

5.2.5 **Пример: Sun Microsystem's Network File System (NFS).**

Изначально реализована Sun Microsystem в 1985 году для использования на своих рабочих станций на базе UNIX. В настоящее время поддерживается также другими фирмами для UNIX и других ОС (включая MS-DOS). Интересны следующие аспекты NFS - архитектура, протоколы и реализация.

Архитектура NFS.

Позволяет иметь произвольное множество клиентов и серверов на произвольных ЭВМ локальной или широкомасштабной сети.

Каждый сервер экспортирует некоторое число своих директорий для доступа к ним удаленных клиентов. При этом экспортируются директории со всеми своими поддиректориями, т.е. фактически поддеревья. Список экспортируемых директорий хранится в специальном файле, что позволяет при загрузке сервера автоматически их экспортировать.

Клиент получает доступ к экспортированным директориям путем их монтирования. Если клиент не имеет дисков, то может монтировать директории в свою корневую директорию.

Если несколько клиентов одновременно смонтировали одну и ту же директорию, то они могут разделять файлы в общей директории без каких либо дополнительных усилий. Простота - достоинство NFS.

Протоколы NFS.

Поскольку одна из целей NFS - поддержка гетерогенных систем, клиенты и серверы могут работать на разных ЭВМ с различной архитектурой и различными ОС. Поэтому необходимо иметь строгие протоколы их взаимодействия. NFS имеет два таких протокола.

Первый протокол поддерживает ***монтирование***. Клиент может послать серверу составное имя директории (имя пути) и попросить разрешения на ее монтирование. Куда будет монтировать директорию клиент для сервера значения не имеет и поэтому не сообщается ему. Если путь задан корректно и директория определена как экспортируемая, то сервер возвращает клиенту дескриптор директории. Дескриптор содержит поля, уникально идентифицирующие тип ЭВМ, диск, номер i-вершины (понятие ОС UNIX) для данной директории, а также информацию о правах доступа к ней. Этот дескриптор используется клиентом в последующих операциях с директорией.

Многие клиенты монтируют требуемые удаленные директории автоматически при запуске (используя командную процедуру shell-интерпретатора ОС UNIX).
Версия ОС UNIX, разработанная Sun (Solaris), имеет свой специальный режим автоматического монтирования. С каждой локальной директорией можно связать множество удаленных директорий. Когда открывается файл, отсутствующий в локальной директории, ОС посылает запросы всем серверам (владеющим указанными директориями). Кто ответит первым, директория того и будет смонтирована. Такой подход обеспечивает и надежность, и эффективность (кто свободнее, тот раньше и ответит). При этом подразумевается, что все альтернативные директории идентичны. Поскольку NFS не поддерживает размножение файлов или директорий, то такой режим автоматического монтирования в основном используется для директорий с кодами программ или других редко изменяемых файлов.

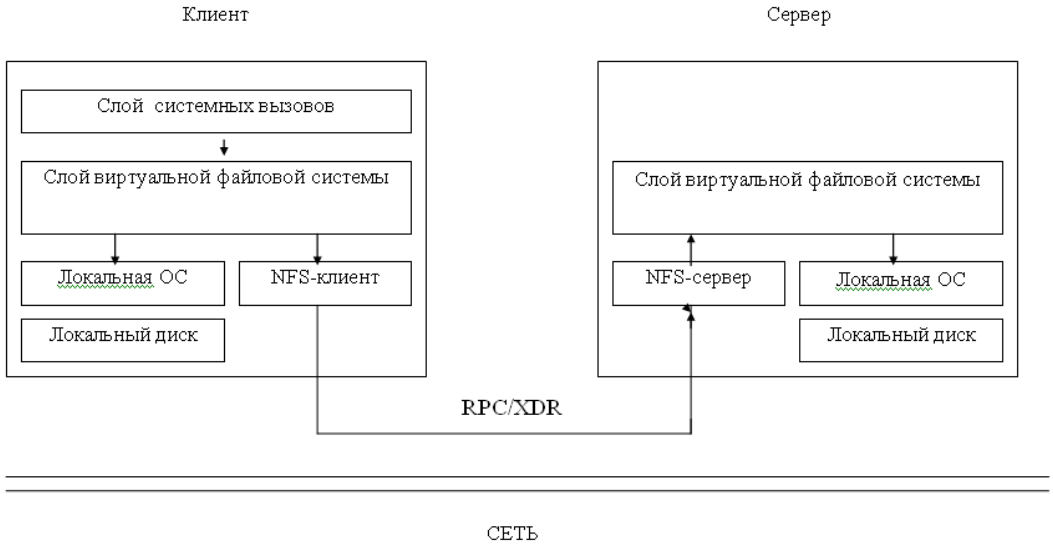
Второй протокол - для доступа к директориям и файлам. Клиенты посылают сообщения, чтобы манипулировать директориями, читать и писать файлы. Можно получить атрибуты файла. Поддерживается большинство системных

вызовов ОС UNIX, исключая OPEN и CLOSE. Для получения дескриптора файла по его символическому имени используется операция LOOKUP, отличающаяся от открытия файла тем, что никаких внутренних таблиц не создается. Таким образом, серверы в NFS не имеют состояния (stateless). Поэтому для захвата файла используется специальный механизм.

NFS использует механизм защиты UNIX. В первых версиях все запросы содержали идентификатор пользователя и его группы (для проверки прав доступа). Несколько лет эксплуатации системы показали слабость такого подхода. Теперь используется криптографический механизм с открытыми ключами для проверки законности каждого запроса и ответа. Данные не шифруются.

Все ключи, используемые для контроля доступа, поддерживаются специальным сервисом (и серверами) - сетевым информационным сервисом (NIS). Храня пары (ключ, значение), сервис обеспечивает выдачу значения кода при правильном подтверждении ключей. Кроме того, он обеспечивает отображение имен машин на их сетевые адреса, и другие отображения. NIS-серверы используют схему главный -подчиненные для реализации размножения («ленивое» размножение).

Реализация NFS.



(XDR - External Data Represantation)

Задача уровня виртуальной файловой системы - поддерживать для каждого открытого файла строку в таблице (v-вершину), аналогичную i-вершине UNIX. Эта строка позволяет различать локальные файлы от удаленных. Для удаленных файлов вся необходимая информация хранится в специальной r-вершине в NFS-клиенте, на которую ссылается v-вершина. У сервера нет никаких таблиц.

Передачи информации между клиентом и сервером NFS производятся блоками размером 8К (для эффективности).

Два кэша - кэш данных и кэш атрибутов файлов (обращения к ним очень часты, разработчики NFS исходили из оценки 90%). Реализована семантика отложенной записи - предмет критики NFS.

Имеется также кэш подсказок для ускорения получения v-вершины по символическому имени. При использовании устаревшей подсказки NFS-клиент будет обращаться к NFS-серверу и корректировать свой кэш (пользователь об этом ничего не должен знать).

6 Распределенная общая память (DSM - Distributed Shared Memory).

Традиционно распределенные вычисления базируются на модели передачи сообщений, в которой данные передаются от процессора к процессору в виде сообщений. Удаленный вызов процедур фактически является той же самой моделью (или очень близкой).

DSM - виртуальное адресное пространство, разделяемое всеми узлами (процессорами) распределенной системы. Программы получают доступ к данным в DSM примерно так же, как это происходит при реализации виртуальной памяти традиционных ЭВМ. В системах с DSM данные могут перемещаться между локальными памятями разных компьютеров аналогично тому, как они перемещаются между оперативной и внешней памятью одного компьютера.

6.1 Достоинства DSM.

(1) В модели передачи сообщений программист обеспечивает доступ к разделяемым данным посредством явных операций посылки и приема сообщений. При этом приходится квантовать алгоритм, обеспечивать своевременную смену информации в буферах, преобразовывать индексы массивов. Все это сильно усложняет программирование и отладку. DSM скрывает от программиста пересылку данных и обеспечивает ему абстракцию разделяемой памяти, к использованию которой он уже привык на мультипроцессорах. Программирование и отладка с использованием DSM гораздо проще.

(2) В модели передачи сообщений данные перемещаются между двумя различными адресными пространствами. Это делает очень трудным передачу сложных структур данных между процессами. Например, передача данных по ссылке и передача структур данных, содержащих указатели, вызывает серьезные проблемы. В системах с DSM этих проблем нет, что несомненно упрощает разработку распределенных приложений.

(3) Объем суммарной физической памяти всех узлов может быть огромным. Эта огромная память становится доступна приложению без издержек, связанных в традиционных системах с дисковыми обменами. Это достоинство становится все весомее в связи с тем, что скорости процессоров и коммуникаций растут быстрее скоростей памяти и дисков.

(4) DSM-системы могут наращиваться практически беспредельно в отличие от систем с разделяемой памятью, т.е. являются масштабируемыми.

(5) Программы, написанные для мультипроцессоров с общей памятью, могут в принципе без каких-либо изменений выполняться на DSM-системах (по крайней мере, они могут быть легко перенесены на DSM-системы).

По существу, DSM-системы преодолевают архитектурные ограничения мультипроцессоров и сокращают усилия, необходимые для написания программ для распределенных систем. Обычно они реализуются программно-аппаратными средствами, но в 90-х годах появилось несколько коммерческих MPP с DSM, реализованной аппаратно (Convex SPP, KSR1, Origin 2000). В конце 2004 года второй в списке TOP-500 являлась система, состоящая из 20 DSM-кластеров SGI ALTIX 3000, каждый по 512 процессоров.

6.2 Алгоритмы реализации DSM.

При реализации DSM центральными являются следующие вопросы.

1) как поддерживать информацию о расположении удаленных данных.
2) как снизить при доступе к удаленным данным коммуникационные задержки и большие накладные расходы, связанные с выполнением коммуникационных протоколов.
3) как сделать разделяемые данные доступными одновременно на нескольких узлах для того, чтобы повысить производительность системы.

Рассмотрим четыре возможных алгоритма реализации DSM.

6.2.1 Алгоритм с центральным сервером.

Все разделяемые данные поддерживает центральный сервер. Он возвращает данные клиентам по их запросам на чтение, по запросам на запись он корректирует данные и посылает клиентам в ответ квитанции. Клиенты могут использовать тайм-аут для посылки повторных запросов при отсутствии ответа сервера. Дубликаты запросов на запись могут распознаваться путем нумерации запросов. Если несколько повторных обращений к серверу остались без ответа, приложение получит отрицательный код ответа (это обеспечит клиент).

Алгоритм прост в реализации, но сервер будет узким местом.

Можно разделяемые данные распределить между несколькими серверами. В этом случае клиент должен уметь определять, к какому серверу надо обращаться при каждом доступе к разделяемой переменной. Можно, например, распределить между серверами данные в зависимости от их адресов и использовать функцию отображения для определения нужного сервера.

Независимо от числа серверов, работа с памятью будет требовать коммуникаций и катастрофически замедлится.

6.2.2 Миграционный алгоритм.

В отличие от предыдущего алгоритма, когда запрос к данным направлялся в место их расположения, в этом алгоритме меняется расположение данных - они перемещаются в то место, где потребовались. Это позволяет последовательные обращения к данным осуществлять локально. Миграционный алгоритм позволяет обращаться к одному элементу данных в любой момент времени только одному узлу.

Обычно мигрирует целиком страницы или блоки данных, а не запрашиваемые единицы данных. Это позволяет воспользоваться присущей приложениям локальностью доступа к данным для снижения стоимости миграции. Однако, такой подход приводит к трэшингу, когда страницы очень часто мигрируют между узлами при малом количестве обслуживаемых запросов. Причиной этого может быть так называемое “ложное разделение”, когда разным процессорам нужны разные данные, но эти данные расположены в одном блоке или странице. Некоторые системы позволяют задать время, в течение которого страница насильно удерживается в узле для того, чтобы успеть выполнить несколько обращений к ней до ее миграции в другой узел.

Миграционный алгоритм позволяет интегрировать DSM с виртуальной памятью, обеспечивающейся операционной системой в отдельных узлах. Если размер страницы DSM совпадает с размером страницы виртуальной памяти (или кратен ей), то можно обращаться к разделяемой памяти обычными машинными командами, воспользовавшись аппаратными средствами проверки наличия в оперативной памяти требуемой страницы и замены виртуального адреса на физический. Конечно, для этого виртуальное адресное пространство процессоров должно быть достаточно, чтобы адресовать всю разделяемую память. При этом, несколько процессов в одном узле могут разделять одну и ту же страницу.

Для определения места расположения блоков данных миграционный алгоритм может использовать сервер, отслеживающий перемещения блоков, либо воспользоваться механизмом подсказок в каждом узле. Возможна и широковещательная рассылка запросов.

6.2.3 Алгоритм размножения для чтения.

Предыдущий алгоритм позволял обращаться к разделяемым данным в любой момент времени только процессам в одном узле (в котором эти данные находятся). Данный алгоритм расширяет миграционный алгоритм механизмом размножения блоков данных, позволяя либо многим узлам иметь возможность одновременного доступа по чтению, либо одному узлу иметь возможность читать и писать данные (протокол многих читателей и одного писателя).

При использовании такого алгоритма требуется отслеживать расположение всех блоков данных и их копий. Например, каждый собственник блока может отслеживать расположение его копий.
Безусловно, производительность повышается за счет возможности одновременного доступа по чтению, но запись требует серьезных затрат для уничтожения всех устаревших копий блока данных или их коррекции. Да и модель многих читателей и одного писателя мало подходит для параллельных программ.

6.2.4 Алгоритм размножения для чтения и записи.

Этот алгоритм является расширением предыдущего алгоритма. Он позволяет многим узлам иметь одновременный доступ к разделяемым данным на чтение и запись (протокол многих читателей и многих писателей). Поскольку много узлов могут писать данные параллельно, требуется для поддержания согласованности данных контролировать доступ к ним.

Одним из способов обеспечения согласованности данных является использование специального процесса для упорядочивания модификаций памяти. Все узлы, желающие модифицировать разделяемые данные должны посылать свои модификации этому процессу. Он будет присваивать каждой модификации очередной номер и рассылать его широковещательно вместе с модификацией всем узлам, имеющим копию модифицируемого блока данных. Каждый узел будет осуществлять модификации в порядке возрастания их номеров. Разрыв в номерах полученных модификаций будет означать потерю одной или нескольких модификаций. В этом случае узел может запросить недостающие модификации.

Данный алгоритм может существенно снизить среднюю стоимость доступа к данным только тогда, когда количество чтений значительно превышает количество записей.

В общем случае, все перечисленные выше алгоритмы являются слишком неэффективными, чтобы их можно было использовать для преодоления архитектурных ограничений мультипроцессоров и сокращения усилий, необходимых для написания программ для распределенных систем.

Добиться эффективности можно только изменив семантику обращений к памяти.

Для упрощения понимания основных идей алгоритмов реализации DSM мы в дальнейшем будем исходить из того, что все работает надежно (например, все сообщения доходят до адресатов) и никаких мер для обеспечения надежности предпринимать не нужно.

6.3 Модели консистентности.

Модель консистентности представляет собой некоторый договор между программами и памятью, в котором указывается, что при соблюдении программами определенных правил работы с памятью будет обеспечена определенная семантика операций чтения/записи, если же эти правила будут нарушены, то память не гарантирует правильность выполнения операций чтения/записи. В этой главе рассматриваются основные модели консистентности используемые в системах с распределенной памятью.

6.3.1 Строгая консистентность.

Модель консистентности, удовлетворяющая условию: «Операция чтения ячейки памяти с адресом X должна возвращать значение, записанное самой последней операцией записи по адресу X», называется моделью строгой консистентности. Указанное выше условие кажется довольно естественным и очевидным, однако оно предполагает наличие в системе понятия абсолютного времени для определения «наиболее последней операции записи».

Все однопроцессорные системы обеспечивают строгую консистентность, но в распределенных многопроцессорных системах ситуация намного сложнее. Предположим, что переменная X расположена в памяти машины B, и процесс, который выполняется на машине A, пытается прочитать значение этой переменной в момент времени T1. Для этого машине B посылается запрос переменной X. Немного позже, в момент времени T2, процесс, выполняющийся на машине B, производит операцию записи нового значения в переменную X. Для обеспечения строгой консистентности операция чтения должна возвратить в машину A старое значение переменной вне зависимости от того, где расположена машина A и насколько близки между собой два момента времени T1 и T2. Однако, если T1-T2 равно 1 нсек, и машины расположены друг от друга на расстоянии 3-х метров, то сигнал о запросе значения переменной должен быть передан на машину B со скоростью в 10 раз превышающей скорость света, что невозможно.

P2: R(x)1 P2: R(x)0 R(x)1

а) Б)

а) Строго консистентная память

б) Память без строгой консистентности

6.3.2 Последовательная консистентность.

Строгая консистентность представляет собой идеальную модель для программирования, но ее, к сожалению программистов, невозможно реализовать для распределенных систем. Однако, практический опыт показывает, что в некоторых случаях можно обходиться и более «слабыми» моделями. Все эти методы опираются на то, что должна соблюдаться последовательность определенных событий записи и чтения.

Последовательную консистентность впервые определил Lamport в 1979 г.

По его определению, модель последовательной консистентности памяти должна удовлетворять следующему условию: «Результат выполнения должен быть тот же, как если бы операторы всех процессоров выполнялись бы в некоторой последовательности, в которой операторы каждого индивидуального процессора расположены в порядке, определяемом программой этого процессора»

Последовательная консистентность не гарантирует, что операция чтения возвратит значение, записанное другим процессом наносекундой или даже минутой раньше, в этой модели только точно гарантируется, что все процессы должны «видеть» одну и ту же последовательность записей в память.

Результат повторного выполнения параллельной программы в системе с последовательной консистентностью (как, впрочем, и при строгой консистентности) может не совпадать с результатом предыдущего выполнения этой же программы, если в программе нет регулирования операций доступа к памяти с помощью механизмов синхронизации.

Два примера правильного выполнения одной программы. В примерах используются следующие обозначения:

W(x)1 - запись значения 1 в переменную x;

R(x)0 - чтение значения 0 из переменной x.

P1:	W(x)1			W(y)1	
P2:			W(z)1		
P3:		R(x)0	R(y)0	R(z)1	R(y)0
P4:		R(x)0	R(y)1	R(z)1	R(x)1

В этом примере процессы «видят» записи в порядке W(z)1, W(x)1,W(y)1 или W(x)1, W(z)1,W(y)1.

P1:	W(x)1			W(y)1	
P2:			W(z)1		
P3:		R(x)0	R(y)1	R(z)0	R(y)1

P4:	R(x)1	R(y)1	R(z)0	R(x)1

В этом примере процессы «видят» записи в порядке W(x)1, W(y)1,W(z)1.

Два примера неправильного выполнения той же программы.

P1: W(x)1			W(y)1	
P2:		W(z)1		
P3:	R(x)0	R(y)0	R(z)1	R(y)0
P4:	R(x)0	R(y)1	R(z)0	R(x)1

Процессы P3 и P4 «видят» записи W(y)1 и W(z)1 в разном порядке.

P1: W(x)1			W(y)1	
P2:		W(z)1		
P3:	R(x)1	R(y)0	R(z)1	R(y)1
P4:	R(x)0	R(y)1	R(z)1	R(x)0

Процесс P4 «видит» записи W(x)1 и W(y)1 не в том порядке, как они выполнялись в процессе P1.

Описанный ранее миграционный алгоритм реализует последовательную консистентность.

Последовательная консистентность может быть реализована гораздо более эффективно следующим образом. Страницы, доступные на запись, размножаются, но операции с разделяемой памятью (и чтение, и запись) не должны начинаться на каждом процессоре до тех пор, пока не завершится выполнение предыдущей операции записи, выданной этим процессором, т.е. будут скорректированы все копии соответствующей страницы.

Централизованный алгоритм. Процесс посылает координатору запрос на модификацию переменной и ждет от него указания о проведении этой модификации. Такое указание координатор рассылает сразу всем владельцам копий этой переменной. Каждый процесс выполняет эти указания по мере их получения. Поскольку сообщения от координатора приходят каждому процессу в том порядке, в котором они были им посланы, то все процессы корректируют свои копии переменных в этом едином порядке.

Децентрализованный алгоритм. Процесс посылает посредством механизма упорядоченного широковещания (неделимые широковещательные рассылки) указание о модификации переменной всем владельцам копий соответствующей страницы (включая и себя) и ждет получения этого своего собственного указания.

6.3.3 Причинная консистентность.

Причинная модель консистентности памяти представляет собой более «слабую» модель по сравнению с последовательной моделью, поскольку в ней не всегда требуется, чтобы все процессы «видели» одну и ту же последовательность записей в память, а проводится различие между потенциально зависимыми операциями записи, и независимыми.

Рассмотрим пример. Предположим, что процесс P1 модифицировал переменную ***x***, затем процесс P2 прочитал ***x*** и модифицировал ***y***. В этом случае модификация ***x*** и модификация ***y*** потенциально причинно зависимы, так как новое значение ***y*** могло зависеть от прочитанного значения переменной ***x***. С другой стороны, если два процесса одновременно изменяют значения одной и той же или различных переменных, то между этими событиями нет причинной связи. Операции записи, которые причинно не зависят друг от друга, называются параллельными.

Причинная модель консистентности памяти определяется следующим условием: «Последовательность операций записи, которые потенциально причинно зависимы, должна наблюдаться всеми процессами системы одинаково, параллельные операции записи могут наблюдаться разными узлами в разном порядке».

Пример.

(а) Нарушение модели причинной консистентности

P1:	W(x)1				
P2:		R(x)1	W(x)2		
P3:				R(x)2	R(x)1
P4:				R(x)1	R(x)2

(б) корректная последовательность для модели причинной консистентности.

P1:	W(x)1			W(x)3	
P2:		R(x)1	W(x)2		
P3:		R(x)1		R(x)3	R(x)2
P4:		R(x)1		R(x)2	R(x)3

При реализации причинной консистентности в случае размножения страниц выполнение записи в общую память требует ожидания выполнения только тех предыдущих операций записи, от которых эта запись потенциально причинно зависит. Параллельные операции записи не задерживают выполнение друг друга (и не требуют неделимости широковещательных рассылок всем владельцам копий страницы).

Реализация причинной консистентности может осуществляться следующим образом:

- все модификации переменных на каждом процессоре нумеруются;
- всем процессорам вместе со значением модифицируемой переменной рассылается номер этой модификации на данном процессоре, а также номера модификаций всех процессоров, известных данному процессору к этому моменту;

- выполнение любой модификации на каждом процессоре задерживается до тех пор, пока он не получит и не выполнит все те модификации других процессоров, о которых было известно процессору - автору задерживаемой модификации.

6.3.4 PRAM консистентность и процессорная консистентность.

PRAM (Pipelined RAM) консистентность определяется следующим образом: «Операции записи, выполняемые одним процессором, видны всем остальным процессорам в том порядке, в каком они выполнялись, но операции записи, выполняемые разными процессорами, могут быть видны в произвольном порядке».

Пример допустимой последовательности событий в системе с PRAM консистентностью.

P1:	W(x)1			
P2:	R(x)1	W(x)2		
P3:			R(x)1	R(x)2
P4:			R(x)2	R(x)1

Преимущество модели PRAM консистентности заключается в простоте ее реализации, поскольку операции записи на одном процессоре могут быть конвейеризованы: можно продолжать выполнение процесса и выполнять другие операции с общей памятью не дожидаясь завершения предыдущих операций записи (модификации всех копий страниц, например), необходимо только быть уверенным, что все процессоры увидят эти записи в одном и том же порядке.

PRAM консистентность может приводить к результатам, противоречащим интуитивному представлению. Пример:

Процесс P1	Процесс P2
.....	.....
a = 1;	b = 1;
if (b==0) kill (P2);	if (a==0) kill (P1);
.....	.....

Оба процесса могут быть убиты, что невозможно при последовательной консистентности.

Модель процессорной консистентности отличается от модели PRAM консистентности тем, что в ней дополнительно требуется когерентность памяти: «Для каждой переменной x есть общее согласие относительно порядка, в котором процессоры модифицируют эту переменную, операции записи в разные переменные - параллельны». Таким образом, к упорядочиванию записей каждого процессора добавляется упорядочивание записей в переменные или группы переменных (например, находящихся в независимых блоках памяти).

Децентрализованный алгоритм. За каждую группу переменных отвечает свой координатор, который получает от процессов запросы на модификацию и рассылает всем указания о проведении модификации. Чтобы не нарушить порядок получения процессами указаний о модификациях различных

переменных, запрошенных одним процессом у разных координаторов, надо каждому процессу нумеровать свои модификации, и эти номера должны рассылаться всем вместе с указаниями о проведении модификаций. Тогда любой процесс, получающий указание о проведении модификации, может задержать его выполнение до получения недостающих указаний о предшествующих модификациях соответствующего процесса.

6.3.5. Слабая консистентность.

Модель PRAM консистентности производительнее и эффективнее моделей с более строгой консистентностью, но и ее ограничения для многих приложений не всегда являются необходимыми, так как требуют получение всеми процессорами информации о каждой операции записи, выполняемой на некотором процессоре.

Рассмотрим, для примера, процесс, который в критической секции циклически читает и записывает значение некоторых переменных. Даже, если остальные процессоры и не пытаются обращаться к этим переменным до выхода первого процесса из критической секции, для удовлетворения требований описанных выше моделей консистентности они должны «видеть» все записи первого процессора в порядке их выполнения, что, естественно, совершенно не нужно. Наилучшее решение в такой ситуации - это позволить первому процессу завершить выполнение критической секции и, только после этого, переслать остальным процессам значения модифицированных переменных, не заботясь о пересылке промежуточных результатов, и порядка их вычисления внутри критической секции.

Предложенная в 1986 г. (Dubois et al.) модель слабой консистентности, основана на выделении среди переменных специальных синхронизационных переменных (доступ к которым производится специальной операцией синхронизации памяти) и описывается следующими правилами:

1. *Доступ к синхронизационным переменным определяется моделью последовательной консистентности;*
2. *Доступ к синхронизационным переменным запрещен (задерживается), пока не выполнены все предыдущие операции записи;*
3. *Доступ к данным (запись, чтение) запрещен, пока не выполнены все предыдущие обращения к синхронизационным переменным.*

Первое правило определяет, что все процессы «видят» обращения к синхронизационным переменным в определенном (одном и том же) порядке (а “видеть” они могут только посредством чтения обычных переменных!).

Второе правило гарантирует, что выполнение процессором операции обращения к синхронизационной переменной возможно только после «выталкивания» конвейера (полного завершения выполнения всех предыдущих операций записи переменных, выданных данным процессором). При этом, все выполненные процессом записи станут гарантированно видны остальным процессам только после выполнения ими синхронизации. Третье правило определяет, что при обращении к обычным (не синхронизационным) переменным на чтение или запись, все предыдущие обращения к синхронизационным переменным должны быть выполнены полностью. Выполнив синхронизацию памяти (эта операция

обозначена ниже буквой S) перед обращением к общей переменной, процесс может быть уверен, что получит правильное значение этой переменной (то, которое записал какой-то процесс, успевший сообщить об этом всем посредством синхронизации памяти).

а) Пример допустимой последовательности событий.

P1:	W(x)1	W(x)2	S			
P2:				R(x)1, R(x)2	S	
P3:				R(x)2, R(x)1	S	

б) Пример недопустимой последовательности событий.

P1:	W(x)1	W(x)2	S		
P2:				S	R(x)1

6.3.6 Консистентность по выходу.

В системе со слабой консистентностью возникает проблема при обращении к синхронизационной переменной: система не имеет информации о цели этого обращения - или процесс завершил модификацию общей переменной, или готовится прочитать значение общей переменной. Для более эффективной реализации модели консистентности система должна различать две ситуации: вход в критическую секцию и выход из нее.

В модели консистентности по выходу введены специальные функции обращения к синхронизационным переменным:

(1) ACQUIRE - захват синхронизационной переменной, информирует систему о входе в критическую секцию;

(2) RELEASE - освобождение синхронизационной переменной, определяет завершение критической секции.

Захват и освобождение используется для организации доступа не ко всем общим переменным, а только к тем, которые защищаются данной синхронизационной переменной. Такие общие переменные называют защищенными переменными.

Пример допустимой последовательности событий для модели с консистентностью по выходу. (Acq(L) - захват синхронизационной переменной L; Rel(L) - освобождение синхронизационной переменной).

P1:	Acq(L)	W(x)1	W(x)2	Rel(L)			
P2:					Acq(L)	R(x)2	Rel(L)
P3:							R(x)1

Следующие правила определяют требования к модели консистентности по выходу:

(1) До выполнения обращения к общей переменной, должны быть полностью выполнены все предыдущие захваты синхронизационных переменных данным процессором.

(2) Перед освобождением синхронизационной переменной должны быть закончены все операции чтения/записи, выполнявшиеся процессором прежде.
(3) Реализация операций захвата и освобождения синхронизационной переменной должны удовлетворять требованиям процессорной консистентности (последовательная консистентность не требуется, захваты разных переменных осуществляются параллельно).

При выполнении всех этих требований и использовании методов захвата и освобождения, результат выполнения программы будет таким же, как при выполнении этой программы в системе с последовательной моделью консистентности.

Существует модификация консистентности по выходу - «ленивая». В отличие от описанной («энергичной») консистентности по выходу, она не требует выталкивания всех модифицированных данных при выходе из критической секции. Вместо этого, при запросе входа в критическую секцию процессу передаются текущие значения защищенных разделяемых переменных (например, от процесса, который последним находился в критической секции, охраняемой этой синхронизационной переменной). При повторных входах в критическую секцию того же самого процесса не требуется никаких обменов сообщениями. Для того, чтобы узнать, какие переменные защищаются конкретной синхронизационной переменной, нужно фиксировать все переменные, изменяемые внутри соответствующих критических секций.

6.3.7 Консистентность по входу.

Эта консистентность представляет собой еще один пример модели консистентности, которая ориентирована на использование критических секций. Так же, как и в предыдущей модели, эта модель консистентности требует от программистов (или компиляторов) использование механизма захвата/освобождения для выполнения критических секций. Однако в этой модели требуется, чтобы каждая общая переменная была явна связана с некоторой синхронизационной переменной (или с несколькими синхронизационными переменными), при этом, если доступ к элементам массива, или различным отдельным переменным, может производиться независимо (параллельно), то эти элементы массива (общие переменные) должны быть связаны с разными синхронизационными переменными. Таким образом, вводится явная связь между синхронизационными переменными и общими переменными, которые они охраняют.

Кроме того, критические секции, охраняемые одной синхронизационной переменной, могут быть двух типов:

- секция с монопольным доступом (для модификации переменных);
- секция с немонопольным доступом (для чтения переменных).

Рассмотрим использование синхронизационных переменных.

Каждая синхронизационная переменная имеет временного владельца - последний процесс, захвативший доступ к этой переменной. Этот владелец

может в цикле выполнять критическую секцию, не посылая при этом сообщений другим процессорам. Процесс, который в данный момент не является владельцем синхронизационной переменной, но требующий ее захвата, должен послать запрос текущему владельцу этой переменной для получения права собственности на синхронизационную переменную и значений охраняемых ею общих переменных. Разрешена ситуация, когда синхронизационная переменная имеет несколько владельцев, но только в том случае, если связанные с этой переменной общие данные используются только для чтения.

Ниже приведены формальные правила, определяющие модель консистентности по входу:

(1) Процесс не может захватить синхронизационную переменную до того, пока не обновлены все переменные этого процесса, охраняемые захватываемой синхронизационной переменной;
(2) Процесс не может захватить синхронизационную переменную в монопольном режиме (для модификации охраняемых данных), пока другой процесс, владеющий этой переменной (даже в немонопольном режиме), не освободит ее;
(3) Если какой-то процесс захватил синхронизационную переменную в монопольном режиме, то ни один процесс не сможет ее захватить даже в немонопольном режиме до тех пор, пока первый процесс не освободит эту переменную, и будут обновлены текущие значения охраняемых переменных в процессе, запрашивающем синхронизационную переменную.

6.3.8 Сравнение моделей консистентности.

В приведенной ниже таблице определены отличительные характеристики описанных моделей консистентности памяти.

(а) Модели консистентности, не использующие операции синхронизации.

Консистентность	Описание
Строгая	Упорядочение всех доступов к разделяемым данным по абсолютному времени
Последовательная	Все процессы видят все записи разделяемых данных в одном и том же порядке
Причинная	Все процессы видят все причинно-связанные записи данных в одном и том же порядке
Процессорная	PRAM-консистентность + когерентность памяти
PRAM	Все процессоры видят записи любого процессора в одном и том же порядке

(б) Модели консистентности с операциями синхронизации.

Консистентность	Описание
Слабая	Разделяемые данные можно считать консистентными только после выполнения синхронизации
По выходу	Разделяемые данные, изменяемые в критической секции, становятся консистентными после выхода из нее.
По входу	Разделяемые данные, связанные с монопольной или немонопольной критической секцией, становятся консистентными при входе в нее

6.4 Протоколы когерентности.

WRITE-INVALIDATE - всем владельцам копий сообщается о их недействительности.

WRITE-UPDATE - организуется обновление копий у всех владельцев.

Выбор определяется частотами чтений и записей, а также временами оповещения и обновления.

6.5 Конструкторские решения.

6.5.1 Страничная DSM.

Общая память разбивается на порции одинаковой длины - страницы или блоки. Если выбрать длину совпадающей (или кратной) длине страницы оперативной памяти процессоров (если их память страничная), то можно будет воспользоваться механизмом защиты памяти для обнаружения отсутствующих страниц DSM и аппаратным механизмом замены виртуального адреса на физический.

К этому же типу DSM (не знающих заранее ничего о содержимом памяти) можно отнести и аппаратные реализации на базе кэшей (Convex SPP).

6.5.2 DSM на базе разделяемых переменных.

В DSM системах с разделяемыми переменными только отдельные структуры данных разделяются между процессорами. Программист должен точно определить, какие переменные в программе должны разделяться, а какие не должны.

Знание информации о режиме использования разделяемых переменных позволяет воспользоваться более эффективными протоколами когерентности.

Рассмотрим следующую программу, которая автоматически распараллеливается посредством размножения всех переменных на все процессоры и распределения между ними витков цикла.

```
s = 0;
```

```
last_A = 0;
for (i = 1; i < L2-1; i++)
{
   r = A[i-1]*A[i+1];
   C[i] = r;
   s = s + r*r;
   if (A[i]!=0) { last_A = A[i]};
 }
A[0] = last_A;
```

Переменные внутри цикла используются в следующих режимах:

A - только на чтение;

r - приватно (фактически не является разделяемой);

C - раздельно используется (любой элемент массива изменяется не более, чем одним процессом, и никакой процесс не читает элемент, изменяемый другими);

s - редукционная переменная, используемая для суммирования;

last_A - переменная, хранящая значение последнего ненулевого элемента массива A.

Приведения в консистентное состояние переменных A и r - не требуется.

Для приведения в консистентное состояние массива C необходимо при завершении цикла каждому процессу послать остальным все свои изменения в массиве C.

Для переменной s в конце цикла надо довыполнить операцию редукции - сложить все частичные суммы, полученные разными процессорами в своих копиях переменной s и разослать результат всем процессорам (если бы начальное значение переменной s было отлично от нуля, то это надо было бы учесть).

Переменной last_A на всех процессорах при выходе из цикла должно быть присвоено то значение, которое было получено на витке цикла с максимальным номером. Для этого можно фиксировать на каждом процессоре максимальный номер витка, на котором переменной присваивается значение. При распределении витков последовательными блоками между процессорами достаточно фиксировать сам факт изменения переменной каждым процессором.

Вне цикла приведение в консистентное состояние переменной A[0] не требуется, поскольку на каждом процессоре выполняется один и тот же оператор, который присваивает одно и то же значение всем копиям переменной.

6.5.3 DSM на базе объектов.

Последнюю группу образуют многопроцессорные системы с объектной организацией распределенной общей памятью. В отличие от всех остальных рассмотренных систем, программы для объектно-ориентированной DSM системы не могут напрямую использовать общие переменные, а только через специальные функции-методы. Система поддержки выполнения параллельных программ, получив запрос на использование некоторой общей

переменной, обрабатывает его, поддерживая при этом консистентное состояние разделяемых данных. Весь контроль осуществляется только программными средствами.

В тех случаях, когда для балансировки загрузки процессоров применяется миграция данных, воспользоваться соседством расположения данных в локальной памяти процессора затруднительно. В таких случаях потери эффективности из-за доступа к данным через функции могут быть вполне приемлемыми, поскольку они могут сполна компенсироваться тем выигрышем, который достигается балансировкой загрузки.

7. Обеспечение надежности в распределенных системах.

Отказом системы называется поведение системы, не удовлетворяющее ее спецификациям. Последствия отказа могут быть различными.

Отказ системы может быть вызван отказом (неверным срабатыванием) каких-то ее компонентов (процессор, память, устройства ввода/вывода, линии связи, или программное обеспечение).

Отказ компонента может быть вызван ошибками при конструировании, при производстве или программировании. Он может быть также вызван физическим повреждением, изнашиванием оборудования, некорректными входными данными, ошибками оператора, и многими другими причинами.

Отказы могут быть случайными, периодическими или постоянными.

Случайные отказы (сбои) при повторении операции исчезают.

Причиной такого сбоя может служить, например, электромагнитная помеха от проезжающего мимо трамвая. Другой пример - редкая ситуация в последовательности обращений к операционной системе от разных задач.

Периодические отказы повторяются часто в течение какого-то времени, а затем могут долго не происходить. Примеры - плохой контакт, некорректная работа ОС после обработки аварийного завершения задачи.

Постоянные (устойчивые) отказы не прекращаются до устранения их причины - разрушения диска, выхода из строя микросхемы или ошибки в программе.

Отказы по характеру своего проявления подразделяются на «византийские» (система активна и может проявлять себя по-разному, даже злонамеренно) и «пропажа признаков жизни» (частичная или полная). Первые распознать гораздо сложнее, чем вторые. Свое название они получили по имени Византийской империи (330-1453 гг.), где расцветали конспирация, интриги и обман.

Для обеспечения надежного решения задач в условиях отказов системы применяются два принципиально различающихся подхода - восстановление решения после отказа системы (или ее компонента) и предотвращение отказа системы (отказоустойчивость).

7.1. Восстановление после отказа.

Восстановление может быть прямым (без возврата к прошлому состоянию) и возвратное.

Прямое восстановление основано на своевременном обнаружении сбоя и ликвидации его последствий путем приведения некорректного состояния системы в корректное. Такое восстановление возможно только для определенного набора заранее предусмотренных сбоев.

При возвратном восстановлении происходит возврат процесса (или системы) из некорректного состояния в некоторое из предшествующих корректных состояний. При этом возникают следующие проблемы.

(1) Потери производительности, вызванные запоминанием состояний, восстановлением запомненного состояния и повторением ранее выполненной работы, могут быть слишком высоки.

(2) Нет гарантии, что сбой снова не повторится после восстановления.

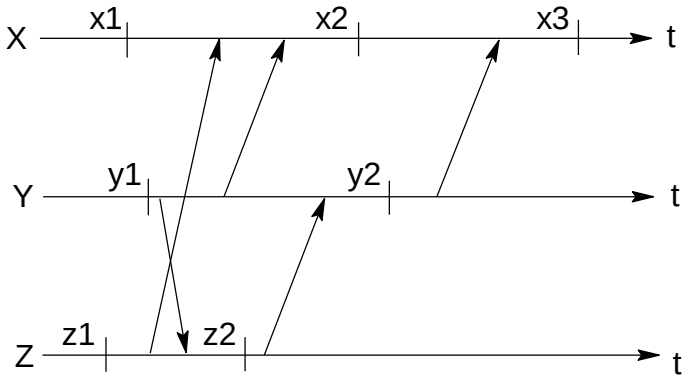
(3) Для некоторых компонентов системы восстановление в предшествующее состояние может быть невозможно (торговый автомат).

Тем не менее этот подход является более универсальным и применяется гораздо чаще первого. Дальнейшее рассмотрение будет ограничено только данным подходом.

Для восстановления состояния в традиционных ЭВМ применяются два метода (и их комбинация), основанные на промежуточной фиксации состояния либо ведении журнала выполняемых операций. Они различаются объемом запоминаемой информацией и временем, требуемым для восстановления.

Применение подобных методов в распределенных системах наталкивается на следующие трудности.

7.1.1. Сообщения-сироты и эффект домино.



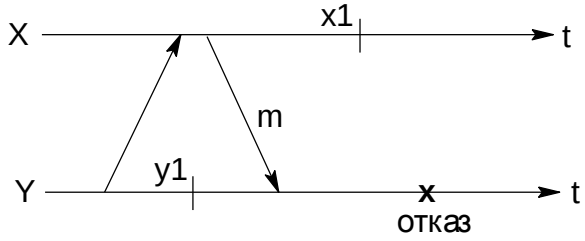
На рисунке показаны три процесса (X,Y,Z), взаимодействующие через сообщения. Вертикальные черточки показывают на временной оси моменты запоминания состояния процесса для восстановления в случае отказа. Стрелочки соответствуют сообщениям и показывают моменты их отправления и получения.

Если процесс X сломается, то он может быть восстановлен с состояния x3 без какого-либо воздействия на другие процессы.

Предположим, что процесс Y сломался после посылки сообщения m и был возвращен в состояние y2. В этом случае получение сообщения m зафиксировано в x3, а его посылка не отмечена в y2. Такая ситуация, возникшая из-за несогласованности глобального состояния, не должна допускаться (пример - сообщение содержит сумму, переводимую с одного счета на другой). Сообщение m в таком случае называется сообщением-сиротой. Процесс X должен быть возвращен в предыдущее состояние x2 и конфликт будет ликвидирован.

Предположим теперь, что процесс Z сломается и будет восстановлен в состояние z2. Это приведет к откату процесса Y в y1, а затем и процессов X и Z в начальные состояния x1 и y1. Этот эффект известен как эффект домино.

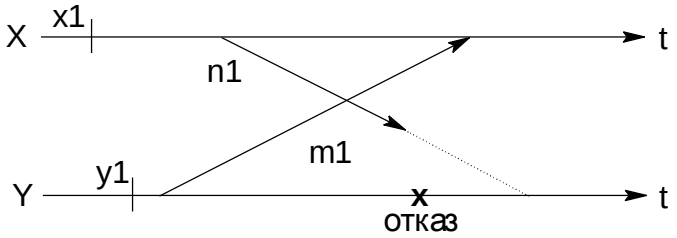
7.1.2. Потеря сообщений.



Предположим, что контрольные точки x1 и y1 зафиксированы для восстановления процессов X и Y, соответственно.

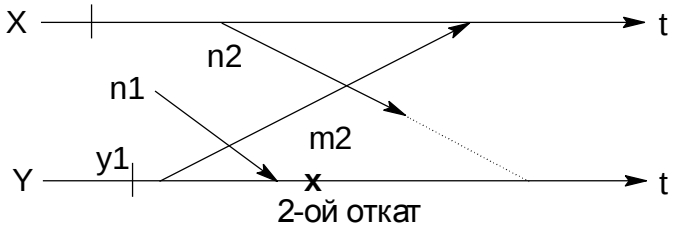
Если процесс Y сломается после получения сообщения m, и оба процесса будут восстановлены (x1,y1), то сообщение m будет потеряно (его потеря будет неотличима от потери в канале).

7.1.3. Проблема бесконечного восстановления.



Процесс Y сломался до получения сообщения n1 от X. Когда Y

вернулся в состояние y1, в нем не оказалось записи о посылке сообщения m1. Поэтому X должен вернуться в состояние x1.



После отката Y посылает m2 и принимает n1 (сообщение-призрак). Процесс X после отката к x1 посылает n2 и принимает m2. Однако X после отката уже не имеет записи о посылке n1. Поэтому Y должен повторно откатиться к y1. Теперь X должен откатиться к x1, поскольку он принял m2, о посылке которого в Y нет записи. Эта ситуация будет повторяться бесконечно.

7.1.4. Консистентное множество контрольных точек.

Описанные выше трудности показывают, что глобальная контрольная точка, состоящая из произвольной совокупности локальных контрольных точек, не обеспечивает восстановления взаимодействующих процессов.

Для распределенных систем запоминание согласованного глобального состояния является серьезной теоретической проблемой.

Множество контрольных точек называется **строго консистентным**, если во время его фиксации никаких обменов между процессами не было. Оно соответствует понятию строго консистентного глобального состояния, когда все посланные сообщения получены и нет никаких сообщений в каналах связи. Множество контрольных точек называется **консистентным**, если для любой зафиксированной операции приема сообщения, соответствующая операция посылки также зафиксирована (нет сообщений-сирот).

Простой метод фиксации консистентного множества контрольных точек - фиксация локальной контрольной точки после каждой операции посылки сообщения. При этом посылка сообщения и фиксация должны быть единой неделимой операцией (транзакцией). Множество последних локальных контрольных точек является консистентным (но не строго консистентным).

Чтобы избежать потерь сообщений при восстановлении с использованием консистентного множества контрольных точек необходимо повторить отправку тех сообщений, квитанции о получении которых стали недействительными в результате отката. Используя временные метки сообщений можно распознавать сообщения-призраки и избежать бесконечного восстановления.

7.1.5. Синхронная фиксация контрольных точек и восстановление.

Ниже описываются алгоритмы создания консистентного множества контрольных точек и использования их для восстановления без опасности бесконечного зацикливания.

Алгоритм создания консистентного множества контрольных точек.

К распределенной системе алгоритм предъявляет следующие требования.

(1) Процессы взаимодействуют посредством посылки сообщений через коммуникационные каналы.

(2) Каналы работают по алгоритму FIFO. Коммуникационные протоколы точка-точка гарантируют невозможность пропажи сообщений из-за ошибок коммуникаций или отката к контрольной точке. (Другой способ обеспечения этого - использование стабильной памяти для журнала посылаемых сообщений и фиксации идентификатора последнего полученного по каналу сообщения).

Алгоритм создает в стабильной памяти два вида контрольных точек - постоянные и пробные.

Постоянная контрольная точка - это локальная контрольная точка, являющаяся частью консистентной глобальной контрольной точки. Пробная контрольная точка - это временная контрольная точка, которая становится постоянной только в случае успешного завершения алгоритма. Алгоритм исходит из того, что только один процесс инициирует создание множества контрольных точек, а

также из того, что никто из участников не сломается во время работы алгоритма.

Алгоритм выполняется в две фазы.

1-ая фаза.

Инициатор фиксации (процесс Pi) создает пробную контрольную точку и просит все остальные процессы сделать то же самое. При этом процессу запрещается посылать неслужебные сообщения после того, как он сделает пробную контрольную точку. Каждый процесс извещает Pi о том, сделал ли он пробную контрольную точку. Если все процессы сделали пробные контрольные точки, то Pi принимает решение о превращении пробных точек в постоянные. Если какой-либо процесс не смог сделать пробную точку, то принимается решение об отмене всех пробных точек.

2-ая фаза.

Pi информирует все процессы о своем решении. В результате либо все процессы будут иметь новые постоянные контрольные точки, либо ни один из процессов не создаст новой постоянной контрольной точки. Только после выполнения принятого процессом Pi решения все процессы могут посылать сообщения.

Корректность алгоритма очевидна, поскольку созданное всеми множество постоянных контрольных точек не может содержать не зафиксированных операций посылки сообщений.
Оптимизация: если процесс не посылал сообщения с момента фиксации предыдущей постоянной контрольной точки, то он может не создавать новую.

Алгоритм отката (восстановления).

Алгоритм предполагает, что его инициирует один процесс и он не будет выполняться параллельно с алгоритмом фиксации.

Выполняется в две фазы.

1-ая фаза.

Инициатор отката спрашивает остальных, готовы ли они откатываться. Когда все будут готовы к откату, то он принимает решение об откате.

2-ая фаза.

Pi сообщает всем о принятом решении. Получив это сообщение, каждый процесс поступает указанным образом. С момента ответа на опрос готовности и до получения принятого решения процессы не должны посылать сообщения (нельзя же посылать сообщение процессу, который уже мог успеть откатиться).

Оптимизация: если процесс не обменивался сообщениями с момента фиксации предыдущей постоянной контрольной точки, то он может к ней не откатываться.

7.1.5. Асинхронная фиксация контрольных точек и восстановление.

Синхронная фиксация упрощает восстановление, но связана с большими накладными расходами:

(1) Дополнительные служебные сообщения для реализации алгоритма.
(2) Синхронизационная задержка - нельзя посылать неслужебные сообщения во время работы алгоритма.

Если отказы редки, то указанные потери совсем не оправданы.

Фиксация может производиться асинхронно. В этом случае множество контрольных точек может быть неконсистентным. При откате происходит поиск подходящего консистентного множества путем поочередного отката каждого процесса в ту точку, в которой зафиксированы все посланные им и полученные другими сообщения (для ликвидации сообщений-сирот). Алгоритм опирается на наличие в стабильной памяти для каждого процесса журнала, отслеживающего номера посланных и полученных им сообщений, а также на некоторые предположения об организации взаимодействия процессов, необходимые для исключения «эффекта домино» (например, организация приложения по схеме сообщение-реакция-ответ).

7.2. Отказоустойчивость.

Изложенные выше методы восстановления после отказов для некоторых систем непригодны (управляющие системы, транзакции в on-line режиме) из-за прерывания нормального функционирования.

Чтобы избежать этих неприятностей, создают системы, устойчивые к отказам. Такие системы либо маскируют отказы, либо ведут себя в случае отказа заранее определенным образом (пример - изменения, вносимые транзакцией в базу данных, становятся невидимыми при отказе).

Два механизма широко используются при обеспечении отказоустойчивости - *протоколы голосования* и *протоколы принятия коллективного решения.*
Протоколы голосования служат для маскирования отказов (выбирается правильный результат, полученный всеми исправными исполнителями).

Протоколы принятия коллективного решения подразделяются на два класса. Во-первых, *протоколы принятия единого решения*, в которых все исполнители являются исправными и должны либо все принять, либо все не принять заранее предусмотренное решение. Примерами такого решения являются решение о завершении итерационного цикла при достижении всеми необходимой точности, решение о реакции на отказ (этот протокол уже знаком нам - он использовался для принятия решения об откате всех процессов к контрольным точкам). Во-вторых, *протоколы принятия согласованных решений* на основе полученных друг от друга данных. При этом необходимо всем исправным исполнителям получить достоверные данные от остальных исправных исполнителей, а данные от неисправных исполнителей проигнорировать.

Ключевой подход для обеспечения отказоустойчивости - избыточность (оборудования, процессов, данных).

7.2.1. **Использование режима «горячего резерва»** (второй пилот, резервное ПО).

Проблема переключения на резервный исполнитель.

7.2.2. Использование активного размножения.

Наглядный пример - тройное дублирование аппаратуры в бортовых компьютерах и голосование при принятии решения.

Другие примеры – размножение страниц в DSM и размножение файлов в распределенных файловых системах. При этом очень важным моментом является наличие механизма неделимых широковещательных рассылок сообщений (они должны приходить всем в одном порядке).

Алгоритмы голосования.

Общая схема использования голосования при размножении файлов может быть представлена следующим образом.

Файл может модифицироваться разными процессами только последовательно (при открытии файла на запись процесс-писатель будет ждать закрытия файла другим писателем или всеми читателями), а читаться всеми одновременно (протокол писателей-читателей). Все модификации файла нумеруются и каждая копия файла характеризуется номером версии – количеством ее модификаций. Каждой копии приписано некоторое количество голосов V_i. Пусть общее количество приписанных всем копиям голосов равно V. Определяется кворум записи V_w и кворум чтения V_r так, что

$V_w + V_r > V$

Для записи информации в файл писатель рассылает ее всем владельцам копий файла и должен получить V_w голосов от тех, кто успешно выполнил запись.

Для получения права на чтение читателю достаточно получить необходимое число голосов (V_r) от любых серверов. Кворум чтения выбран так, что хотя бы один из тех серверов, от которых получено разрешение, является владельцем текущей копии файла. За чтением информации из файла читатель может обратиться к любому владельцу текущей копии файла.

Описанная схема базируется на **статическом распределении голосов.** Различие в голосах, приписанных разным серверам, позволяет учесть их особенности (надежность, эффективность). Еще большую гибкость предоставляет метод **динамического перераспределения голосов.**

Для того, чтобы выход из строя некоторых серверов не привел к ситуации, когда невозможно получить кворум, применяется механизм **изменения состава голосующих.**

Протоколы принятия единого решения

Необходимо отметить, что в условиях отсутствия надежных коммуникаций (с ограниченным временем задержки) не может быть алгоритма достижения единого решения. Рассмотрим известную «проблему двух армий».

Армия зеленых численностью 5000 воинов располагается в долине.

Две армии синих численностью по 3000 воинов находятся далеко друг от друга в горах, окружающих долину. Если две армии синих одновременно атакуют зеленых, то они победят. Если же в сражение вступит только одна армия синих, то она будет полностью разбита.

Предположим, что командир 1-ой синей армии генерал Александр посылает (с посыльным) сообщение командиру 2-ой синей армии генералу Михаилу «Я имею план - давай атаковать завтра на рассвете». Посыльный возвращается к Александру с ответом Михаила - «Отличная идея, Саша. Увидимся завтра на рассвете». Александр приказывает воинам готовиться к атаке на рассвете.

Однако, чуть позже Александр вдруг осознает, что Михаил не знает о возвращении посыльного и поэтому может не отважиться на атаку. Тогда он отправляет посыльного к Михаилу чтобы подтвердить, что его (Михаила) сообщение получено Александром и атака должна состояться.

Посыльный прибыл к Михаилу, но теперь тот боится, что не зная о прибытии посыльного Александр может не решиться на атаку. И т.д. Ясно, что генералы никогда не достигнут согласия.

Предположим, что такой протокол согласия с конечным числом сообщений существует. Удалив избыточные последние сообщения, получим минимальный протокол. Самое последнее сообщение является существенным (поскольку протокол минимальный). Если это сообщение не дойдет по назначению, то войны не будет. Но тот, кто посылал это сообщение, не знает, дошло ли оно. Следовательно, он не может считать протокол завершенным и не может принять решение об атаке. Даже с надежными процессорами (генералами), принятие единого решения невозможно при ненадежных коммуникациях.

Теперь предположим, что коммуникации надежны, а процессоры нет.

Классический пример ***протокола принятия согласованных решений*** - задача «Византийских генералов».

В этой задаче армия зеленых находится в долине, а n синих генералов возглавляют свои армии, расположенные в горах. Связь осуществляется по телефону и является надежной, но из n генералов m являются предателями. Предатели активно пытаются воспрепятствовать согласию лояльных генералов.

Согласие в данном случае заключается в следующем. Каждый генерал знает, сколько воинов находится под его командой. Ставится цель, чтобы все лояльные генералы узнали численности всех лояльных армий, т.е. каждый из них получил один и тот же вектор длины n, в котором i-ый элемент либо содержит численность i-ой армии (если ее командир лоялен) либо не определен (если командир предатель).

Соответствующий рекурсивный алгоритм был предложен в 1982 г. (Lamport).

Проиллюстрируем его для случая n=4 и m=1. В этом случае алгоритм осуществляется в 4 шага.

1 шаг. Каждый генерал посылает всем остальным сообщение, в котором указывает численность своей армии. Лояльные генералы указывают истинное количество, а предатели могут указывать различные числа в разных сообщениях. Генерал-1 указал 1 (одна тысяча воинов), генерал-2 указал 2,

генерал-3 указал трем остальным генералам соответственно x,y,z, а генерал-4 указал 4.

2-ой шаг. Каждый формирует свой вектор из имеющейся информации.

Получается:

vect1 (1,2,x,4)

vect2 (1,2,y,4)

vect3 (1,2,3,4)

vect4 (1,2,z,4)

3-ий шаг. Каждый посылает свой вектор всем остальным (генерал-3 посылает опять произвольные значения).

Генералы получают следующие вектора:

g1	g2	g3	g4
(1,2,y,4)	(1,2,x,4)	(1,2,x,4)	(1,2,x,4)
(a,b,c,d)	(e,f,g,h)	(1,2,y,4)	(1,2,y,4)
(1,2,z.4)	(1,2,z.4)	(1,2,z.4)	(i,j,k.l)

4-ый шаг. Каждый генерал проверяет каждый элемент во всех полученных векторах. Если какое-то значение совпадает по меньшей мере в двух векторах, то оно помещается в результирующий вектор, иначе соответствующий элемент результирующего вектора помечается «неизвестен».

Все лояльные генералы получают один вектор (1,2,»неизвестен»,4) - согласие достигнуто.

Если рассмотреть случай n=3 и m=1, то согласие не будет достигнуто.

Lamport доказал, что в системе с m неверно работающими процессорами можно достичь согласия только при наличии 2m+1 верно работающих процессоров (более 2/3).

Другие авторы доказали, что в распределенной системе с асинхронными процессорами и неограниченными коммуникационными задержками согласие невозможно достичь даже при одном неработающем процессоре (даже если он не подает признаков жизни).

Применение алгоритма - надежная синхронизация часов.

Алгоритм надежных неделимых широковещательных рассылок сообщений.

Алгоритм выполняется в две фазы и предполагает наличие в каждом процессоре очередей для запоминания поступающих сообщений. В качестве уникального идентификатора сообщения используется его начальный приоритет - логическое время отправления, значение которого на разных процессорах различно.

1-ая фаза.

Процесс-отправитель посылает сообщение группе процессов (список их идентификаторов содержится в сообщении).

При получении этого сообщения процессы:

- Приписывают сообщению приоритет, помечают сообщение как «недоставленное» и буферизуют его. В качестве приоритета используется временная метка (текущее логическое время).
- Информируют отправителя о приписанном сообщению приоритете.

2-ая фаза.

При получении ответов от всех адресатов, отправитель:

- Выбирает из всех приписанных сообщению приоритетов максимальный и устанавливает его в качестве окончательного приоритета сообщения.
- Рассылает всем адресатам этот приоритет.

Получив окончательный приоритет, получатель:

a) Приписывает сообщению этот приоритет.
b) Помечает сообщение как «доставленное».
c) Упорядочивает все буферизованные сообщения по возрастанию их приписанных приоритетов.
d) Если первое сообщение в очереди отмечено как «доставленное», то оно будет обрабатываться как окончательно полученное.

Если получатель обнаружит, что он имеет сообщение с пометкой «недоставленное», отправитель которого сломался, то он для завершения выполнения протокола осуществляет следующие два шага в качестве координатора.

1. Опрашивает всех получателей о статусе этого сообщения.

Получатель может ответить одним из трех способов:

- Сообщение отмечено как «недоставленное» и ему приписан такой-то приоритет.
- Сообщение отмечено как «доставленное» и имеет такой-то окончательный приоритет.
- Он не получал это сообщение.

2. Получив все ответы координатор выполняет следующие действия:

- Если сообщение у какого-то получателя помечено как «доставленное», то его окончательный приоритет рассылается всем. (Получив это сообщение каждый процесс выполняет шаги фазы 2).
- Иначе координатор заново начинает весь протокол с фазы 1. (Повторная посылка сообщения с одинаковым приоритетом не должна вызывать коллизий).

Необходимо заметить, что алгоритм требует хранения начального и окончательного приоритетов даже для принятых и уже обработанных сообщений.

Hadoop

Введение в Hadoop и MapReduce.

HDFS

Данные

- 2006 год – 0.18 зеттабайт
- 2011 год – 1.8 зеттабайт (10^21)

Нью-Йорская фондовая биржа 1 терабайт данных в день

Internet Archive более 2 петабайт и растет со скоростью 20 терабайт в месяц

Большой адронный коллайдер до 15 петабайт данных в год

Хранение и анализ данных

- 1990 год – 1370 Мбайт, скорость передачи до 4.4 Мбайт/с, 5 минут
- 2010 год – несколько терабайт, скорость передачи 100 Мбайт/с, 2.5 часа

Используем 100 дисков, на каждом из которых хранится 1/100 часть данных => при параллельной работе дисков данные будут прочитаны за 2 минуты

Hadoop

- Система надежного общего хранения и анализа данных
  - HDFS обеспечивает хранение
  - MapReduce - анализ

Hadoop и РСУБД

- Скорость позиционирования улучшается медленнее скорости передачи данных
- Позиционирование – процесс перемещения считывающей головки к определенному месту диска для чтения или записи данных.
- Скорость позиционирования определяет задержку при выполнении дисковых операций, тогда как скорость передачи данных определяют пропускную способность канала взаимодействия с диском
- Если в схеме обращения к данным преобладают операции позиционирования, то чтение и запись больших частей набора данных займут больше времени, чем при потоковых операциях, выполняемых со скоростью передачи данных

Hadoop и РСУБД

Параметр	Традиционная РСУБД	MapReduce
Размер данных	Гигабайты	Петабайты
Доступ	Интерактивный и пакетный	Пакетный
Обновления	Многократное чтение и запись	Однократная запись, многократное чтение
Структура	Статическая схема	Динамическая схема
Целостность	Высокая	Низкая
Масштабирование	Нелинейное	Линейное

Что такое Hadoop

- Инфраструктура (framework) для параллельной обработки больших объемов данных (терабайты)
- Особенности:
  - Функциональное программирование
  - Автоматическое распараллеливание
  - Перемещение вычислений к данным
- Open Source, http://hadoop.apache.org

Концепции

- Парадигмы программирования:
  - Императивное программирование (ИП)
  - Функциональное программирование (ФП)
- Работа с данными:
  - Перемещение данных к вычислительным ресурсам (ПДкВ)
  - Перемещение вычислений к данным (ПВкД)

Концепции

- Наиболее популярная сейчас технология:
  - императивное программирование + перемещение данных к вычислениям
- Примеры:
  - MPI
  - GPU

GPU nVidia

- Высокая производительность:
  - nVidia Tesla M2050/2070 – 0,5 TFlops double
- Шаги вычислений:
  - Копирование данных в память GPU
  - Обработка данных в GPU
  - Копирование данных в память хоста
- Программист полностью управляет процессом вычислений и перемещения данных

Недостатки ИП + ПДкВ

- MPI и GPU эффективны при:
  - Небольших объемах данных
  - Высокой сложности вычислений
  - Небольшом количестве узлов (сотни)
- Терабайты данных перемещать долго
- Управлять логикой передачи данных на тысячи узлов сложно

Проблемы разработки для крупных параллельных систем

- Масштабирование на тысячи узлов
- Эффективное распределение нагрузки
- Эффективный обмен данными в процессе вычислений
- Обработка отказов вычислительных узлов
- Императивное программирование:
  - Все эти задачи программист должен решать сам

Функциональное программирование

- Программист описывает функцию, которую надо вычислить, но не процесс вычисления
- Входные данные не изменяются, создаются новые
- Поток данных жестко встроен в программу

Задачи Hadoop/MapReduce

- Эффективная обработка терабайтов данных
- Автоматическое распараллеливание на тысячи узлов
- Автоматическое распределение нагрузки
- Автоматическая обработка отказов оборудования
- Простота использования

Примеры приложений

- Распределенный grep
- Распределенная сортировка
- Инвертированный индекс
- Подсчет количества запросов к URL
- Реверсивный web-link граф

История

- Google:
  - 2003 - The Google File System
  - 2004 - MapReduce: Simplified Data Processing on Large Clusters
- 2005 - Open Source поисковик Apache Nutch использует MapReduce
- 2006 – Open Source реализация MapReduce выделяется в отдельный проект Apache Hadoop

Кто использует Hadoop

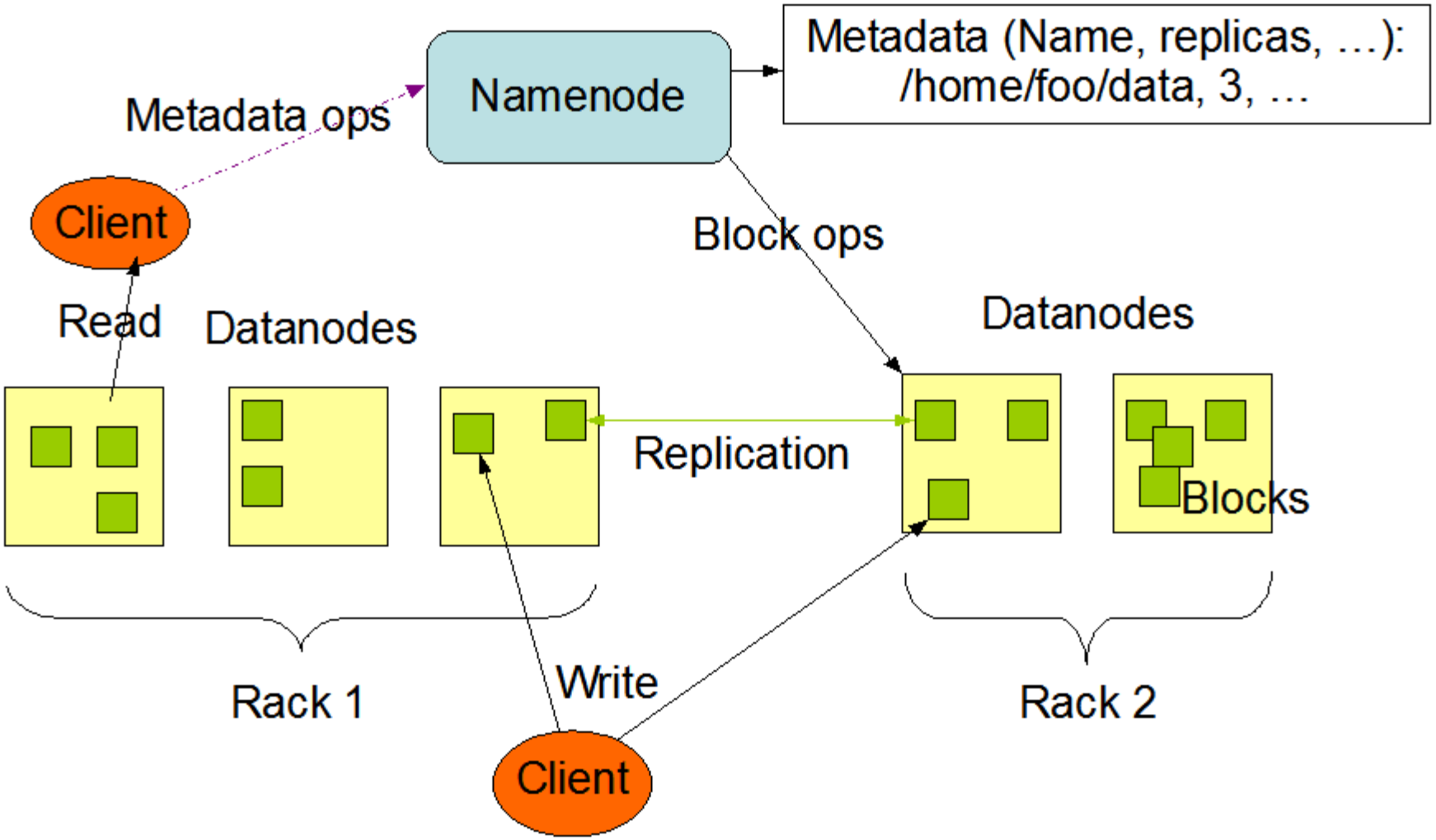
Состав Hadoop

- Hadoop Common – общие компоненты Hadoop
- Hadoop HDFS – распределенная файловая система
- Hadoop MapReduce – реализация MapReduce на Java

Hadoop HDFS

- Специализированная распределенная файловая система для хранения Терабайтов данных
- Цели разработки:
  - Надежное хранение данных на дешевом ненадежном оборудовании
  - Высокая пропускная способность ввода-вывода
  - Потоковый доступ к данным
  - Упрощенная модель согласованности: WORM
- Архитектура аналогична Google File System

Архитектура Hadoop HDFS



Архитектура HDFS

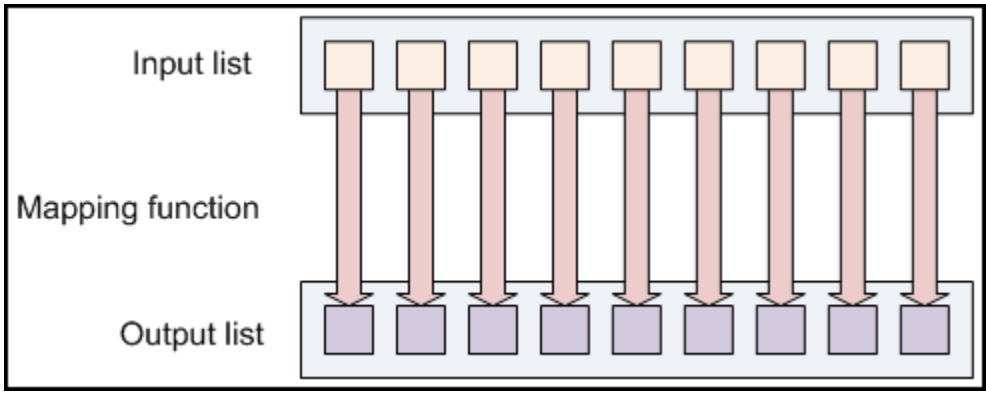
- Узлы хранения – серверы стандартной архитектуры
- Данные хранятся на внутренних дисках серверов
- Единое адресное пространство
- Параллельное чтение и запись на узлы – высокая пропускная способность

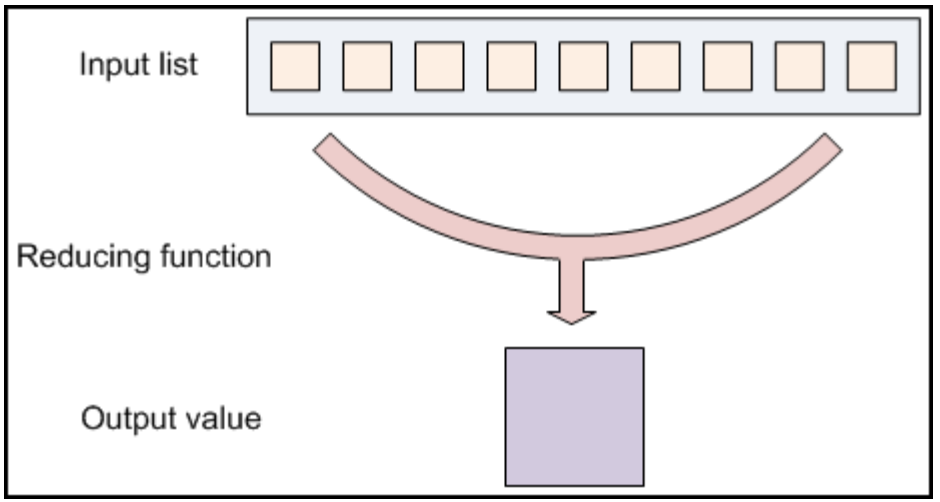
MapReduce

- Программная модель параллельной обработки **больших объемов данных** за путем разделения на **независимые** задачи
- MapReduce разработан в Google для поисковой системы
- Использует функциональное программирование, обработку списков

Функции MAP и Reduce

- Названия заимствованы из функциональных языков (LISP, ML)
- Обработка списков





MapReduce в Hadoop

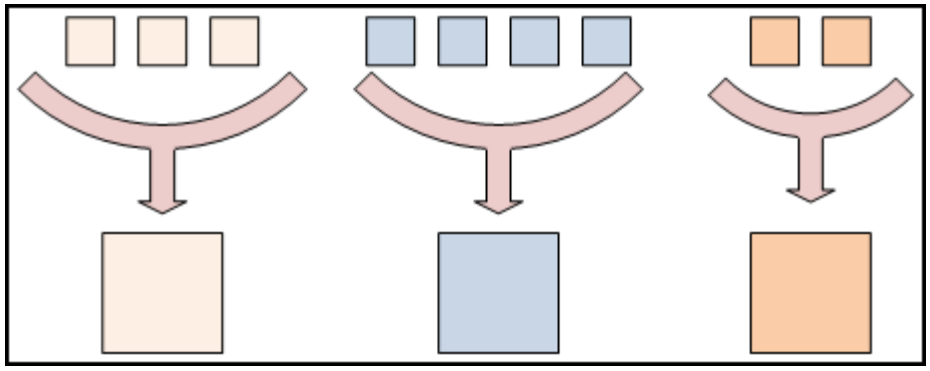
- Списки пар: ключ-значение

  AAA-123 65mph, 12:00pm

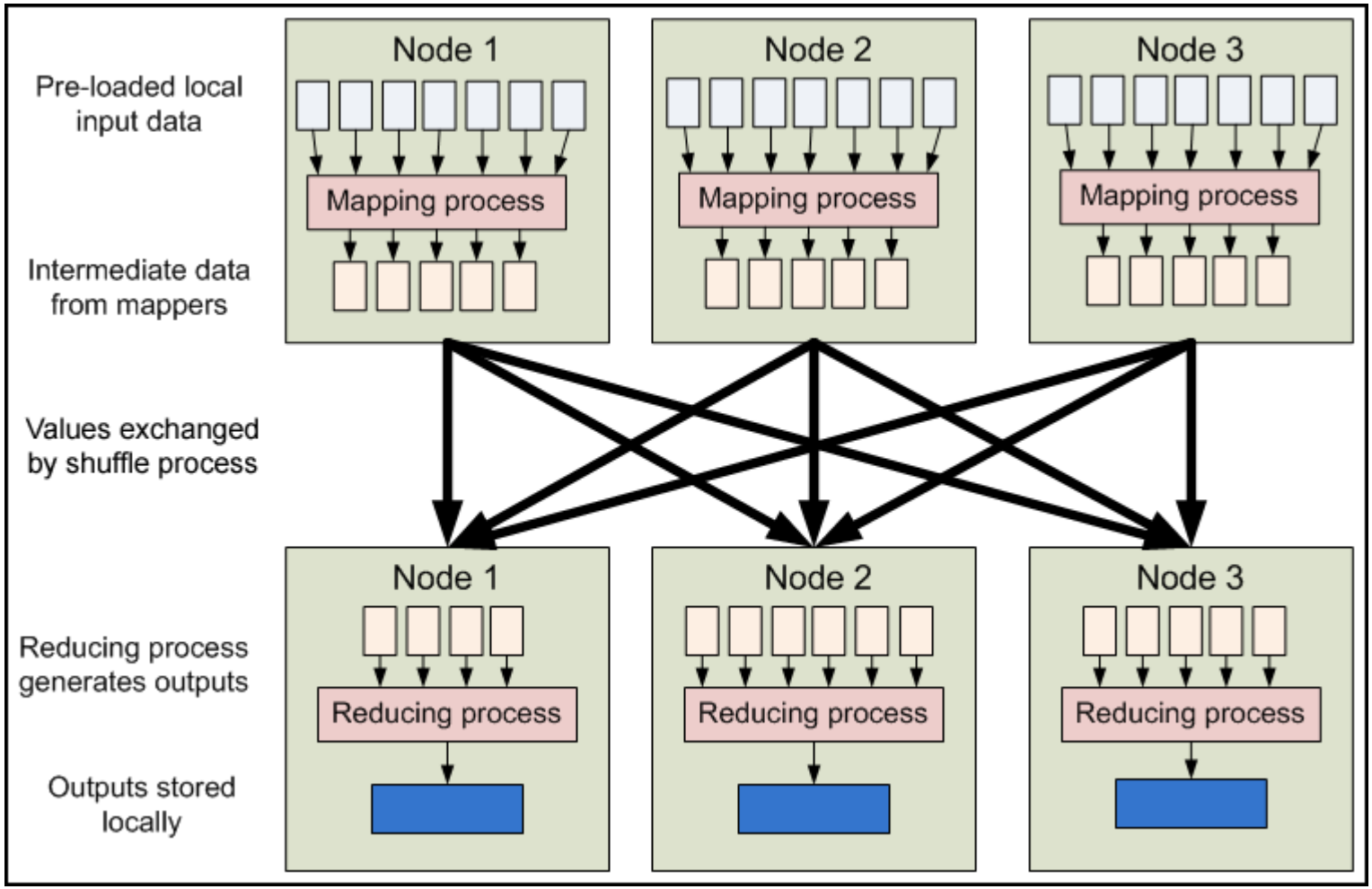
  ZZZ-789 50mph, 12:02pm

  AAA-123 40mph, 12:05pm

- Reduce выполняется отдельно для разных ключей

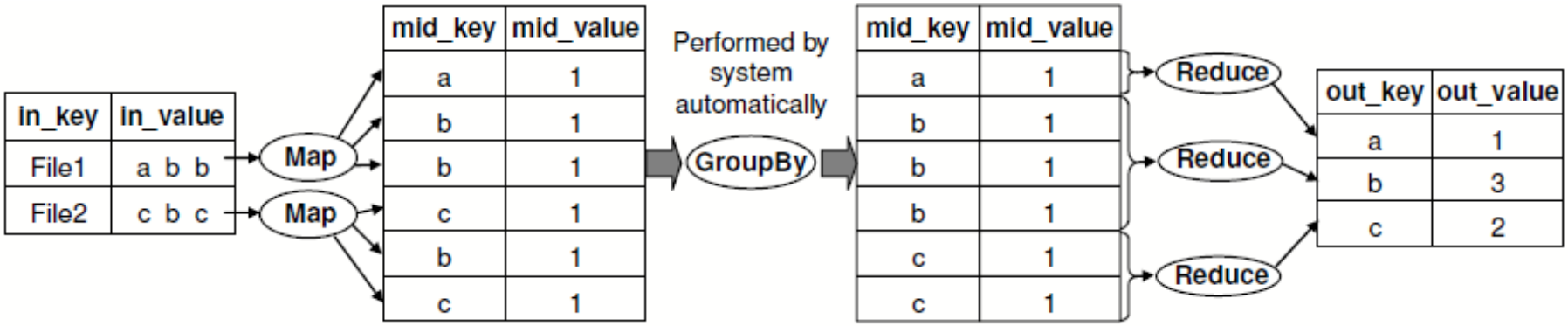


Поток данных MapReduce

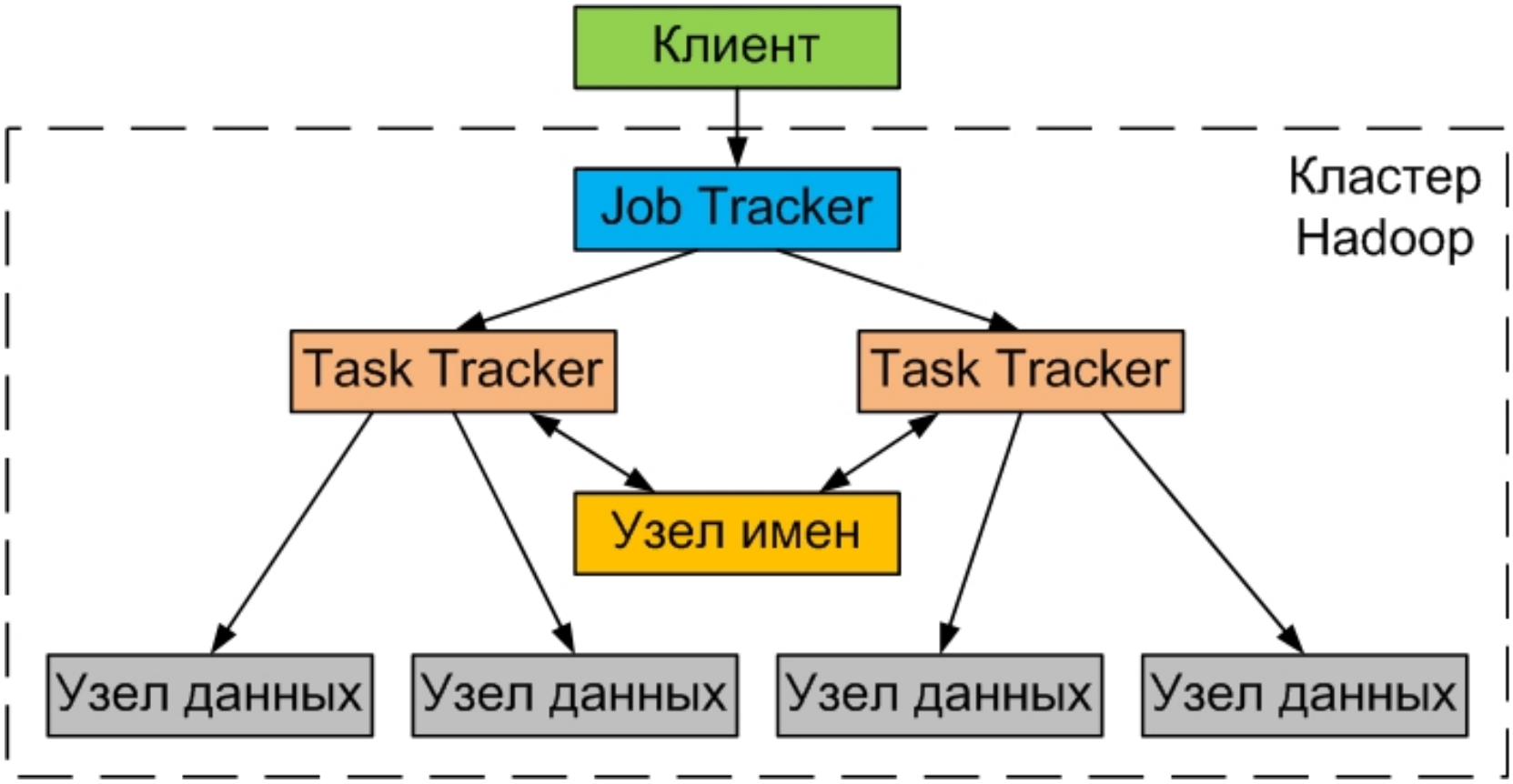


Пример WordCount

- Подсчет количества слов в файлах



Архитектура Hadoop



Перемещение вычислений к данным

- Задача запускается на том узле хранения, который содержит данные для обработки (фаза MAP)
- Перемещаются только входные списки для Reduce, их объем мал (как правило)

Результаты Hadoop в TeraSort

Байты	Узлы	Maps	Reduces	Время
$5*10^{11}$	1 406	8 000	2 600	59 секунд
10^{12}	1 460	8 000	2 700	62 секунды
10^{14}	3 452	190 000	10 000	173 минуты
10^{15}	3 658	80 000	20 000	975 минут

Источник: Owen O'Malley and Arun C. Murth. Winning a 60 Second Dash with a Yellow Elephant.

ОС и режимы работы

- Java 6
- Поддерживаемые ОС:
  - Linux (продуктив)
  - Windows (только тестирование)
  - Любой UNIX (не гарантируется)
- Режимы работы:
  - Локальный
  - Псевдо-распределенный
  - Распределенный

Программирование Hadoop

- Java API
- Hadoop Plugin для Eclipse
- Hadoop Streaming - другие языки:
  - Shell
  - Python
  - Ruby
  - и др.

Системы на основе Hadoop

- Pig – высокоуровневый язык потоков данных
- HBase – распределенная база данных
- Cassandra – multi-master база данных без единой точки отказа
- Hive – хранилище данных (warehouse)
- Mahout – машинное обучение и извлечение знаний

Распределенная файловая система HDFS

- Мотивация использования распределенных файловых систем
- Архитектура HDFS
- Команды работы с HDFS
- Права доступа в HDFS
- Работа с HDFS из Java программ

Мотивация

- Что нужно для эффективной обработки терабайтов данных?
  - Большая емкость
  - Высокая производительная
  - Надежность

Традиционное решение

- Системы хранения данных
  - Емкость: сотни и тысячи дисков
  - Производительность: сотни ГБ/с
  - Надежность: RAID, дублирование компонентов, репликация
- Примеры: EMC Symmetrix VMAX, Hitachi VSP, HP XP20000
- Недостаток: высокая стоимость (миллионы долларов)

Распределенные файловые системы

- Можно ли получить емкость, производительность и надежность дешево?
- Да, можно. Google:
  - “The Google File System”, Sanjay Ghemawat, Howard Gobioff, Shun-Tak Leung. *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, 2003, pp. 20-43.
  - Для хранения данных используются диски недорогих обычных серверов
  - Независимые диски объединяются в распределенную файловую систему GFS

Преимущества распределенных файловых систем

- Высокая емкость:
  - Много серверов с внутренними дисками
- Высокая производительность:
  - Параллельная запись на диски, много сетевых интерфейсов
- Высокая надежность:
  - Репликация данных на разные серверы
- Низкая стоимость:
  - Серверы стандартной архитектуры с Linux

HDFS

- Hadoop Distributed File System (HDFS) – распределенная файловая система, входящая в состав Hadoop
- Основывается на архитектуре Google File System
- HDFS - специализированная файловая система для приложений Hadoop

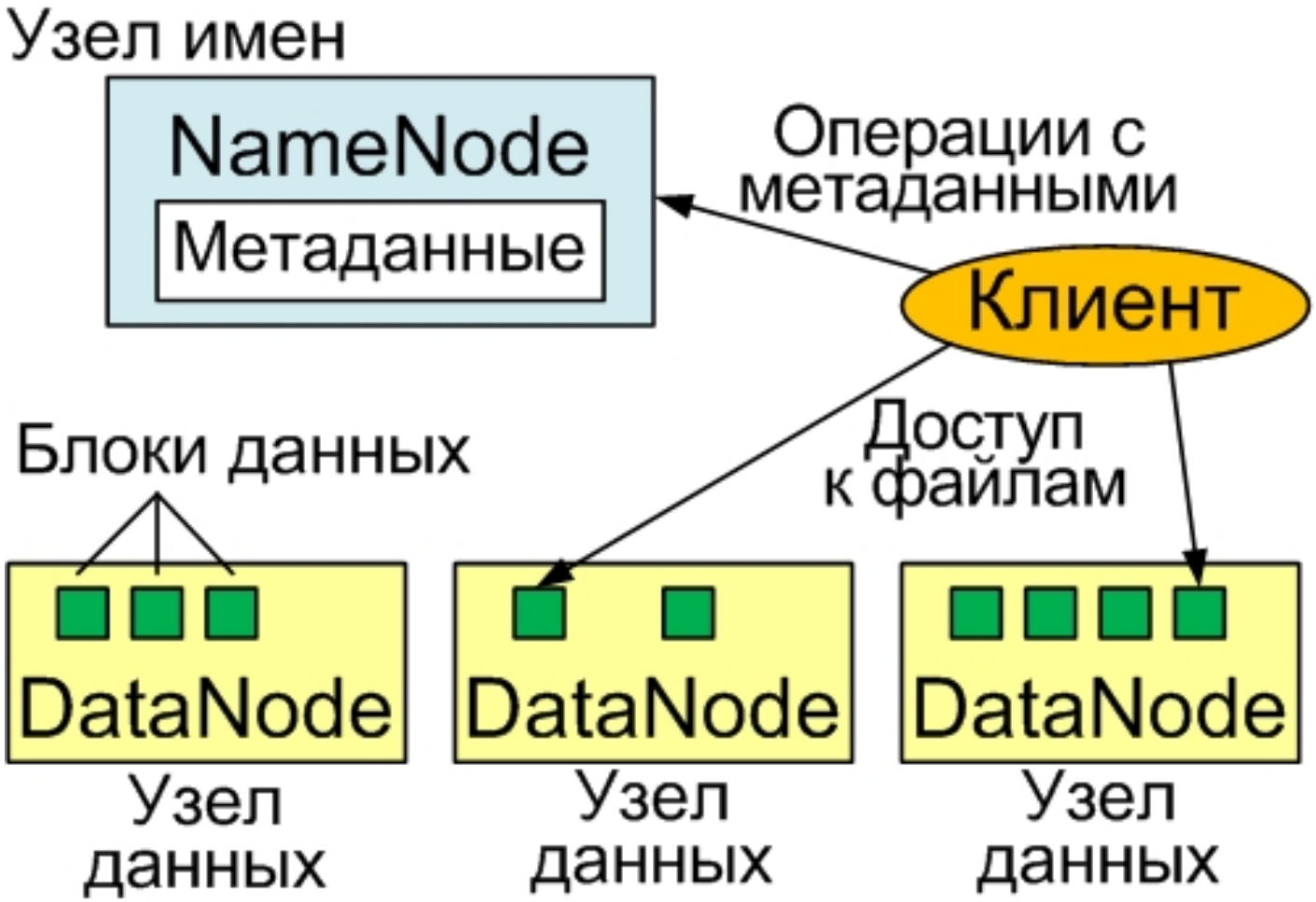
Потребности приложений Hadoop

- Типовое приложение – поисковый робот
  - Файлы индексов содержимого Web больших размеров
  - Файлы индексов записываются один раз, а затем только читаются (без изменений)
  - Потоковые операции ввода-вывода
  - Пакетная обработка

Ограничения HDFS

- Оптимизация для потоковых операций с большими файлами
  - Случайный доступ работает медленно
- Модель доступа к файлам WORM (Write-Once-Read-Many)
  - Запись в файл производиться только один раз, потом только чтение
- Не поддерживается POSIX
  - Нельзя подмонтировать, не работают стандартные Linux команды ls, cp, mkdir и т.п.
- Кэширование не используется
  - Накладные расходы слишком велики

Архитектура HDFS



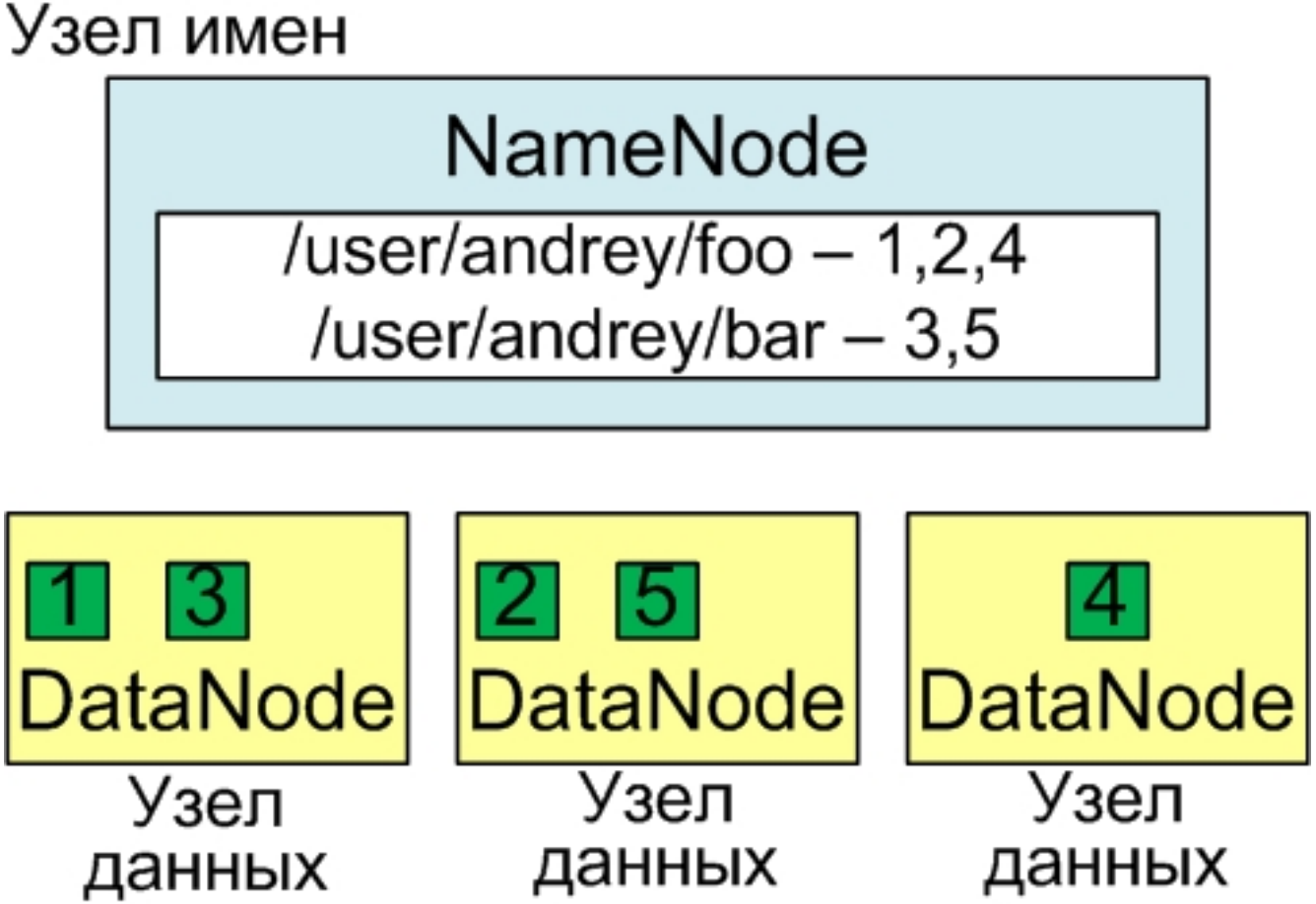
Архитектура HDFS

- Namenode (узел имен):
  - Управляющий узел
  - Обеспечивает единое пространство имен
  - Регулирует доступ клиентов
  - Хранит метаданные
- Datanode (узел данных)
  - Хранит данные
- Узлы имен и данных – серверы Linux (как правило)

Хранение файлов в HDFS

- Блочная структура:
  - Файл разбивается на блоки одинакового размера (64МБ по умолчанию)
  - Блоки хранятся на одном или нескольких узлах хранения
  - Возможна репликация блоков
- Узел имен хранит метаданные о распределении блоков по узлам хранения

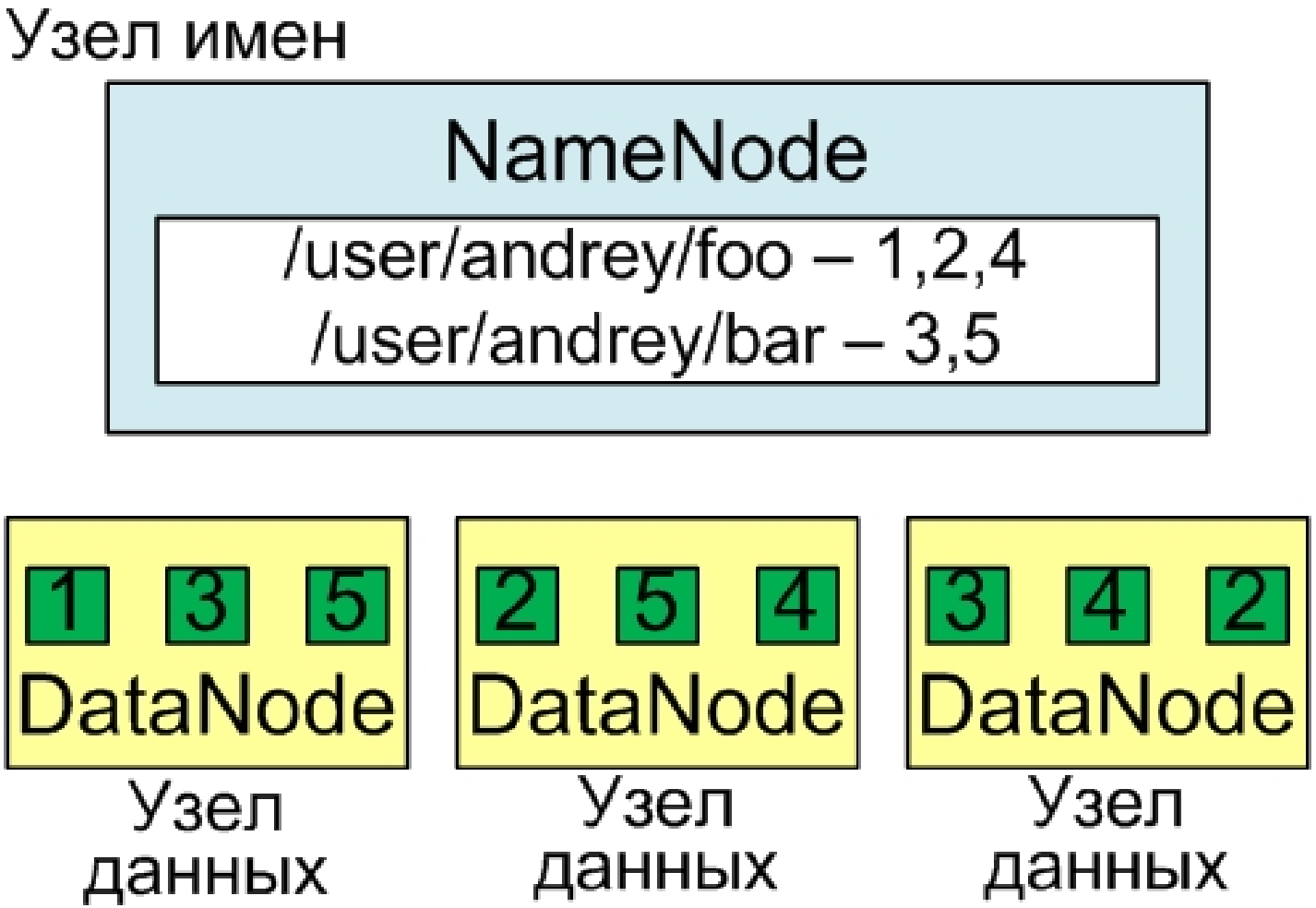
Хранение файлов в HDFS



Репликация

- В большом кластере всегда будут неисправные узлы
- Для защиты от сбоев HDFS использует репликацию – хранение нескольких копий блока
- Фактор репликации – количество копий блока (3 шт. по умолчанию)
- Отказ сервера снижает производительность, но не приводит к потере данных

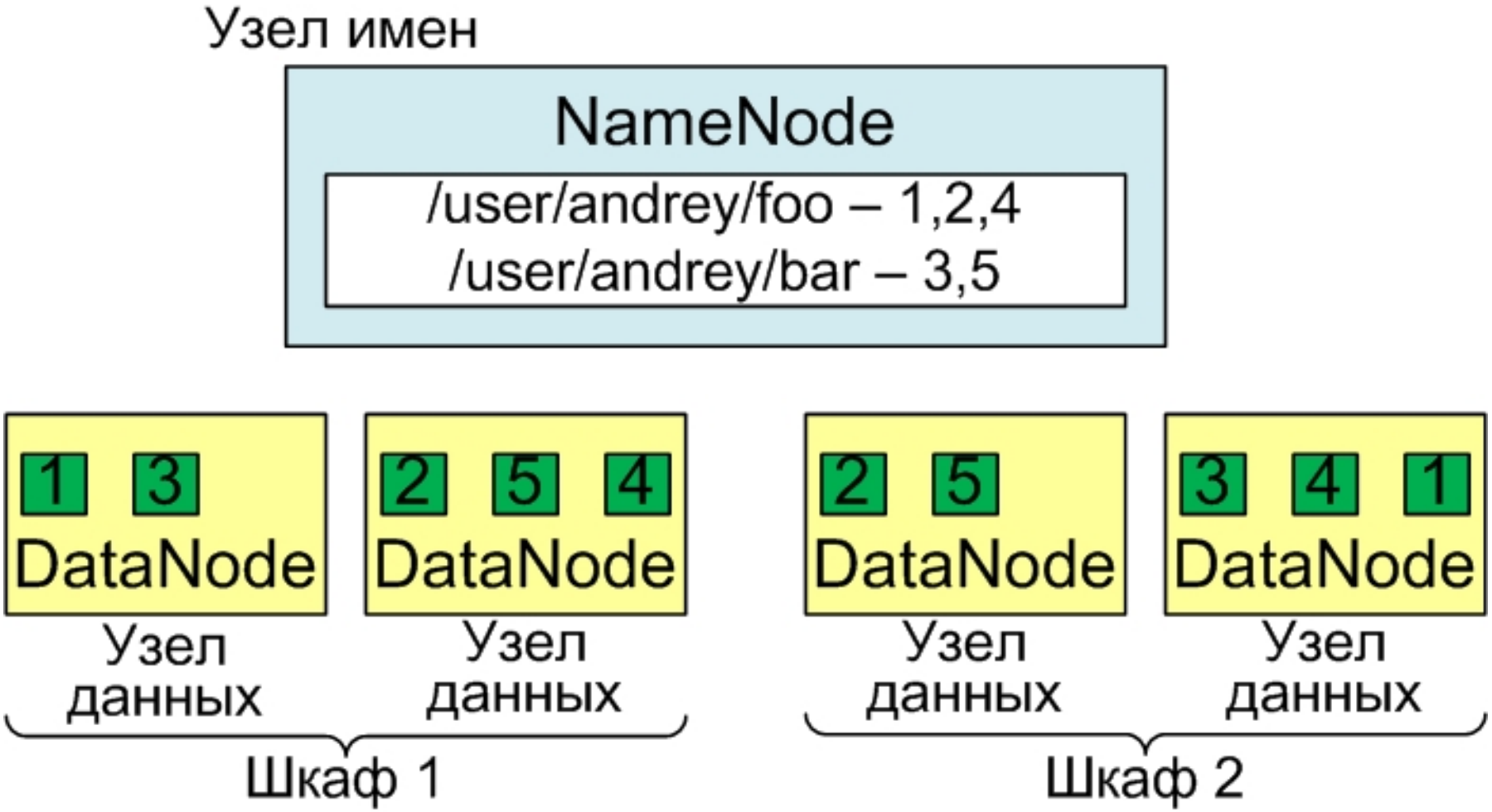
Репликация


Узел имен
NameNode
/user/andrey/foo – 1,2,4
/user/andrey/bar – 3,5
1 3 5
DataNode
Узел данных
2 5 4
DataNode
Узел данных
3 4 2
DataNode
Узел данных

Rack Awareness

- Дополнительный механизм защиты от сбоя группы серверов
- Rack – серверный шкаф:
  - Отключение питания всего шкафа
  - Потеря сетевого соединения со шкафом
- HDFS умеет распределять реплики между разными шкафами
- Отказ всего шкафа не приводит к потере данных

Rack Awareness



Доступ к HDFS

- Блоки HDFS распределены по разным серверам:
  - HDFS нельзя подмонтировать
  - Не работают стандартные Linux команды: ls, cp, mv и .т.
- Для работы с HDFS используются специальные команды:
  - `$bin/hadoop dfs -cmd`

Структура HDFS

- Корневой каталог HDFS - /
- Домашние каталоги пользователей - /user/$USER
- Временный каталог - /tmp
- Нет понятия текущий каталог
  - Нет команд cd, pwd
- Пути:
  - Полные – начиная с /
  - Относительные – из домашнего каталога пользователя

Просмотр файлов в каталоге

- Домашний каталог:

```
hadoop@hadoop:~/hadoop$ bin/hadoop dfs -ls
Found 3 items
-rw-r--r--   1 hadoop supergroup    0 2011-06-22 13:58 /user/hadoop/file1
-rw-r--r--   1 hadoop supergroup    0 2011-06-22 13:58 /user/hadoop/file2
-rw-r--r--   1 hadoop supergroup    0 2011-06-22 13:58 /user/hadoop/file3
```

- Корневой каталог:

```
hadoop@hadoop:~/hadoop$ bin/hadoop dfs -ls /
Found 2 items
drwxr-xr-x   - hadoop supergroup          0 2011-05-17 18:32 /tmp
drwxr-xr-x   - hadoop supergroup          0 2011-05-18 14:35 /user
```

Просмотр файла

- Список файлов:

```
hadoop@hadoop:~/hadoop$ bin/hadoop dfs -ls
Found 3 items
-rw-r--r--   1 hadoop supergroup    0 2011-06-22 13:58 /user/hadoop/file1
-rw-r--r--   1 hadoop supergroup    0 2011-06-22 13:58 /user/hadoop/file2
-rw-r--r--   1 hadoop supergroup    0 2011-06-22 13:58 /user/hadoop/file3
```

- Просмотр файла:

```
hadoop@hadoop:~/hadoop$ bin/hadoop dfs -cat file1
Hello, world!
Hello, Hadoop!
```

- Просмотр файла, полный путь:

```
hadoop@hadoop:~/hadoop$ bin/hadoop dfs -cat /user/hadoop/file1
Hello, world!
Hello, Hadoop!
```

Запись файлов в HDFS

- Команда:
  - `$bin/hadoop dfs –put` *`localSrc hdfsDest`*
- Копирует из локальной файловой системы в HDFS
- Работает как с файлами, так и с каталогами
- Если файл уже существует, выдает ошибку
- Синоним -copyFromLocal

Запись файлов в HDFS

- Запись файла:

```
hadoop@hadoop:~/hadoop$ bin/hadoop dfs -put /tmp/file1 /user/hadoop
```

- Запись каталога:

```
hadoop@hadoop:~/hadoop$ bin/hadoop dfs -put /tmp/dir1 /user/hadoop
hadoop@hadoop:~/hadoop$ bin/hadoop dfs -ls dir1
Found 3 items
-rw-r--r--   1 hadoop supergroup  0 2011-06-22 14:49 /user/hadoop/dir1/filea
-rw-r--r--   1 hadoop supergroup  0 2011-06-22 14:49 /user/hadoop/dir1/fileb
-rw-r--r--   1 hadoop supergroup  0 2011-06-22 14:49 /user/hadoop/dir1/filec
```

- Файл существует:

```
hadoop@hadoop:~/hadoop$ bin/hadoop dfs -put /tmp/file1 file1
put: Target file1 already exists
```

Получение файлов из HDFS

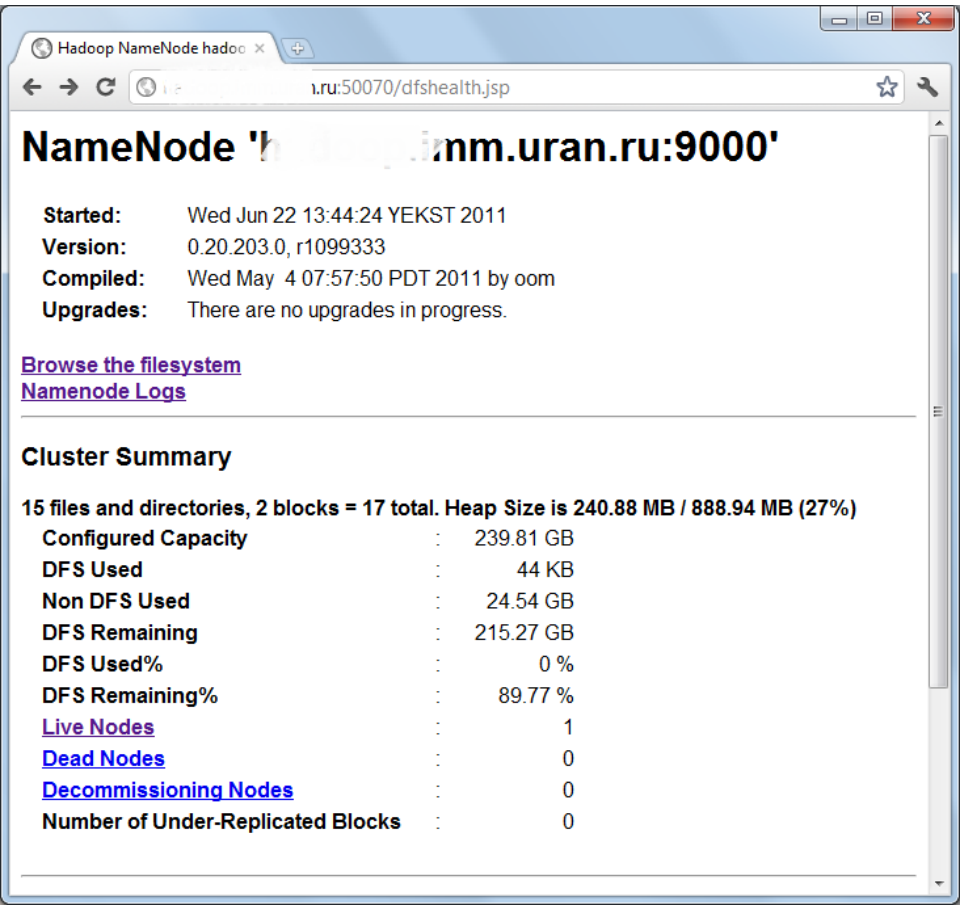
- Команда:
  - `$bin/hadoop dfs –get` *`hdfsSrc localDest`*
- Копирует из HDFS в локальную файловую систему
- Работает как с файлами, так и с каталогами
- Синоним -copyToLocal

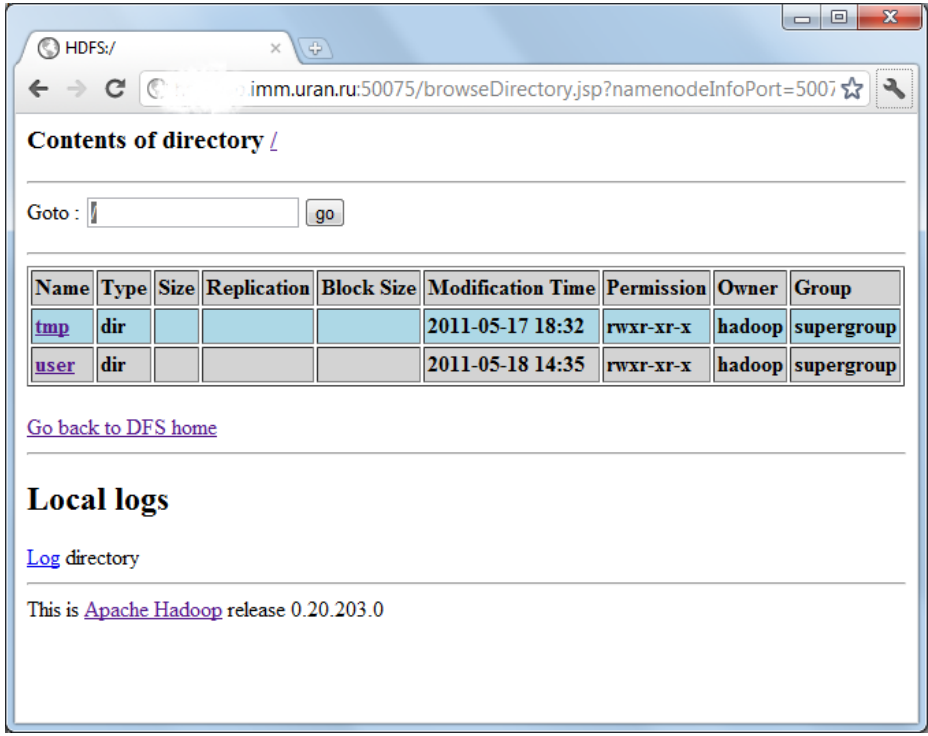
Команды для работы с HDFS

Команда	Назначение
`-mv` *`src dest`*	Перемещение файлов внутри HDFS
`-cp` *`src dest`*	Копирование файлов внутри HDFS
`-rm` *`path`*	Удаление файла или пустого каталога
`-rmr` *`path`*	Удаление файла или каталога рекурсивно
`-mkdir` *`path`*	Создание каталога (работает как mkdir –p в Linux)
`-stat` *`path`*	Выводит информацию по файлу или каталогу
`-tail [-f]` *`path`*	Вывод последнего килобайта файла (с -f выводит добавляемые данные)
`-help`	Перечень команд работы с HDFS

Web-интерфейс к HDFS

- http://namenode-hostname:50070





Права доступа в HDFS

- Модель прав доступа HDFS похожа на POSIX:
  - Файл имеет владельца (owner) и группу (group)
  - Права задаются отдельно для владельца, группы и всех остальных
  - Права доступа rwx
  - Нет sticky bit, setuid or setgid

Семантика прав доступа

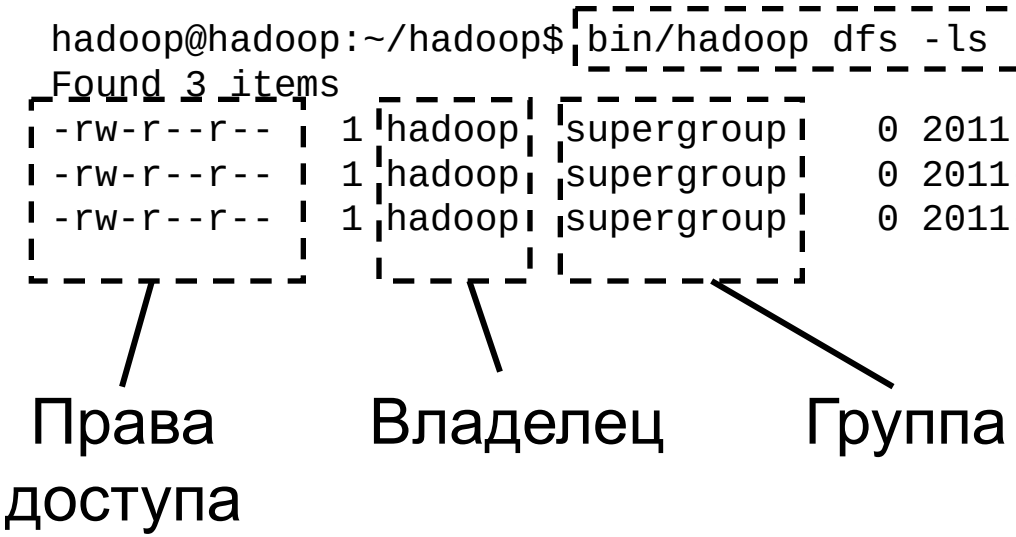
- Для файлов:
  - r – чтение
  - w – запись
  - x – не используется
- Для каталогов:
  - r – просмотр содержимого каталога
  - w – создание файлов или каталогов
  - x – доступ к файлам и подкаталогам

Пользователи HDFS

- Пользователи HDFS соответствуют пользователям Linux:
  - Пользователь: `whoami`
  - Список групп: `bash -c groups`
- Суперпользователь
  - Не действуют ограничения прав доступа
  - Пользователь, который запустил Hadoop
  - Нет постоянного суперпользователя

Просмотр прав доступа

```
hadoop@hadoop:~/hadoop$ bin/hadoop dfs -ls
Found 3 items
-rw-r--r--  1 hadoop supergroup   0 2011-06-22 13:58 /user/hadoop/file1
-rw-r--r--  1 hadoop supergroup   0 2011-06-22 13:58 /user/hadoop/file2
-rw-r--r--  1 hadoop supergroup   0 2011-06-22 13:58 /user/hadoop/file3
```



Управление правами доступа

- Команды

Команда	Назначение
`-chmod [-R] mode` *`path`*	Изменение прав доступа
`-chown [-R] owner[:group]` *`path`*	Изменение владельца (и группы)
`-chgroup [-R] grm` *`path`*	Изменение группы

- Опция *–R* – рекурсивные изменения
- Права доступа записываются как в Linux

Изменение прав доступа

- Исходные права доступа

```
$ bin/hadoop dfs -ls /user/Andrey/file1
-rw-r--r--   1 Andrey supergroup   0 2011-06-23 11:20 /user/Andrey/file1
```

- Цифровой режим

```
$ bin/hadoop dfs -chmod 600 /user/Andrey/file1

$ bin/hadoop dfs -ls /user/Andrey/file1
-rw-------   1 Andrey supergroup  0 2011-06-23 11:20 /user/Andrey/file1
```

- Символьный режим

```
$ bin/hadoop dfs -chmod g+rw /user/Andrey/file1

$ bin/hadoop dfs -ls /user/Andrey/file1
-rw-rw----   1 Andrey supergroup  0 2011-06-23 11:20 /user/Andrey/file1
```

Изменение владельца и группы

- Исходное состояние

```
$ bin/hadoop dfs -ls /user/Andrey/file1
-rw-rw----   1 Andrey supergroup          0 2011-06-23 11:20 /user/Andrey/file1
```

- Изменение владельца

```
$ bin/hadoop dfs -chown anton /user/Andrey/file1

$ bin/hadoop dfs -ls /user/Andrey/file1
-rw-rw----   1 anton supergroup          0 2011-06-23 11:20 /user/Andrey/file1
```

- Изменение группы

```
$ bin/hadoop dfs -chgrp project1 /user/Andrey/file1
$ bin/hadoop dfs -ls /user/Andrey/file1
-rw-rw----   1 anton project1          0 2011-06-23 11:20 /user/Andrey/file1
```

Работа с HDFS из Java

```
// Настройка путей
Configuration conf = new Configuration();
FileSystem fs = FileSystem.get(conf);
Path filenamePath = new Path("hello.txt");
// Запись в файл
FSDataOutputStream out = fs.create(filenamePath);
out.writeUTF(«Hello, world");
out.close();
// Чтение файла
FSDataInputStream in = fs.open(filenamePath);
String messageIn = in.readUTF();
System.out.print(messageIn);
in.close();
```

Подключение к файловой системе

- org.apache.hadoop.fs.FileSystem – интерфейс для работы с DFS и другими файловыми системами
- org.apache.hadoop.conf.Configuration – конфигурация Hadoop и HDFS
- Подключение к файловой системе:

```
Configuration conf = new Configuration();
FileSystem fs = FileSystem.get(conf);
```

Структура имени файла

- Файл в HDFS:
  - `hdfs://namenode:port/path/file`
  - `hdfs://localhost:9000/user/hadoop/file1`
  - `hdfs://namenode:port` – можно не указывать, тогда используется namenode из текущего конфигурационного файла Hadoop
- Файл на локальном диске:
  - `file://path/file`

Запуск программы

- FileSystem может работать как с HDFS, так и с локальной файловой системой, в зависимости от способа запуска
- Локальный запуск:
  - `java HDFSHelloWorld`
- Запуск через Hadoop (запись в HDFS):
  - `$bin/hadoop HDFSHelloWorld`

Методы FileSystem

Метод	Назначение
`copyFromLocalFile`	Копирование из локальной файловой системы в HDFS
`copyToLocalFile`	Копирование из HDFS в локальную файловую систему
`create`	Создание файла
`mkdirs`	Создание каталога
`delete`	Удаление файла или каталога
`rename`	Переименование файла
`setOwner`	Установка владельца и группы файла
`setPermissions`	Установка прав доступа к файлу
`getFileBlockLocations`	Возвращает список серверов, хранящих блоки файла

Дополнительные материалы

- The Google File System
  - http://labs.google.com/papers/gfs.html
- HDFS Architecture Guide
  - http://hadoop.apache.org/common/docs/current/hdfs_design.html
- HDFS Permissions Guide
  - http://hadoop.apache.org/common/docs/current/hdfs_permissions_guide.html
- HDFS Users Guide
  - http://hadoop.apache.org/common/docs/current/hdfs_user_guide.html

Вопросы?