

14. Сложность алгоритма как функция одного или нескольких числовых аргументов. Сложность в худшем случае.

Пусть по входным данным (входу) x алгоритм A вычисляет результат (выход) y . Такое вычисление связано с затратами времени и памяти. Допустим, что достигнуто соглашение о том, как измеряются эти затраты, тогда можно рассмотреть функции затрат

$$C_A^T(x), \quad C_A^S(x),$$

где верхний индекс T указывает на временные затраты¹, S — на пространственные затраты (затраты памяти и пространственные затраты — это синонимы), соответствующие вычислениям, связанным с применением A к входу x ; буквы T и S возникают из английских слов *time* — время и *space* — пространство. Вид этих функций может быть очень непростым; их трудно исследовать методами математического анализа, потому что аргумент x не является, вообще говоря, числом и не принадлежит какому-либо метрическому пространству. Можно ввести некоторую неотрицательную числовую характеристику $\|x\|$ возможных входов алгоритма и оценивать функции затрат с помощью функций, зависящих не от x , а от $\|x\|$:

$$C_A^T(x) \leq \varphi(\|x\|), \quad C_A^S(x) \leq \psi(\|x\|). \quad (1.1)$$

Избираемую нами величину $\|\cdot\|$ принято называть *размером входа*. В соответствии с нашим замыслом размер входа должен характеризовать «громоздкость» исходных данных; то, что $\|x\|$ является числом, позволяет говорить о росте φ, ψ и исследовать этот рост. Выбор размера входа — существенный этап оценивания функций затрат; нужна, во всяком случае, уверенность, что оценки вида (1.1) будут нести полезную информацию.

Определение 1.1. Пусть на возможных входах x алгоритма A определена неотрицательная числовая функция $\|x\|$ (размер входа). Пусть также определены целочисленные неотрицательные функции $C_A^T(x)$, $C_A^S(x)$ временных и пространственных затрат алгоритма A . Тогда *временной и пространственной сложностями* A называются функции числового аргумента

$$T_A(n) = \max_{\|x\|=n} C_A^T(x), \quad S_A(n) = \max_{\|x\|=n} C_A^S(x) \quad (1.2)$$

(областью изменения n как аргумента функций $T_A(n)$ и $S_A(n)$ является множество значений размера $\|\cdot\|$). Более полно каждая такая сложность именуется *сложностью в худшем случае*.

Вместо $T_A(n), S_A(n)$ мы иногда будем писать просто $T(n), S(n)$, если ясно, о каком алгоритме идет речь.

Два примечания к определению 1.1.

1) Значение $C_A^T(x)$ есть в подавляющем большинстве случаев число каких-то операций, а $C_A^S(x)$ — число хранимых величин какого-то типа, поэтому предположение о целочисленности значений этих функций вполне естественно. Без этого предположения в (1.2) пришлось

бы использовать $\sup$ вместо $\max$, что сделало бы многие рассуждения о сложности более тяжеловесными.

2) Видимо, вместо «сложность в худшем случае» правильнее было бы сказать «сложность как затраты в худшем случае», но такова принятая здесь терминология. Мы пока (до § 5) не рассматриваем сложность в среднем, и поэтому, не опасаясь путаницы, будем называть сложность в худшем случае просто сложностью.

Определение 1.2. Пусть функция $C_A^T(x)$ в определении 1.1 отражает затраты на выполнение лишь какого-то одного типа операций в предположении, что все эти операции требуют одинакового времени выполнения, не зависящего от вида и размера операндов, и это время равно 1. Тогда соответствующая функция $T_A(n)$ — это сложность *по числу операций* рассматриваемого типа. Привлечение в качестве $C_A^S(x)$ функции, отражающей только количество хранимых дополнительных величин того или иного фиксированного типа, приводит к пространственной сложности $S_A(n)$ *по числу величин* рассматриваемого типа.

Такую сложность называют *алгебраической* сложностью по числу операций или числу величин рассматриваемого типа, подчерки-

вая этим предположение о равнозатратности всех рассматриваемых операций и равнообъемности всех учитываемых величин. Алгебраическую сложность арифметических алгоритмов по числу умножений и делений называют *мультипликативной* сложностью. Из основных арифметических операций умножение и деление являются наиболее дорогими, и при рассмотрении арифметических алгоритмов часто интересуются именно (алгебраической) мультипликативной сложностью.

Сложность на входе и на выходе.

Формулы (1.2) вновь заставляют вспомнить о важности понятия размера входа. В силу разных причин не всегда этот размер легко и естественно может быть введен так, что сложность алгоритма возрастает при увеличении размера входа. Но, по крайней мере, функция $\|\cdot\|$ должна обеспечивать определенность значений $T_A(n), S_A(n)$ для всех достаточно больших n — иначе было бы невозможно говорить о поведении $T_A(n), S_A(n)$ при $n \rightarrow \infty$.

Достаточно очевидно, что если алгоритм A состоит в последовательном (друг за другом) выполнении алгоритмов U и V , то мы, вообще говоря, не можем утверждать, что $T_A(n) = T_U(n) + T_V(n)$, так как, например, размер входа алгоритма U может существенно отличаться, как в большую, так и в меньшую сторону, от размера его выхода.

Пусть A и B — некоторые алгоритмы. Как можно заметить, из выполнения для всех n неравенства $T_A(n) < T_B(n)$ не следует, что $C_A^T(x) < C_B^T(x)$ для всех возможных входов x (достаточно вспомнить сортировку слиянием и пузырьковую сортировку; последняя выполняется быстрее первой, коль скоро массив задан уже в упорядоченном виде). Аналогично дело обстоит с пространственными сложностью и затратами.

Пусть некоторому алгоритму A сопоставлены две, скажем, временные сложности $T'_A(n)$ и $T''_A(n)$. Например, $T'_A(n)$ и $T''_A(n)$ могут быть сложностями по разным операциям. Если операции первого и второго вида предполагаются имеющими одинаковую затратность, то это еще не дает оснований считать, что сложность $\tilde{T}_A(n)$, которую мы определим, рассматривая совместно операции обоих видов, будет равна $T'_A(n) + T''_A(n)$, потому что максимумы этих двух видов затрат могут достигаться на разных входах одного и того же размера. Можно только утверждать, что

$$\tilde{T}_A(n) \leq T'_A(n) + T''_A(n).$$

Определение 2.1. Функции $f(n)$ и $g(n)$ имеют одинаковый порядок (пишут $f(n) = \Theta(g(n))$) тогда и только тогда, когда найдутся положительные c_1, c_2, N такие, что неравенства

$$c_1|g(n)| \leq |f(n)| \leq c_2|g(n)| \quad (2.2)$$

выполнены для всех $n > N$.

Определение 2.2. Соотношение $f(n) = \Omega(g(n))$ имеет место тогда и только тогда, когда найдутся положительные c, N такие, что для всех $n > N$ выполнено $|f(n)| \geq c|g(n)|$.

Сильное утверждение, чем $f(n) = O(g(n))$, так как $f(n) = O(g(n))$ является лишь асимптотической верхней оценкой: в соответствии с известным из математического анализа определением

$$f(n) = O(g(n)) \iff \exists_{c, N > 0} \forall_{n > N} |f(n)| \leq c|g(n)|,$$

Предложение 2.1. Соотношение $f(n) = \Theta(g(n))$ имеет место тогда и только тогда, когда одновременно $f(n) = O(g(n))$ и $f(n) = \Omega(g(n))$; помимо этого, $f(n) = \Omega(g(n))$ тогда и только тогда, когда $g(n) = O(f(n))$.

Если размер входа является целым положительным числом, то возникающие функции являются последовательностями. Для единообразия мы, как правило, будем говорить о функциях, подразумевая, но не упоминая специально, что каждая такая функция определена лишь для целых положительных значений аргумента (возможно даже, только для достаточно больших целых положительных значений аргумента). Итак, при $n \rightarrow \infty$ оценки вида $f(n) = \Lambda(g(n))$, где Λ — один из символов Ω , O , Θ , предполагают, что функции $f(n), g(n)$ определены для всех достаточно больших n . Соответствующее неравенство из числа

$$|f(n)| \geq c_1 |g(n)|, \quad |f(n)| \leq c_2 |g(n)|, \quad c_1 |g(n)| \leq |f(n)| \leq c_2 |g(n)| \quad (2.3)$$

тоже, в соответствии с определением, должно выполняться лишь для n , больших некоторого N . Заметим, однако, что если $f(n)$ и $g(n)$ определены для всех $n \in \mathbb{N}^+$ и принимают при $1 \leq n \leq N$ ненулевые значения, то можно считать, что соответствующее неравенство из перечисленных в (2.3) выполняется для всех n , так как, положив

$$m = \min_{1 \leq n \leq N} \frac{|f(n)|}{|g(n)|}, \quad M = \max_{1 \leq n \leq N} \frac{|f(n)|}{|g(n)|},$$

мы можем заменить c_1, c_2 в (2.3) на $c'_1 = \min\{c_1, m\}$, $c'_2 = \min\{c_2, M\}$.

Определение 2.3. Если имеет место оценка $f(n) = O(g(n))$, то она называется *точной*, коль скоро существует неограниченно возрастающая последовательность неотрицательных целых чисел $\{n_k\}$ такая, что для $\varphi(k) = f(n_k)$, $\psi(k) = g(n_k)$ имеет место $\varphi(k) = \Omega(\psi(k))$.

Для упомянутых $\varphi(k)$ и $\psi(k)$ в силу этого определения и семантики символа Θ выполнено $\varphi(k) = \Theta(\psi(k))$.

Предложение 2.2. Пусть $f(n) = O(g(n))$. Эта оценка является *точной*, если и только если неверно, что $f(n) = o(g(n))$.

Доказательство. Пусть оценка является *точной*, и $\{n_k\}$ — возрастающая последовательность, о которой говорится в определении 2.3. Тогда существует положительная константа c такая, что $|f(n_k)| \geq c |g(n_k)|$, $k = 1, 2, \dots$, и соотношение $f(n) = o(g(n))$ места не имеет. Обратно, если неверно, что $f(n) = o(g(n))$, то по определению символа o существуют $\varepsilon > 0$ и возрастающая последовательность $\{n_k\}$ натуральных чисел такие, что $|f(n_k)| \geq \varepsilon |g(n_k)|$, $k = 1, 2, \dots$. Если при

этом выполнена оценка $f(n) = O(g(n))$, то эта оценка точна в соответствии с определением 2.3. $\square$

Часто, хотя и не всегда, для алгоритмов целочисленной арифметики, входом которых является целое неотрицательное число n , размером входа выбирают не само n , а его *битовую длину*, или, иными словами, количество $\lambda(n)$ цифр в его двоичной записи:

$$\lambda(n) = \begin{cases} 1, & \text{если } n = 0, \\ \lfloor \log_2 n \rfloor + 1, & \text{если } n > 0. \end{cases}$$

Наряду с его уже рассмотренной сложностью $T_A(n)$ введем еще одну сложность $T_A^*(m)$, $m = \lambda(n)$.

Лемма 4.1. Пусть $f(x)$ — неубывающая функция вещественной переменной. Тогда если $T_A^*(m) \leq f(m)$, то $T_A(n) \leq f(\log_2 n + 1)$.

Доказательство. Пусть m и n фиксированы, $2^{m-1} \leq n < 2^m$, и пусть значение $\hat{n}$ таково, что

$$2^{m-1} \leq \hat{n} < 2^m, \quad (4.3)$$

и при этом

$$T_A(\hat{n}) = T_A^*(m). \quad (4.4)$$

Используем неубывание $f(x)$:

$$T_A(n) \leq T_A(\hat{n}) = T_A^*(m) \leq f(m) = f(\lfloor \log_2 n \rfloor + 1) \leq f(\log_2 n + 1). \quad \square$$

Лемма 4.2. Пусть $g(x)$ — неубывающая функция вещественной переменной. Тогда

- (i) если $T_A(n) \leq g(n)$, то $T_A^*(m) \leq g(2^m)$,
- (ii) если $T_A(n) \geq g(n)$, то $T_A^*(m) \geq g(2^{m-1})$.

Доказательство. Пусть m фиксировано, $2^{m-1} \leq n < 2^m$, и пусть значение $\hat{n}$ таково, что выполнены (4.3) и (4.4). Используем неубывание $g(x)$:

- (i) $T_A^*(m) = T_A(\hat{n}) \leq g(\hat{n}) \leq g(2^m)$,
- (ii) $T_A^*(m) = T_A(\hat{n}) \geq g(\hat{n}) \geq g(2^{m-1})$. □

Утверждения следующих двух теорем непосредственно следуют из доказанных лемм. (При рассмотрении асимптотической оценки, содержащей некоторую переменную, подразумевается, что значение этой переменной стремится к бесконечности.)

Теорема 4.1. Пусть $f(x)$ — неубывающая функция вещественной переменной. Тогда если $T_A^*(m) = O(f(m))$, то $T_A(n) = O(f(\log_2 n + 1))$; как следствие, при $f(x+1) = O(f(x))$ имеем $T_A(n) = O(f(\log_2 n))$.

Теорема 4.2. Пусть $g(x)$ — неубывающая функция вещественной переменной. Тогда

(i) если $T_A(n) = O(g(n))$, то $T_A^*(m) = O(g(2^m))$,

(ii) если $T_A(n) = \Omega(g(n))$, то $T_A^*(m) = \Omega(g(2^{m-1}))$; как следствие, при $g\left(\frac{x}{2}\right) = \Omega(g(x))$ имеем $T_A^*(m) = \Omega(g(2^m))$.

15. Сложность в среднем. Сложность рандомизированного алгоритма.

До сих пор мы исследовали сложность в худшем случае, теперь обратимся к сложности в среднем. Мы ограничимся ситуацией, когда при каждом фиксированном значении s размера входа сами соответствующие входы образуют конечное множество $X_s = \{x: \|x\| = s\}$. Будем предполагать, что каждому $x \in X_s$ приписана некоторая вероятность $P_s(x)$:

$$0 \leq P_s(x) \leq 1,$$

и при этом

$$\sum_{x \in X_s} P_s(x) = 1.$$

Таким образом, при каждом допустимом значении s размера входа множество X_s превращается в вероятностное пространство; по умолчанию считается, что вероятность распределена равномерно:

$$P_s(x) = \frac{1}{N_s}, \quad (5.1)$$

где N_s — количество элементов множества X_s . Функции от s

$$\sum_{x \in X_s} C_A^T(x) P_s(x) \quad \text{и} \quad \sum_{x \in X_s} C_A^S(x) P_s(x) \quad (5.2)$$

называются временной и, соответственно, пространственной сложностью в среднем алгоритма A . Более подробно:

Определение 5.1. Пусть при любом допустимом значении s множество X_s всех входов размера s является вероятностным пространством, в силу чего временные и пространственные затраты алгоритма A для входов x (т. е. $C_A^T(x)$ и $C_A^S(x)$) размера s являются случайными величинами на X_s . Сложностью в среднем называется математическое ожидание (первая или вторая сумма из (5.2)) соответствующей случайной величины.

Сложность в среднем может принимать и нецелые значения, даже если функция затрат целочисленна.

Для временной и пространственной сложности в среднем алгоритма A мы будем использовать обозначения $\bar{T}_A(\cdot)$, $\bar{S}_A(\cdot)$.

Теорема 5.1. Для любого алгоритма A при любом распределении вероятностей на множестве X_s , где s — некоторое допустимое значение размера $\|\cdot\|$, сложность в среднем не превосходит сложности в худшем случае:

$$\bar{T}_A(s) \leq T_A(s), \quad \bar{S}_A(s) \leq S_A(s).$$

Доказательство. Докажем, например, первое неравенство:

$$\begin{aligned} \bar{T}_A(s) &= \sum_{x \in X_s} C_A^T(x) P_s(x) \leq \sum_{x \in X_s} (\max_{x \in X_s} C_A^T(x)) P_s(x) = \\ &= \sum_{x \in X_s} T_A(s) P_s(x) = T_A(s) \sum_{x \in X_s} P_s(x) = T_A(s). \end{aligned}$$

Второе неравенство доказывается аналогично. $\square$

Определение 5.2. Вещественная неотрицательная функция $f(m)$, определенная для целых положительных значений аргумента, называется *полиномиально ограниченной*, если существует полином $P(m)$ с вещественными коэффициентами такой, что $f(m) \leq P(m)$ для всех $m \in \mathbb{N}^+$. (Очевидно, что мы получим эквивалентное определение, если дополнительно потребуем, чтобы все коэффициенты полинома $P(m)$ были неотрицательными; другие эквивалентные варианты определения: $f(m) = O(m^d)$ для некоторого $d \in \mathbb{N}$ и $f(m) = m^{O(1)}$.)

второго варианта есть $\frac{n^2}{2} + O(n)$, поскольку для любых случайны величин ξ, η , определенных на каком-либо вероятностном пространстве, выполняется

$$E(\xi + \eta) = E\xi + E\eta. \quad (6.8)$$

Теорема 6.1. Сложность в среднем некоторого алгоритма по суммарному числу операций нескольких различных типов равна сумме сложностей в среднем этого алгоритма по числу операций каждого типа в отдельности.

Определение 8.1. Алгоритмы с элементами случайности, реализующими обращениями к генераторам случайных чисел, называются *рандомизированными*.

Рандомизированные алгоритмы можно разделить на вероятностные, или, что то же самое, алгоритмы типа *Монте-Карло*, и алгоритмы типа *Лас-Вегас*. Первый тип допускает, что ответ, который дает алгоритм для поставленной задачи с некоторым конкретным входом, может быть неправильным, хотя бы и с малой вероятностью; второй тип — Лас-Вегас — гарантирует правильный ответ, но (как и для алгоритмов типа Монте-Карло) время получения ответа для конкретного входа не определяется, вообще говоря, однозначно этим входом¹. Исключая специально оговариваемые редкие случаи, мы будем рассматривать только алгоритмы типа Лас-Вегас, не упоминая этого каждый раз.

Анализ сложности рандомизированного алгоритма сводится к нахождению математических ожиданий некоторых случайных величин. Но ситуация здесь отличается от той, когда множество входов фиксированного размера рассматривается как вероятностное пространство и затраты алгоритма, однозначно определенные для каждого конкретного входа, становятся случайными величинами на этом пространстве (мы шли этим путем в двух предыдущих параграфах). При исследовании рандомизированных алгоритмов вероятностное пространство, на котором рассматриваются случайные величины, состоит из *сценариев* выполнения алгоритма для фиксированного входа, и каждый сценарий определяется тем, какие случайные числа будут сгенерированы в соответствующие моменты выполнения алгоритма; за каждым таким сценарием закрепляется некоторая вероятность. В нашем контексте генератор случайных чисел можно представлять себе как стандартную функцию $random(N)$ целого положительного аргумента N , результатом выполнения которой является элемент множества $\{0, 1, \dots, N - 1\}$, но невозможно предсказать точно, каким именно будет значение этой функции, — любой из элементов указанного множества может появиться с вероятностью $1/N$.

Таким образом, затраты рандомизированного алгоритма при фиксированном входе, вообще говоря, не определяются однозначно, но зависят от сценария вычисления. При фиксированном входе мы можем рассмотреть множество всех сценариев и, приписав адекватным образом каждому из сценариев некоторую вероятность, ввести на полученном вероятностном пространстве случайную величину, значение которой для данного сценария равно соответствующим вычислительным затратам. Значение функции затрат на данном входе можно положить равным математическому ожиданию этой случайной величины (*усредненным* затратам для данного входа). А после того как определена функция затрат и принято соглашение о том, что такое размер входа, мы можем, как обычно, рассматривать сложность алгоритма — в худшем случае или же в среднем (последнее возможно, если только множество входов каждого фиксированного размера является вероятностным пространством).

16. Основные принципы построения сети Интернет. Иерархическая модель компьютерной сети. Адресация в сети Интернет, протоколы ARP, DHCP. Модели основных протоколов IP, TCP, ICMP. Модель взаимодействия приложений в Интернет.

Эталонная модель OSI

Эталонная модель OSI (за исключением физической среды) показана на рис. 1.16. Эта модель основана на разработке Международной организации по стандартизации (International Organization for Standardization, ISO) и является первым шагом к международной стандартизации протоколов, используемых на различных уровнях (Day и Zimmerman, 1983). Затем она была пересмотрена в 1995 году (Day, 1995). Называется эта структура эталонной моделью взаимодействия открытых систем ISO (**ISO OSI (Open System Interconnection) Reference Model**), поскольку она связывает открытые системы, то есть системы, открытые для связи с другими системами. Для краткости мы будем называть эту модель просто «модель OSI».

Модель OSI имеет семь уровней. Появление именно такой структуры было обусловлено следующими соображениями.

1. Уровень должен создаваться по мере необходимости отдельного уровня абстракции.
2. Каждый уровень должен выполнять строго определенную функцию.
3. Выбор функций для каждого уровня должен осуществляться с учетом создания стандартизированных международных протоколов.
4. Границы между уровнями должны выбираться так, чтобы поток данных между интерфейсами был минимальным.
5. Количество уровней должно быть достаточно большим, чтобы различные функции не объединялись в одном уровне без необходимости, но не слишком высоким, чтобы архитектура не становилась громоздкой.

Далее мы обсудим каждый уровень модели, начиная с самого нижнего. Обратите внимание: модель OSI не является сетевой архитектурой, поскольку она не описывает службы и протоколы, используемые на каждом уровне. Она просто определяет, что должен делать каждый уровень. Тем не менее ISO также разработала стандарты для каждого уровня, хотя эти стандарты не входят в саму эталонную модель. Каждый из них был опубликован как отдельный международный стандарт.

Физический уровень

Физический уровень занимается реальной передачей необработанных битов по каналу связи. При разработке сети необходимо убедиться, что когда одна сторона передает единицу, то принимающая сторона получает также единицу, а не ноль. Принципиальными вопросами здесь являются следующие: какое напряжение должно использоваться для отображения единицы, а какое — для нуля; сколько микросекунд длится бит; может ли передача производиться одновременно в двух направлениях; как устанавливается начальная связь и как она прекращается, когда обе стороны закончили свои задачи; из какого количества проводов должен состоять кабель и какова функция каждого провода. Вопросы разработки в основном связаны с механическими, электрическими и процедурными интерфейсами, а также с физическим носителем, лежащим ниже физического уровня.

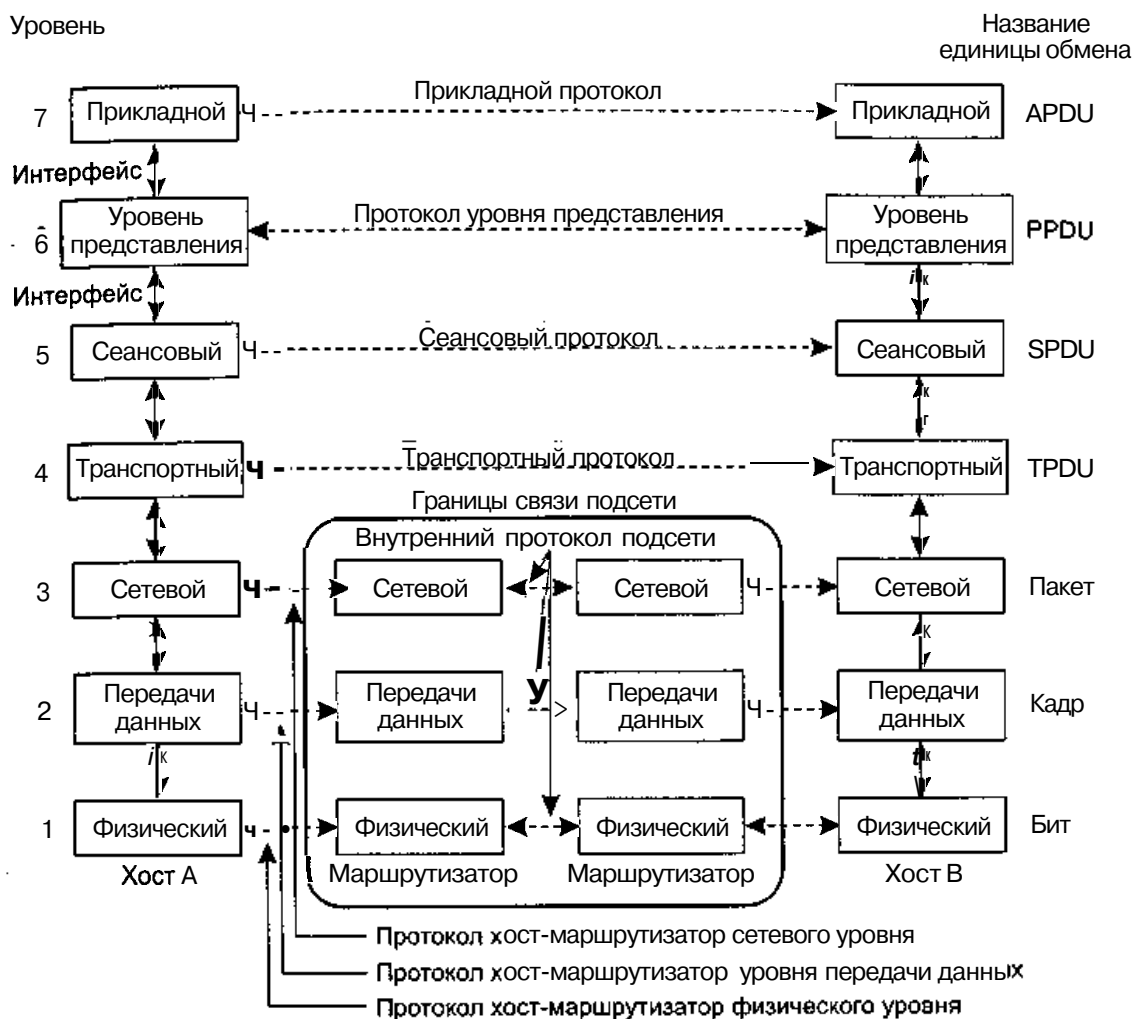


Рис. 1.16. Эталонная модель OSI

Уровень передачи данных

Основная задача уровня передачи данных — быть способным передавать «сырые» данные физического уровня по надежной линии связи, свободной от необ-

наруженных ошибок с точки зрения вышестоящего сетевого уровня. Уровень выполняет эту задачу при помощи разбиения входных данных на кадры, обычный размер которых колеблется от нескольких сотен до нескольких тысяч байт. Кадры данных передаются последовательно с обработкой кадров **подтверждения**, отсылаемых обратно получателем.

Еще одна проблема, возникающая на уровне передачи данных (а также и на большей части более высоких уровней), — как не допустить ситуации, когда быстрый передатчик заваливает приемник данными. Должен быть предусмотрен некий механизм регуляции, который информировал бы передатчик о наличии свободного места в буфере приемника на текущий момент. Часто подобное управление объединяется с механизмом обработки ошибок.

В широковещательных сетях существует еще одна проблема уровня передачи данных: как управлять доступом к совместно используемому каналу. Эта проблема разрешается введением специального дополнительного подуровня уровня передачи данных — подуровня доступа к носителю.

Сетевой уровень

Сетевой уровень занимается управлением операциями подсети. Важнейшим моментом здесь является определение маршрутов пересылки пакетов от источника к пункту назначения. Маршруты могут быть жестко заданы в виде таблиц и редко меняться. Кроме того, они могут задаваться в начале каждого соединения, например терминальной сессии. Наконец, они могут быть в высокой степени динамическими, то есть вычисляемыми заново для каждого пакета с учетом текущей загруженности сети.

Если в подсети одновременно присутствует слишком большое количество пакетов, то они могут закрыть дорогу друг другу, образуя заторы в узких местах. Недопущение подобной закупорки также является задачей сетевого уровня. В более общем смысле сетевой уровень занимается предоставлением определенного уровня сервиса (это касается задержек, времени передачи, вопросов синхронизации).

При путешествии пакета из одной сети в другую также может возникнуть ряд проблем. Так, способ адресации, применяемый в одной сети, может отличаться от принятого в другой. Сеть может вообще отказаться принимать пакеты из-за того, что они слишком большого размера. Также могут различаться протоколы, и т. д. Именно сетевой уровень должен разрешать все эти проблемы, позволяя объединять разнородные сети.

В ширококонтинентальных сетях проблема маршрутизации очень проста, поэтому в них сетевой уровень очень примитивный или вообще отсутствует.

Транспортный уровень

Основная функция **транспортного уровня** — принять данные от сеансового уровня, разбить их при необходимости на небольшие части, передать их сетевому уровню и гарантировать, что эти части в правильном виде придут по назначению. Кроме того, все это должно быть сделано эффективно и таким образом, чтобы

изолировать более высокие уровни от каких-либо изменений в аппаратной технологии.

Транспортный уровень также определяет тип сервиса, предоставляемого сеансовому уровню и, в конечном счете, пользователям сети. Наиболее популярной разновидностью транспортного соединения является защищенный от ошибок канал между двумя узлами, поставляющий сообщения или байты в том порядке, в каком они были отправлены. Однако транспортный уровень может предоставлять и другие типы сервисов, например пересылку отдельных сообщений без гарантии соблюдения порядка их доставки или одновременную отправку сообщения различным адресатам по принципу широковещания. Тип сервиса определяется при установке соединения. (Строго говоря, полностью защищенный от ошибок канал создать невозможно. Говорят лишь о таком канале, уровень ошибок в котором достаточно мал, чтобы ими можно было пренебречь на практике.)

Транспортный уровень является настоящим сквозным уровнем, то есть доставляющим сообщения от источника адресату. Другими словами, программа на машине-источнике поддерживает связь с подобной программой на другой машине при помощи заголовков сообщений и управляющих сообщений. На более низких уровнях для поддержки этого соединения устанавливаются соединения между всеми соседними машинами, через которые проходит маршрут сообщений. Различие между уровнями с 1-го по 3-й, действующими по принципу звеньев цепи, и уровнями с 4-го по 7-й, являющимися сквозными, проиллюстрировано на рис. 1.16.

Сеансовый уровень

Сеансовый уровень позволяет пользователям различных компьютеров устанавливать **сеансы связи** друг с другом. При этом предоставляются различные типы сервисов, среди которых **управление диалогом** (отслеживание очередности передачи данных), **управление маркерами** (предотвращение одновременного выполнения критичной операции несколькими системами) и **синхронизация** (установка служебных меток внутри длинных сообщений, позволяющих после устранения ошибки продолжить передачу с того места, на котором она оборвалась).

Уровень представления

В отличие от более низких уровней, задача которых — достоверная передача битов и байтов, **уровень представления** занимается по большей части синтаксисом и семантикой передаваемой информации. Чтобы было возможно общение компьютеров с различными представлениями данных, необходимо преобразовывать форматы данных друг в друга, передавая их по сети в некоем стандартизированном виде. Уровень представления занимается этими преобразованиями, предоставляя возможность определения и изменения структур данных более высокого уровня (например, записей баз данных).

Прикладной уровень

Прикладной уровень содержит набор популярных протоколов, необходимых пользователям. Одним из наиболее распространенных является протокол пере-

дачи гипертекста **HTTP** (HyperText Transfer Protocol), который составляет основу технологии Всемирной Паутины. Когда браузер запрашивает веб-страницу, он передает ее имя (адрес) и рассчитывает на то, что сервер будет использовать HTTP. Сервер в ответ отправляет страницу. Другие прикладные протоколы используются для передачи файлов, электронной почты, сетевых рассылок.

Эталонная модель TCP/IP

Интернет-уровень

Все эти требования обусловили выбор модели сети с коммутацией пакетов, в основе которой лежал не имеющий соединений межсетевой уровень. Этот уровень, называемый интернет-уровнем или межсетевым уровнем, является основой всей архитектуры. Его задача заключается в обеспечении возможности для каждого хоста посылать в любую сеть пакеты, которые будут независимо двигаться к пункту назначения (например, в другой сети). Они могут прибывать не в том порядке, в котором были отправлены. Если требуется соблюдение порядка отправления, эту задачу выполняют более верхние уровни. Обратите внимание, что сло-

во «интернет» здесь используется в своем первоначальном смысле несмотря на то, что этот уровень присутствует в сети Интернет.

Здесь можно увидеть аналогию с почтовой системой. Человек может бросить несколько международных писем в почтовый ящик в одной стране, и если повезет, большая часть из них будет доставлена по правильным адресам в других странах. Вероятно, письма по дороге пройдут через несколько международных почтовых шлюзов, однако это останется тайной для корреспондентов. В каждой стране (то есть в каждой сети) могут быть свои марки, свои предпочитаемые размеры конвертов и правила доставки, незаметные для пользователей почтовой службы.

Межсетевой уровень определяет официальный формат пакета и протокол, называемый **IP** (Internet Protocol). Задачей межсетевого протокола является доставка IP-пакетов к пунктам назначения. Основными аспектами здесь являются выбор маршрута пакета и недопущение закупорки транспортных артерий. Поэтому можно утверждать, что межсетевой уровень модели TCP/IP функционально близок сетевому уровню модели OSI. Это соответствие показано на рис. 1.17.

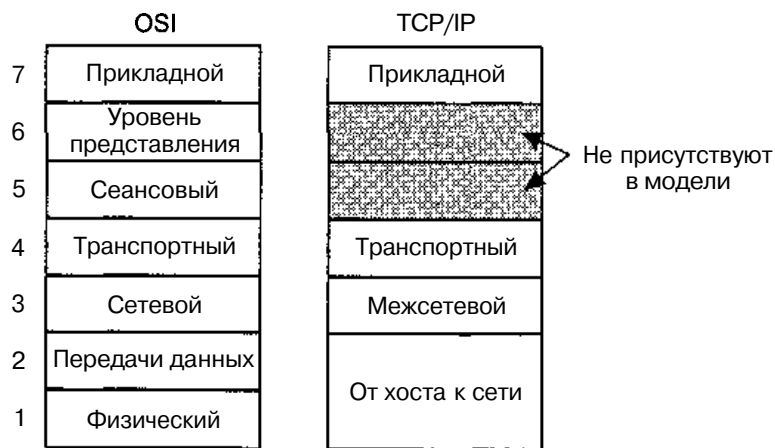


Рис. 1.17. Эталонная модель TCP/IP

Транспортный уровень

Уровень, расположенный над межсетевым уровнем модели TCP/IP, как правило, **называют транспортным**. Он создан для того, чтобы одноранговые сущности на приемных и передающих хостах могли поддерживать связь, подобно транспортному уровню модели OSI. На этом уровне должны быть описаны два сквозных протокола. Первый, **TCP** (Transmission Control Protocol — протокол управления передачей), является надежным протоколом с установлением соединений, позволяющим без ошибок доставлять байтовый поток с одной машины на любую другую машину объединенной сети. Он разбивает входной поток байтов на отдельные сообщения и передает их межсетевому уровню. В пункте назначения получающий TCP-процесс собирает из полученных сообщений выходной поток. Кроме того, TCP осуществляет управление потоком, чтобы быстрый отправитель не **завалил** информацией медленного получателя.

Второй протокол этого уровня, UDP (User Data Protocol - пользовательский протокол данных), является ненадежным протоколом без установления соединения, не использующим последовательное управление потоком протокола TCP, а предоставляющим свое собственное. Он также широко используется в одноразовых клиент-серверных запросах и приложениях, в которых оперативность важнее аккуратности, например, при передаче речи и видео. Взаимоотношения протоколов IP, TCP и UDP показаны на рис. 1.18. Со времени создания протокола IP этот протокол был реализован во многих других сетях.

Прикладной уровень

В модели TCP/IP нет сеансового уровня и уровня представления. В этих уровнях просто не было необходимости, поэтому они не были включены в модель. Опыт работы с моделью OSI доказал правоту этой точки зрения: большинство приложений в них мало нуждаются.

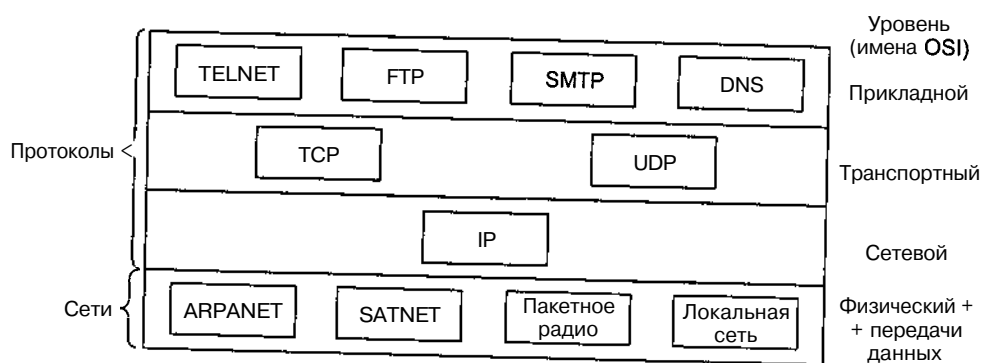


Рис. 1.18. Протоколы и сети в модели TCP/IP

Над транспортным уровнем располагается прикладной уровень. Он содержит все протоколы высокого уровня. К старым протоколам относятся протокол виртуального терминала (TELNET), протокол переноса файлов (FTP) и протокол электронной почты (SMTP), как показано на рис. 1.18. Протокол виртуального терминала позволяет пользователю регистрироваться на удаленном сервере и работать на нем. Протокол переноса файлов предоставляет эффективный способ перемещения информации с машины на машину. Электронная почта изначально представляла собой разновидность переноса файлов, однако позднее для нее был разработан специальный протокол. С годами было добавлено много других протоколов, таких как DNS (Domain Name Service — служба имен доменов), позволяющая преобразовывать имена хостов в сетевые адреса, NNTP (Network News Transfer Protocol — сетевой протокол передачи новостей), HTTP, протокол, используемый для создания страниц на World Wide Web, и многие другие.

Хост-сетевой уровень

В эталонной модели TCP/IP не описывается подробно, что располагается ниже межсетевого уровня. Сообщается только, что хост соединяется с сетью при помощи какого-нибудь протокола, позволяющего ему посылать по сети IP-пакеты. Этот протокол никак не определяется и может меняться от хоста к хосту и от сети к сети. В книгах и статьях, посвященных модели TCP/IP, этот вопрос обсуждается редко.

В настоящее время в Интернете действуют две версии IP-протокола: IPv4 и IPv6. Начнем рассмотрение с более ранней версии IPv4, а затем рассмотрим и IPv6, поэтому если противное не оговорено, то сейчас под IP-протоколом мы будем понимать IPv4.

На рис. 2.23 показан заголовок IP-пакета (дейтаграммы), имеющий обязательную часть размером 20 байт, которая может быть расширена до 60 байт. Дейтаграмму передают начиная с поля Version. Назначение полей заголовка следующее:

Version — указывает версию IP-протокола;

IHL — длина заголовка, который может содержать от 5 до 60 32-разрядных слов;

Type of service — вид необходимого сервиса. Здесь можно указать различные комбинации скорости и надежности (например, передача голоса, аккуратная доставка строки битов, файла и т. п.), а также задать приоритет дейтаграммы;

Total length — указывает общую длину пакета, включая заголовок и поле данных. Максимальная длина составляет 65 535 байт;

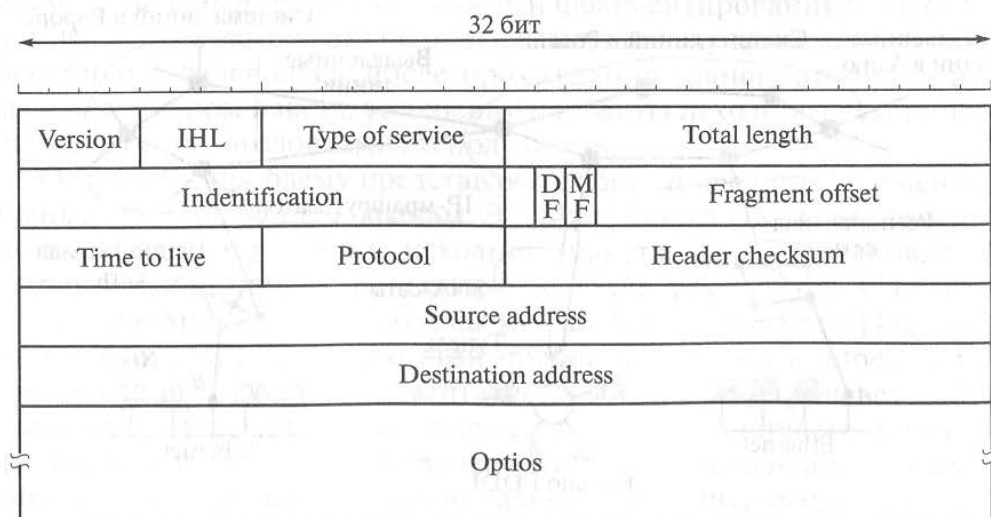


Рис. 2.23. Заголовок IP-пакета

Identification — позволяет отличать фрагменты одного и того же пакета, т. е. указывает, какому пакету принадлежит очередной поступивший фрагмент. Все фрагменты одного и того же пакета имеют в этом поле одно и то же значение;

DF — признак управления фрагментацией, причем если он равен 1, то фрагментация невозможна;

MF — содержит единицу у всех фрагментов пакета кроме последнего. Это поле позволяет отличить последний фрагмент от всех остальных;

Fragment offset — указывает, где в пакете располагается данный фрагмент пакета. Длина всех фрагментов, кроме последнего, должна быть кратна 8 байт. Поскольку поле имеет 13 разрядов, то у одного пакета максимально может быть 8 192 фрагментов. Таким образом, длина минимального фрагмента равна 8 байт;

Time to live — время жизни пакета. Максимальное значение этого поля составляет 255 с. Очень часто здесь используется счетчик скачков;

Protocol — показывает, какому протоколу на транспортном уровне передать собранную дейтаграмму (TCP, UDP и т.д.);

Header checksum — контрольная сумма, охватывающая только заголовок;

Source address, Destination address — идентифицируют машину отправителя и получателя на сетевом уровне;

Options — необязательная часть заголовка, предусмотренная для расширения возможностей протокола, которая может иметь следующие поля:

- Security — указывает уровень секретности передаваемой информации. Маршрутизатор может на основании значения этого поля запретить определенные маршруты, например если они пролегают через небезопасные регионы;

- Strict source routing — указывает полный маршрут в виде списка IP-адресов. Используется в алгоритме маршрутизации от источника, а также в критических ситуациях, например когда таблица маршрутизации по какой-то причине оказалась испорченной;

- Loose source routing — список маршрутизаторов, через которые фрагмент обязан пройти. Фрагмент может пройти и через другие маршрутизаторы, но перечисленные здесь обязательно должны принадлежать его маршруту;

- Record route — указывает маршрутизаторам на необходимость заносить в поле свои адреса. Это позволяет проследить маршрут, которым следовал фрагмент;

- Time stamp — вместе с полем Record route указывает маршрутизаторам на необходимость записывать не только свои адреса, но и время, когда фрагмент проходил через них. Это поле очень полезно при отладке алгоритмов маршрутизации.

Более подробное описание заголовка IP-протокола см. в [14].

2.5.3. IP-адресация

Каждая машина в Интернете имеет уникальный IP-адрес, состоящий из адреса сети и адреса машины в этой сети. Все IP-адреса имеют длину 32 разряда. На рис. 2.24 показаны форматы IP-адресов. Если машина подключена к нескольким сетям, то в каждой сети у нее будут свой IP-адрес и своя сетевая карта.

Все адреса на сетевом уровне в TCP/IP подразделяются на классы. Всего существует пять классов адресов: А, В, С, D, Е. В классе А адрес сети занимает 7 бит, под адрес машины в сети выделяется 24 бит. Таким образом, класс А позволяет адресовать до 126 сетей по 16 млн машин в каждой. В классе В под адрес сети выделено 14 разрядов, а под адрес машины в сети — 16. Тем самым класс В позволяет адресовать 16 382 сетей по 64 000 машин в каждой. В классе С под адрес сети отдано 19 разрядов, а под адрес машины — 8, т.е. 2 млн сетей по 256 машин. Адреса класса D предназначены для групповой передачи, а класса Е — зарезервированы для развития.

Адреса выделяет только организация Internet Network Information Center (www.internic.net).

На рис. 2.25 показаны IP-адреса, имеющие специальное назначение. IP-адрес с нулевым адресом сети используют для адресации

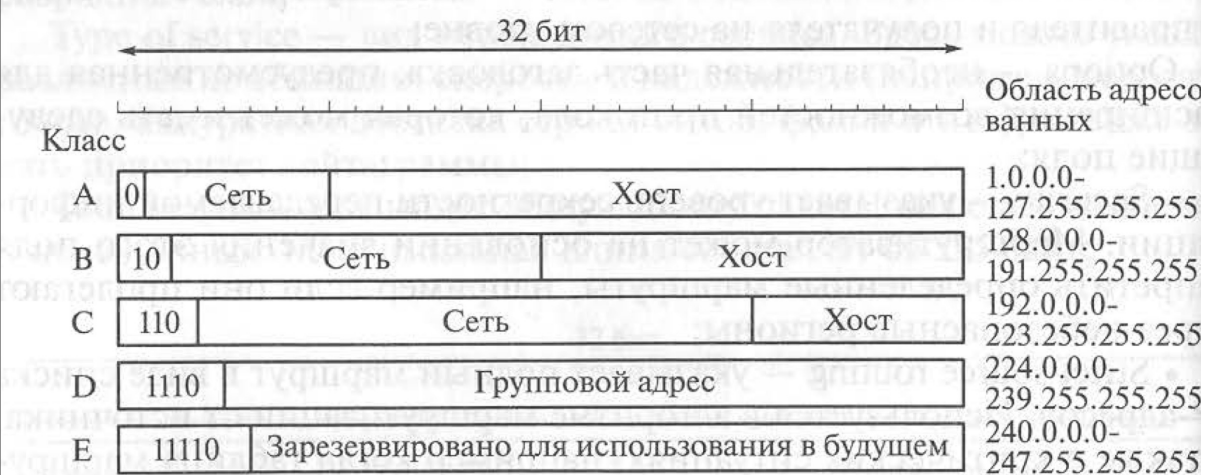


Рис. 2.24. Форматы IP-адресов



машин одной и той же сети. При этом необходимо соблюдать класс сети и соответственно число нулей в поле адреса сети. Адрес из одних нулей соответствует машине, на которой располагается программное обеспечение, используемое при загрузке машины, обратившейся по этому адресу. Адрес из одних единиц относится ко всем машинам в данной сети. Если единицы проставлены только в поле адреса машины, то такой адрес обозначает все хосты* в сети, адрес которой указан в поле адреса сети. Ну, и, наконец, адрес с номером сети 127 используется для тестирования сетевого программного обеспечения, когда сетевой интерфейс хоста заворачивает пакет с таким адресом себе же на вход. Адрес IP записывают в виде четырех десятичных чисел, разделенных точками.

2.5.4. Подсети

Все машины, принадлежащие одной и той же сети, должны иметь одинаковый номер сети в своем адресе. Это приводит к целому ряду проблем. По мере роста сети организации приходится изменять класс сети. Появление новых адресов приводит к проблеме повсеместной модификации таблиц маршрутизации и распространению информации о новых адресах. Перенос машины из одной сети в другую может потребовать изменения маршрутизации. Эти изменения происходят не сразу, и пока они не выполнены, все сообщения идут по старому адресу.

Решением указанных проблем является разделение сети на части, но таким образом, чтобы извне она по-прежнему представляла собой единое целое, а внутри нее каждая часть имела бы свой адрес. Часть адреса, идентифицирующая определенную группу сетевых адресов транспортной среды, называется адресом подсети, а все хосты, имеющие одинаковый адрес подсети, образуют подсеть. Итак, подсеть — это часть транспортной среды, невидимая извне. Изменение адреса подсети или введение новой подсети не требует обращения в NIC или изменения какой-либо глобальной базы данных.

На рис. 2.26 показано разбиение сети класса В на подсети.

Чтобы понять, как используется адрес подсети, следует проследить, как маршрутизатор использует IP-адрес. У маршрутизатора есть две таблицы. В первой таблице содержатся записи вида «сеть — 0», а во второй — вида «эта_сеть, адрес машины»**. Первая таблица показывает, как достичь интересующей сети, а вторая — как достичь хоста внутри сети. Когда поступает IP-пакет, маршрутизатор ищет его адрес доставки в первой таблице маршрутизации. Если этот адрес является

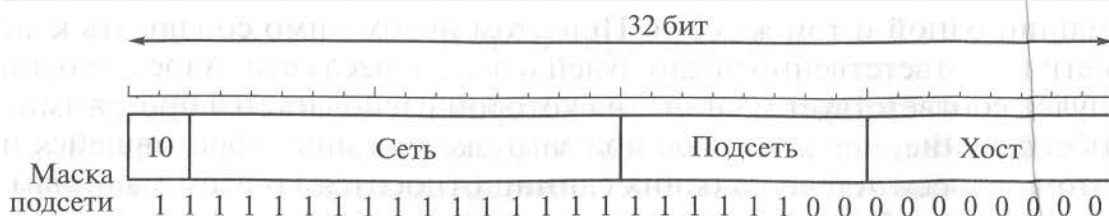


Рис. 2.26. Разбиение сети класса В на подсети

адресом другой сети, то пакет передается тому маршрутизатору, который отвечает за связь с указанной сетью. Если это адрес в сети данного маршрутизатора, то он, используя вторую таблицу, направляет пакет прямо по месту назначения. Если адреса нет во второй таблице, то маршрутизатор направляет пакет специально выделенному по умолчанию маршрутизатору, который должен разобраться с этим случаем с помощью более подробной таблицы.

Из приведенного описания видно, что алгоритм маршрутизации имеет дело только с сетями или локальными машинами, а не с парами сеть—машина. Такая организация алгоритма позволяет существенно сократить размер таблиц в маршрутизаторах.

С появлением подсети структура записей в таблице изменяется. Теперь записи в таблице имеют вид «эта_сеть, подсеть, 0» и «эта_сеть, эта_подсеть, машина». Таким образом, маршрутизатор подсети знает, как достичь любой подсети в данной локальной сети и как найти конкретную машину в своей подсети. Все что ему необходимо для этого — знать маску подсети. С помощью логической операции И и маски, показанной на рис. 2.26, маршрутизатор выделяет адрес подсети, а затем по своим таблицам определяет, как достичь требуемой подсети или (если это локальная подсеть для маршрутизатора) как достичь конкретной машины.

DNS - адрес

IP - адрес имеет числовой вид, так как его используют в своей работе компьютеры. Но он весьма сложен для запоминания, поэтому была разработана доменная система имен: DNS. DNS - адрес включает более удобные для пользователя буквенные сокращения, которые также разделяются точками на отдельные информационные блоки (домены). Например:

www.klyaksa.net

Если Вы вводите DNS - адрес, то он сначала направляется в так называемый сервер имен, который преобразует его в 32 - битный IP - адрес для машинного считывания.

Доменные имена

DNS - адрес обычно имеет три составляющие (хотя их может быть сколько угодно).

Доменная система имен имеет иерархическую структуру: домены верхнего

уровня - домены второго уровня и так далее. Домены верхнего уровня бывают двух типов: географические (двухбуквенные - каждой стране свой код) и административные (трехбуквенные).

URL

URL (Uniform Resource Locator, унифицированный определитель ресурсов) - это адрес некоторой информации в Интернет. Он имеет следующий формат:

тип ресурса://адрес узла/прочая информация

Наиболее распространенными считаются следующие типы ресурсов:

ftp:// ftp - сервер

gopher:// меню gopher

http:// адрес в WWW

mailto:// адрес электронной почты

news:// группа новостей UseNet

telnet:// компьютер, в котором можно зарегистрироваться, используя telnet

Протокол определения адреса — ARP

Хотя каждая машина в Интернете имеет уникальный IP-адрес, при передаче пакета через СПД от этого мало пользы, так как канальный уровень не понимает IP-адресов. Как правило, машина подключается к СПД через сетевую карту, которая понимает только адреса канального уровня, например Ethernet-адрес, имеющий 48 разрядов.

Сетевая карта знает только такие адреса и ничего не знает о 32-разрядных IP-адресах. Как отобразить 32-разрядный IP-адрес в адресах канального уровня, поясним с помощью рис. 2.27.

Когда машина 1 посылает сообщение машине 2, адрес доставки имеет вид smel@cs.msu.su. Сначала машина 1 с помощью службы имен домена — DNS (Domain Name Service — см. подразд. 4.2 определяет IP-адрес машины 2. Далее для отображения IP-адреса в Ethernet-адресе машина 1 посылает через СПД Ethernet следующий запрос: «У кого такой IP-адрес?». Этот запрос, также содержащий адрес СПД машины 1, увидят все машины в сети 192.31.65.0. Машина 2, увидев такой запрос и свой IP-адрес в нем, пошлет ответ, в котором указан ее СПД-адрес (в данном случае, Ethernet-адрес). Протокол, который реализует рассылку запросов и сбор ответов, т.е. протокол определения адреса — ARP (Address Resolution Protocol), описан в RFC 826. Практически у каждой машины в Интернете имеется этот протокол.

Теперь, зная СПД-адрес получателя, т.е. адрес требуемой машины в СПД (в данном случае, MAC-адрес), пакет с адресом 192.31.65.5 можно передать на канальный уровень. На канальном уровне полученный пакет помещают в Ethernet-кадр, в заголовке которого размещается найденный MAC-адрес машины 2. Машина 1 при этом запоминает MAC-адрес машины 2 на случай, если ей еще раз к ней придется обратиться. Ясно, что протокол должен учитывать время актуальности этой информации. Например, если на машине 2 заменят сетевую карту, то изменится и ее сетевой адрес. Поэтому данная информация должна периодически обновляться. Для того чтобы машине 2 не пришлось повторять всю процедуру поиска MAC-адреса машины 1, в заголовке кадра машина 1 укажет свой MAC-адрес, а в пакете — свой IP-адрес.

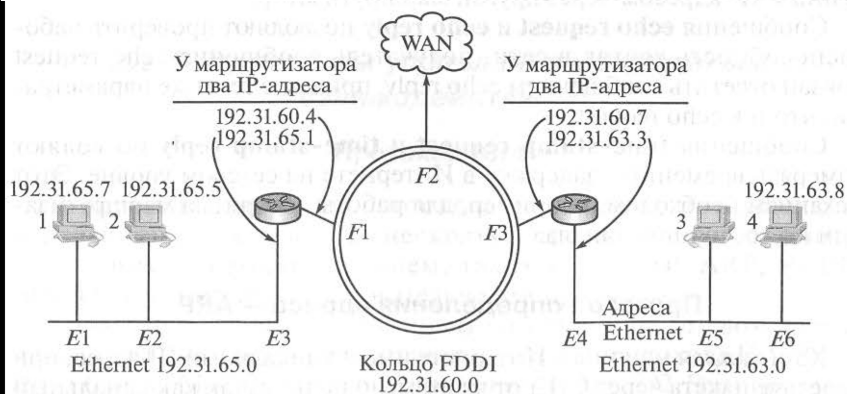


Рис. 2.27. Пример объединения сети класса C:

Теперь рассмотрим случай, когда обращение посылается в другую подсеть. Здесь возможны два решения. Первое решение: имеется определенный маршрутизатор, который принимает все сообщения, адресованные в определенную подсеть или группе адресов. Доступ в эту определенную подсеть возможен только через данный маршрутизатор, который называется *проху* и на котором работает протокол *проху* ARP. Этот маршрутизатор знает, как найти адресуемую машину в этой определенной подсети. Второе решение — использование выделенного маршрутизатора, который управляет маршрутизацией удаленного трафика. Машина-отправитель сама определяет, что обращение идет в удаленную сеть, и посылает сообщение на этот выделенный маршрутизатор. В нашем случае на маршрутизатор с MAC-адресом E3, который распаковывает кадр и находит искомый IP-адрес. По своим таблицам этот маршрутизатор определяет, что все пакеты для машин сети 192.31.60.0 следует пересылать через интерфейс с MAC-адресом F3. Маршрутизатор с MAC-адресом E3 посылает сообщение маршрутизатору F3 по кольцу FDDI. Если же он не знает MAC-адреса адреса F3, то посылает вещательный запрос с указанием искомого IP-адреса, на которые откликнется маршрутизатор с MAC-адресом F3.

Подробнее ARP-протокол рассмотрен в [8].

DHCP (англ. Dynamic Host Configuration Protocol – протокол динамической настройки узла)

– сетевой протокол, позволяющий компьютерам автоматически получать IP-адрес и другие параметры, необходимые для работы в сети TCP/IP. Данный протокол работает по модели «клиент-сервер». Для автоматической конфигурации компьютер-клиент на этапе конфигурации сетевого устройства обращается к так называемому серверу DHCP, и получает от него нужные параметры. Сетевой администратор может задать диапазон адресов, распределяемых сервером среди компьютеров. Это позволяет избежать ручной настройки компьютеров сети и уменьшает количество ошибок. Протокол DHCP используется в большинстве сетей TCP/IP.

Протокол DHCP предоставляет три способа распределения IP-адресов:

- **Ручное распределение.** При этом способе сетевой администратор сопоставляет аппаратному адресу (для Ethernet сетей это MAC-адрес) каждого клиентского компьютера определённый IP-адрес. Фактически, данный способ распределения адресов отличается от ручной настройки каждого компьютера лишь тем, что сведения об адресах хранятся централизованно (на сервере DHCP), и потому их проще изменять при необходимости.
- **Автоматическое распределение.** При данном способе каждому компьютеру на постоянное использование выделяется

произвольный свободный IP-адрес из определённого администратором диапазона.

• **Динамическое распределение.** Этот способ аналогичен автоматическому распределению, за исключением того, что адрес выдаётся компьютеру не на постоянное пользование, а на определённый срок. Это называется арендой адреса. По истечении срока аренды IP-адрес вновь считается свободным, и клиент обязан запросить новый (он, впрочем, может оказаться тем же самым). Кроме того, клиент сам может отказаться от полученного адреса. Некоторые реализации службы DHCP способны автоматически обновлять записи DNS, соответствующие клиентским компьютерам, при выделении им новых адресов. Это производится при помощи протокола обновления DNS, описанного в RFC 2136. Протокол DHCP является клиент-серверным, то есть в его работе участвуют клиент DHCP и сервер DHCP. Передача данных производится при помощи протокола UDP, при этом сервер принимает сообщения от клиентов на порт 67 и отправляет сообщения клиентам на порт 68.

IP объединяет сегменты сети в единую сеть, обеспечивая доставку данных между любыми узлами сети. Он

классифицируется как протокол третьего уровня по сетевой модели OSI. IP не гарантирует надёжной доставки пакета до адресата. В частности, пакеты могут прийти не в том порядке, в котором были отправлены, продублироваться (приходят две копии одного пакета), оказаться повреждёнными (обычно повреждённые пакеты уничтожаются) или не прийти вовсе. Гарантию безошибочной доставки пакетов дают некоторые протоколы более высокого уровня — *транспортного уровня* сетевой модели OSI, — например, TCP, которые используют IP в качестве транспорта.

- Предотвращает «зацикливание» пакетов
- Фрагментирует пакеты если они слишком длинные
- Использует контрольную сумму, чтобы сократить возможности доставки в неправильное место назначения

- Две версии
 - IPv4 с 32 битным адресом
 - IPv6 с 128 битным адресом
- Позволяет добавлять новые опции к заголовку Модель TCP.

3.3.2. Модель сервиса TCP

Доступ к TCP-сервису происходит через сокет. Сокет состоит из IP-адреса хоста и 16-разрядного локального номера на хосте, называемого *порт*. Сокеты создаются как отправителем, так и получателем. Порт — это TSAP для TCP. Каждое соединение идентифицируется парой сокетов, между которыми оно установлено. Один и тот же сокет может быть использован для разных соединений. Никаких дополнительных виртуальных соединений не создается.

Порты до 256 номера зарезервированы для стандартных сервисов, которые постоянно активны и готовы к работе. Например, для обеспечения FTP-передачи файла (см. гл. 4) соединение должно выполняться через 21-й порт, где находится FTP-демон, а для TELNET — через 23-й порт. Полный список таких портов можно найти в RFC 1700.

Все TCP-соединения — дуплексные, т. е. передача по ним происходит независимо в оба направления. TCP-соединение поддерживает только соединение точка—точка. Не существует TCP-соединений типа «от одного ко многим».

TCP-агент поддерживает поток байтов, а не поток сообщений. Напомним, это означает, что границы сообщений не поддерживаются автоматически в потоке. Например, если по TCP-соединению передается текст, разбитый на страницы, то TCP-агент не будет каким-либо образом обозначать конец каждой страницы.

После передачи приложением данных TCP-агенту, эти данные могут быть отправлены сразу на сетевой уровень, а могут быть буферизованы. Как поступить в этом случае решает TCP-агент. Однако в ряде случаев бывает необходимо, чтобы данные были отправлены

сразу, например если эти данные представляют собой команду для удаленной машины. Для этого в заголовке TCP-сегмента имеется флаг PUSH, и если он установлен, то это говорит TCP-агенту о том, что данные должны быть переданы немедленно.

Наконец, если в заголовке TPDU-сегмента установлен флаг URGENT, то TCP-агент передает такой сегмент незамедлительно. Когда срочные данные поступают к месту назначения, то их передают получателю немедленно.

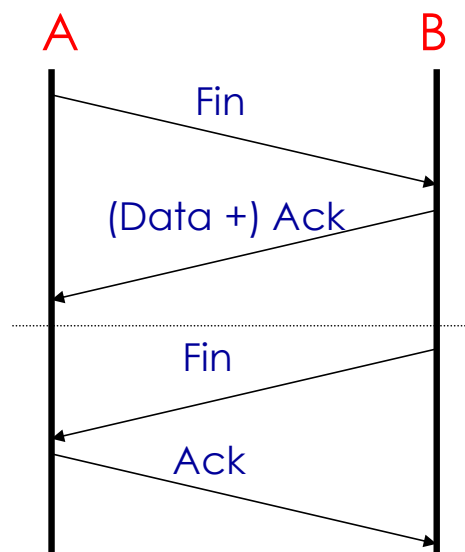
Основным протоколом, который используется TCP-агентом, является протокол скользящего окна. Это означает, что каждый посланный сегмент должен быть подтвержден. Одновременно с отправлением сегмента запускается таймер. Подтверждение придет либо с очередными данными в попутном направлении, если они имеются, либо без данных само по себе и будет иметь порядковый номер очередного ожидаемого получателем сегмента. Если таймер сработает прежде, чем придет подтверждение, то сегмент посылается повторно.

которому.

Несмотря на кажущуюся простоту, TCP-протокол достаточно сложен и должен решать следующие основные проблемы:

- восстанавливать порядок сегментов;
- убирать дубликаты сегментов, в каком бы виде они не поступали;
- определять разумную задержку для time-out на подтверждение в получении сегмента;
- надежно устанавливать и разрывать соединения;
- управлять потоком;
- управлять перегрузками.

- **Инициатор шлет 1-й пакет**
 - SYN + sequence number
- **Вызываемый отвечает**
 - SYN + sequence number
 - ACK для SYN инициатора
- **Инициатор отвечает**
 - ACK на SYN вызываемого
- **Также поддерживается одновременное открытие**
 - Два SYN передаются друг другу
 - Каждая сторона шлет ACK другой стороне



Интернет модель ICMP сервиса

- Предназначен для поддержки IP протокола
- IP протокол
 - Создает IP дейтаграммы
 - Передает их hop-by-hop
- Таблицы маршрутизации и алгоритмы распространения таких таблиц
- ICMP протокол
 - Обеспечивает коммуникацию между сетевыми уровнями хостов и маршрутизаторов
 - Сообщения об ошибках
 - Диагностика проблем
- ICMP предоставляет информацию о сетевом уровне хостам и маршрутизаторам
- ICMP работает над IP и относится к транспортному уровню

! к транспортному уровню он относится только на слайдах у Смелянского, в остальных относится к сетевому

- ping и traceroute реализованы с помощью ICMP

настройка над сетевым уровнем

Сервис без соединения и подтверждения

Управление функционированием Интернета на сетевом уровне происходит через маршрутизаторы с помощью протокола ICMP (Internet Control Message Protocol), описанного в RFC 792. Этот протокол обеспечивает доставку сообщений любой машине, имеющей IP-адрес, от маршрутизаторов и других хостов в сети. С помощью этого протокола реализуют обратную связь для решения проблем, возни-

кающих при передаче. Он также выявляет и рассылает сообщения о десятках событий. Для доставки своих сообщений протокол ICMP использует пакеты IP-протокола. Приведем наиболее важные сообщения.

Сообщение **destination unreachable** охватывает множество случаев, например, случай, когда маршрутизатор не знает, как достигнуть необходимой подсети или хоста, или случай, когда дейтаграмма при доставке должна быть фрагментирована, но установлен флаг, который запрещает это делать.

Сообщение **time exceeded** посылает маршрутизатор, если он обнаружил дейтаграмму с истекшим времени жизни. Это сообщение также генерирует хост, если он не успел завершить обработку IP-пакета до истечения времени его жизни. (Далее слова «пакет», «IP-пакет», «дейтаграмма» будем понимать как синонимы, если специально не оговорено что-либо другое.)

Синтаксические или семантические ошибки в заголовке IP-пакета вызывают появление сообщения **parameter problem**.

Маршрутизатор или хост-получатель высылает этот пакет хосту-отправителю, если последнему необходимо понизить скорость передачи. Другими словами, это пример подавляющего пакета (см. подразд. 2.4). Сообщения такого типа будут генерироваться до тех пор, пока скорость поступления пакетов от отправителя не достигнет значения, необходимого хосту-получателю. Это сообщение система может использовать для предотвращения перегрузки, поскольку оно возникает всякий раз, когда маршрутизатор вынужден сбросить пакет из-за переполнения своего буфера.

Сообщение **redirect** позволяет маршрутизатору отправить рекомендацию о лучшем маршруте и впредь посылать пакеты с определенным IP-адресом через другой маршрутизатор.

Сообщения **echo request** и **echo reply** позволяют проверить работоспособность хостов в сети: получатель сообщения echo request обязан ответить сообщением echo reply, причем с теми же параметрами, что и в echo request.

Сообщения **time-stamp request** и **time-stamp reply** позволяют измерять временную задержку в Интернете на сетевом уровне. Этот механизм необходим, например, для работы алгоритма маршрутизации по состоянию канала.

17. Физический уровень стека сетевых протоколов. Технологии Ethernet и Wi-Fi. Алгоритмы работы, коллизии, управление множественным доступом к каналу.

Назначение физического уровня — передавать данные в виде потока битов от одной машины к другой. При этом для передачи данных можно использовать различные физические среды, каждую из которых характеризуют следующие параметры:

- ширина полосы пропускания;
- пропускная способность;;
- задержка сигнала;;
- стоимость;;
- сложность прокладки;;
- сложность обслуживания.

Кроме перечисленных, физическую среду характеризуют и другие параметры, например:

- достоверность передачи;;
- затухание;;
- помехоустойчивость.

Магнитная лента или магнитный диск:

- + Высокая пропускная способность.
- + Стоимость передачи в расчёте на один бит.
- + Надежность.
- - Огромная задержка.

Витая пара — два медных изолированных провода, один из которых обвит вокруг другого. Этот второй вьющийся провод предназначен для устранения взаимного влияния между соседними витыми парами. Широко используется в телефонии. Протяжённость линии из витой пары между абонентами и автоматизированной телефонной станцией может составлять несколько километров без промежуточных усилений. Витая пара может быть использована для передачи как цифровых, так и аналоговых сигналов. Её пропускная способность зависит от толщины используемых проводов и длины линии. Скорость в несколько мегабит вполне достижима с помощью соответствующих методов передачи. Витые пары объединяют в многопарные кабели. Кабель категории 3 содержит четыре витые пары с невысокой плотностью намотки и имеет ширину полосы пропускания до 16 МГц. Наиболее часто используется кабель категории 5, который тоже состоит из четырёх пар, но имеет более плотную намотку, что позволяет достигать более высоких скоростей передачи и ширину полосы пропускания 100 МГц.

Как и у витой пары, у коаксиального кабеля есть два проводника. Однако устроены они иначе, что позволяет существенно увеличить ширину полосы

пропускания. Центральный проводник представляет собой толстый медный провод, окружённый изолятором. Эта конструкция помещается внутри второго цилиндрического проводника, который обычно представляет собой плетённую плотную металлическую сетку. Всё это закрывается плотным защитным слоем пластика. Обычно толщина коаксиального кабеля составляет от 1 до 2.5 см, поэтому монтировать и прокладывать его сложнее, чем витую пару, однако у такого кабеля шире полоса пропускания и характеристики по затуханию сигнала лучше, чем у витой пары. Основным ограничителями скорости и расстояния при передаче без усиления в этих кабелях являются затухания сигнала, тепловой и интермодуляционный шум. Когда всю полосу пропускания кабеля разбивают на более узкие и каждую такую полосу используют как отдельный канал, на границах таких каналов возникает интермодуляционный шум.

Для создания оптической связи требуется источник света и постоянной длиной волны, светопроводящая среда и детектор, преобразующий световой поток в электрический. На одном конце оптоволоконной линии имеется передатчик – источник света, световой импульс от которого проходит по светопроводящему волокну и попадает на детектор, расположенный на другом конце этой линии и преобразующий этот импульс в электрический. Одна из основных проблем при создании оптоволоконных систем состояла в том, чтобы не дать световому пучку рассеяться через боковую поверхность силиконового шнура. Количество рассеиваемой энергии зависит от угла падения светового луча на стенки шнура. При углах больше некоторого критического угла, называемого углом полного внутреннего отражения, вся энергия луча отражается обратно внутрь. Оказалось, что силиконовый шнур, имеющий толщину, близкую к длине волны источника света, работает как провод для тока, т.е. без потерь на внутреннее отражение. По такому одномодовому шнуру можно передать данные со скоростью несколько гигабит в секунду на сотни километров без промежуточного усиления. Поскольку можно испускать несколько лучей разной длины волны так, что они попадали на границы шнура под углом больше, чем угол полного внутреннего отражения, следовательно, по одному шнуру можно пускать несколько лучей. При этом каждый луч, как говорят, имеет свою моду. Так получается многомодовый шнур. Другой проблемой при использовании оптоволоконна является дисперсия – потеря по мере распространения

исходными световыми импульсами начальных форм и размеров. При этом искажения также зависят от длины волны. Одно из возможных решений этой проблемы – увеличение расстояния между соседними сигналами. Однако это сокращает скорость передачи. К счастью, исследования показали, что при придании сигналу некоторой специальной формы дисперсионные эффекты почти исчезают и сигнал можно передавать на тысячи километров. Сигналы, имеющие такую форму, называются силитонами.

Преимущества относительно коаксиального:

- Ширина полосы пропускания больше.
- Более компактное и имеет меньшую массу.
- Затухание сигнала существенно меньше.
- Не восприимчиво к внешним электромагнитным излучениям.

Стандарты Ethernet определяют проводные соединения и электрические сигналы на [физическом уровне](#), формат [кадров](#) и протоколы управления доступом к среде — на [канальном уровне модели OSI](#).

Кабели в стандарте IEEE 802.3.

Первым появился так называемый толстый Ethernet(10Base5).

Коаксиальный кабель жёлтого цвета с отметками через каждые 2.5 м, указывающими, где можно производить подключение. Подключение выполняется через специальные розетки, которые монтируются прямо на кабеле. В эти розетки встраивается специальный прибор – трансивер, отвечающий за обнаружение несущей частоты и коллизий. Когда трансивер обнаруживает коллизию, он посылает специальный сигнал по кабелю, гарантирующий, что другие трансиверы услышат эту коллизию. Кабель обеспечивает пропускную способность 10 Мбит/с, а максимальная длина равна 500м(10Base5).

Вторым появился 10Base2. Это более простой в употреблении коаксиальный кабель с простым подключением через BNC-коннектор, представляющий собой Т-образное соединение коаксиальных кабелей. Его сегмент не должен превышать 200м и объединять более 30 машин.

Решение проблемы поиска обрыва, частичного повреждения кабеля или плохого контакта в коннекторе привели к созданию совершенно иной кабельной конфигурации на основе витой пары. В этом случае каждая машина соединяется витой парой со специальным устройством – хабом. Такой способ обозначается 10Base-T.

Трансиверы в 10Base5 размещаются прямо на кабеле и соединяются с компьютером трансиверным кабелем, длина которого не может превышать 50м. Трансиверный кабель состоит из пяти витых пар. Две из них используются для передачи данных к компьютеру и от него, две служат для передачи управляющей информации в обе стороны, а пятая – для подачи питания на трансивер.

Трансиверный кабель подключается к контроллеру в компьютере через интерфейс AUI. В контроллере имеется специальная микросхема, отвечающая за приём кадров и их отправку, проверку и формирование контрольной суммы. В некоторых случаях эта микросхема отвечает и за управление буферами на канальном уровне, очередью буферов на отправку и обеспечивает прямой доступ к памяти машины, а также решает другие вопросы доступа к сети.

В 10Base2 трансивер располагается на контроллере, и каждая машина должна иметь свой индивидуальный трансивер.

В 10Base-T трансивера нет вовсе. Машины соединяются кабелем с витой парой, длина которой не должна превышать 100м. Вся электроника сосредоточена в нём.

Последний используемый в стандарте IEEE 802.3 тип кабеля – оптоволокно 10Base-F, которое относительно дорогое, но обеспечение низкого уровня шума и большая длина одного сегмента являются его преимуществами. Для увеличения длины сегментов в этом стандарте используются репитеры – устройства физического уровня, отвечающие за очистку, усиление и передачу сигнала. Репитеры не могут отстоять друг от друга более чем на 2.5 км, и на одном сегменте их не может быть более четырёх.

В сетях Ethernet используется метод доступа к среде передачи данных, называемый методом коллективного доступа с опознаванием несущей и обнаружением коллизий МДКН/ОК (CSMA/CD).

Этот метод используется исключительно в сетях с общей шиной. Все компьютеры такой сети имеют непосредственный доступ к общей шине, поэтому она может быть использована для передачи данных между любыми двумя узлами сети. Простота схемы подключения – это один из факторов, определивших успех стандарта Ethernet. Говорят, что кабель, к которому подключены все станции, работает в режиме коллективного доступа.

Все данные, передаваемые по сети, помещаются в кадры определенной структуры и снабжаются уникальным адресом станции назначения. Затем кадр передается по кабелю. Все станции, подключенные к кабелю, могут распознать факт передачи кадра, и та станция, которая узнает собственный адрес в заголовках кадра, записывает его содержимое в свой внутренний буфер, обрабатывает полученные данные и посылает по кабелю кадр-ответ. Адрес станции-источника также включен в исходный кадр, поэтому станция-получатель знает, кому нужно послать ответ.

При описанном подходе возможна ситуация, когда две станции одновременно пытаются передать кадр данных по общему кабелю (рис. 3). Для уменьшения вероятности этой ситуации непосредственно перед отправкой кадра передающая станция слушает кабель (то есть принимает и анализирует возникающие на нем электрические сигналы), чтобы обнаружить, не передается ли уже по кабелю кадр данных от другой станции. Если опознается несущая (carrier-sense, CS), то станция откладывает передачу своего кадра до

окончания чужой передачи, и только потом пытается вновь его передать. Но даже при таком алгоритме две станции одновременно могут решить, что по шине в данный момент времени нет передачи, и начать одновременно передавать свои кадры. Говорят, что при этом происходит коллизия, так как содержимое обоих кадров сталкивается на общем кабеле, что приводит к искажению информации.

Чтобы корректно обработать коллизию, все станции одновременно наблюдают за возникающими на кабеле сигналами. Если передаваемые и наблюдаемые сигналы отличаются, то фиксируется обнаружение коллизии. Для увеличения вероятности немедленного обнаружения коллизии всеми станциями сети, ситуация коллизии усиливается посылкой в сеть станциями, начавшими передачу своих кадров, специальной последовательности битов, называемой jam-последовательностью.

После обнаружения коллизии передающая станция обязана прекратить передачу и ожидать в течение короткого случайного интервала времени, а затем может снова сделать попытку передачи кадра.

Из описания метода доступа видно, что он носит вероятностный характер, и вероятность успешного получения в свое распоряжение общей среды зависит от загрузки сети, то есть от интенсивности возникновения в станциях потребности передачи кадров. При разработке этого метода предполагалось, что скорость передачи данных в 10 Мб/с очень высока по сравнению с потребностями компьютеров во взаимном обмене данными, поэтому загрузка сети будет всегда небольшой. Это предположение остается часто справедливым и по сей день, однако уже появились приложения, работающие в реальном масштабе времени с мультимедийной информацией, для которых требуются гораздо более высокие скорости передачи данных.

Метод CSMA/CD определяет основные временные и логические соотношения, гарантирующие корректную работу всех станций в сети:

- Между двумя последовательно передаваемыми по общей шине кадрами информации должна выдерживаться пауза в 9.6 мкс; эта пауза нужна для приведения в исходное состояние сетевых адаптеров узлов, а также для предотвращения моно-польного захвата среды передачи данных одной станцией.
- При обнаружении коллизии станция выдает в среду специальную 32-х битную последовательность (jam-последовательность), усиливающую явление коллизии для более надежного распознавания ее всеми узлами сети.
- После обнаружения коллизии каждый узел, который передавал кадр и столкнулся с коллизией, после некоторой задержки пытается повторно передать свой кадр. Узел делает максимально 16 попыток передачи этого кадра информации, после чего отказывается от его передачи.

Четкое распознавание коллизий всеми станциями сети является необходимым условием корректной работы сети Ethernet. Если какая-либо передающая станция не распознает коллизию и решит, что кадр данных ею передан верно, то этот кадр данных будет утерян, так как информация кадра исказится из-за наложения сигналов при

коллизии, он будет отбракован принимающей станцией.

С увеличением скорости передачи кадров, что имеет место в новых стандартах, базирующихся на том же методе доступа CSMA/CD, например, Fast Ethernet, максимальная длина сети уменьшается пропорционально увеличению скорости передачи. В стандарте Fast Ethernet она составляет 210 м, а в гигабитном Ethernet ограничена 25 метрами.

Независимо от реализации физической среды, все сети Ethernet должны удовлетворять двум ограничениям, связанным с методом доступа:

- максимальное расстояние между двумя любыми узлами не должно превышать 2500 м,
- в сети не должно быть более 1024 узлов.

Кроме того, каждый вариант физической среды добавляет к этим ограничениям свои ограничения, которые также должны выполняться. Уточним основные параметры операций передачи и приема кадров Ethernet, кратко описанные выше.

Станция, которая хочет передать кадр, должна сначала с помощью MAC-узла упаковать данные в кадр соответствующего формата. Затем для предотвращения смещения сигналов с сигналами другой передающей станции, MAC-узел должен прослушивать электрические сигналы на кабеле и в случае обнаружения несущей частоты 10 МГц отложить передачу своего кадра. После окончания передачи по кабелю станция должна выждать небольшую дополнительную паузу, называемую межкадровым интервалом, что позволяет узлу назначения принять и обработать передаваемый кадр, и после этого начать передачу своего кадра.

Одновременно с передачей битов кадра приемно-передающее устройство узла следит за принимаемыми по общему кабелю битами, чтобы вовремя обнаружить коллизию. Если коллизия не обнаружена, то передается весь кадр, после чего MAC-уровень узла готов принять кадр из сети либо от LLC-уровня.

Если же фиксируется коллизия, то MAC-узел прекращает передачу кадра и посылает jam-последовательность, усиливающую состояние коллизии. После посылки в сеть jam-последовательности MAC-узел делает случайную паузу и повторно пытается передать свой кадр.

В случае повторных коллизий существует максимально возможное число попыток повторной передачи кадра, которое равно 16. При достижении этого предела фиксируется ошибка передачи кадра, сообщение о которой передается протоколу верхнего уровня.

MAC обеспечивает адресацию и механизмы управления доступом к каналам, что позволяет нескольким терминалам или точкам доступа общаться между собой в многоточечной сети (например, в [локальной](#) или [городской](#) вычислительной сети), и эмулирует [полнодуплексный](#) логический канал связи в многоточечной сети.

Для того, чтобы уменьшить интенсивность коллизий, каждый МАС-узел с каждой новой попыткой случайным образом увеличивает длительность паузы между попытками. Временное расписание длительности паузы определяется на основе усеченного двоичного экспоненциального алгоритма отсрочки. Пауза всегда составляет целое число так называемых интервалов отсрочки.

Интервал отсрочки - это время, в течение которого станция гарантированно может узнать, что в сети нет коллизии.

3.4.7. Стандарт беспроводных локальных сетей IEEE 802.11

Рассмотрим один из активно развивающихся стандартов беспроводной связи для локальных сетей, так называемый стандарт WiFi (Wireless Fidelity). Как и для кабельных сетей, стандартизацией беспроводных протоколов в настоящее время занимается в основном организация IEEE (см. подразд. 2.6.7). Семейство стандартов, определяющих функционирование беспроводных локальных сетей, описаны в секции 11 семейства стандартов IEEE 802.

Первой и ключевой технологией стандарта 802.11 является технология расширения спектра передачи методом прямой последовательности (Direct Sequence Spread Spectrum — DSSS). Использование DSSS позволяет беспроводным интерфейсам передавать данные со скоростью от 1 до 2 Мбит/с.

Идея этого метода состоит в следующем. Пусть имеется широкий канал, т.е. канал с широкой полосой пропускания. Разобьем его на

полосы. Каждому значению бита сопоставим определенный код с длиной, равной числу полос, на которые разбили канал. Теперь будем передавать каждый бит, параллельно передавая его код, причем каждый элемент кода в своей полосе. Элементы кода называются *чипами*. Такой способ передачи позволяет эффективнее использовать полосу пропускания канала, и он более надежен по сравнению с традиционным способом передачи.

Рассмотрим пример. Пусть бинарную единицу представляет набор чипов вида 00110011011, а бинарный 0 — набор чипов вида 11001100100. Обратим внимание, что если эти последовательности рассматривать как векторы и взять их скалярное произведение, то получим 0, т.е. можно сказать, что эти последовательности ортогональны друг другу.

Далее все чипы одного набора передают в эфир параллельно.

Пусть вследствие искажений получена последовательность вида 01111011011. Сравнив ее с эталонными последовательностями, видим следующее: отличие от эталона бинарной единицы составляет 2 бит, а отличие от эталона бинарного нуля — 9 бит. Значит, перед нами единица. Это так называемый код с исправлением ошибок. (Подробнее построение таких кодов рассматривается в подразд. 4.1).

Метод кодирования сигнала при передаче отдельного чипа в полосе может быть разным. Например, BPSK (Binary Phase Shift Keying) — кодирование бинарным фазовым сдвигом. Сдвиг фаз здесь может иметь два значения — 0 и π . Скорость передачи данных таким способом составляет в разрешенных диапазонах (шириной в 22 МГц) 1 Мбит/с.

Для получения скорости передачи 2 Мбит/с используется метод QPSK (Quadrature Phase Shift Keying) — кодирование квадратурно-фазовым сдвигом. Сдвиг фаз здесь может иметь четыре значения 0, $\pi/2$, π , $3\pi/2$. Соответственно скорость этого метода в два раза выше, чем при использовании метода BPSK.

Для обеспечения скоростей передачи 5,5 и 11 Мбит/с используется метод CCK (Complimentary Code Keying) — кодирование комплиментарными кодами, смысл которого поясним на примере комплексных чисел. Пусть разрешено использовать только те комплексные числа, модуль которых принадлежит множеству $\{1, -1, i, -i\}$. Таких чисел восемь, т. е. можно сформировать восемь одинаковых по модулю, но не одинаковых по фазе комплексных чисел. При этом комплексная последовательность чипов может принимать следующие значения: $1, -1, i, -i, 1+i, 1-i, -1-i, -1+i$, а каждый элемент 8-чиповой последовательности может иметь восемь значений фазы. Тогда число вариантов составит 8 в степени 8. Конечно, не все значения в этом случае будут комплиментарны, но истинно комплиментарных значений хватит для того, что бы передавать сразу до восьми бит.

Если продолжать бездумно двигаться по этому пути, можно еще больше увеличить скорость передачи, но здесь начинает сказываться...

эффект интерференции множества сигналов с разными амплитудами и разным временем прибытия к приемнику. Этот эффект, который может вызывать даже отражение части сигнала от потолков, стен и прочих поверхностей, получил название MultiPath Distortion — искажения, вызванные множественностью путей прохождения сигнала.

Для борьбы с этим эффектом, а также для обеспечения возможности увеличения скорости передачи данных был разработан метод OFDM (Orthogonal Frequency Division Multiplexing) — мультиплексирование с ортогональным частотным разделением каналов. Его идея заключается в том, что сигнал поляризуют в ортогональной по отношению к соседним полосам плоскости. Кроме того, между передаваемыми символами вставляется охранный интервал длительностью 0,8 мкс. Для ликвидации взаимного влияния соседей по частотному диапазону по его краям имеются также защитные подканалы.

Минимально сеть WiFi может содержать всего два устройства, например две рабочие станции или два ноутбука. В этом случае организуется *выделенная* (ad hoc) сеть, в которую входят беспроводные интерфейсы указанных устройств. Данное соединение можно сравнить с соединением типа точка—точка в проводной связи. Однако соединение WiFi имеет следующие существенные недостатки: во-первых, безопасность передачи данных в таком соединении минимальна; во-вторых, зачастую беспроводные интерфейсы разных производителей несовместимы друг с другом в режиме ad hoc.

Для решения задачи совместимости в сети устанавливается выделенный узел — точка доступа, представляющая собой устройство, которое имеет проводной интерфейс для подключения к проводной сети и доступа во внешний мир, а также антенну, образующую вокруг себя зону покрытия точки доступа. В зависимости от того, где установлена точка доступа, а также от того, каковы мощность и размер ее антенны, размер зоны покрытия может сильно различаться. Обычно радиус зоны покрытия точки доступа составляет от 90 до 150 м. Для обслуживания больших площадей устанавливают несколько точек доступа с перекрывающимися зонами покрытия. Перекрывание этих зон позволяет организовать прозрачное перемещение абонентских устройств в пределах общей зоны покрытия (по такому же принципу строятся сети сотовой связи). Размер зоны перекрывания не установлен в стандарте, но обычно он составляет 20...30 %.

Когда абонентское устройство включается внутри зоны покрытия беспроводной сети, оно начинает прослушивать эфир в поисках совместимого устройства, с которым оно могло бы взаимодействовать.

Этот этап называется *сканированием*, которое может быть активным или пассивным.

При активном сканировании генерируется широкополосный запрос от абонентского устройства, обязательно включающий в себя идентификатор сети, к которой он хочет присоединиться (Service Set Identifier — SSID). Когда запрос достигает точки доступа, имеющей запрашиваемый идентификатор сети, эта сеть генерирует ответ на запрос.

При пассивном сканировании абонентское устройство слушает эфир и ожидает появления кадров-маяков, которые периодически рассылаются точками доступа или интерфейсами других узлов, находящимися в режиме ad hoc. Когда абонентское устройство получает кадр-маяк, в котором указан идентификатор сети SSID, оно пытается присоединиться к указанной сети. Пассивное сканирование — это постоянный процесс, в котором узлы могут присоединяться к точке доступа или отсоединяться по мере изменения мощности радиосигнала.

После присоединения узла к беспроводной сети он может отправлять и принимать кадры точно так же, как в стандарте IEEE 802.3. Беспроводные стандарты семейства 802.11 используют формат кадра, отличный от Ethernet. В WiFi определены три типа кадров: контрольные, управляющие и кадры данных. Формат кадров данных здесь аналогичен формату кадра в Ethernet. Размер поля данных в WiFi, как и в стандарте 802.3, равен 1500 байт. В то же время тогда как размер кадра в Ethernet не может превышать 1518 байт, максимальный размер кадра в WiFi составляет 2346 байт. Как правило, размер кадра в беспроводной сети ограничивается значением 1518 байт для облегчения передачи кадров между сегментами проводной и беспроводной сетей.

Так как радиоканал является средой множественного доступа, в нем могут происходить коллизии, как и в проводной Ethernet. Основное отличие между ними заключается в том, что в беспроводной сети источник никак не может определить возникновение коллизии. Для разрешения коллизий в беспроводной сети используется метод множественного доступа с обнаружением несущей и предупреждением коллизий (Carrier Sense Multiple Access/Collision Avoidance — CSMA/CA, в котором узел-источник отправляет кадр, узел-получатель отвечает кадром с подтверждением (ACK). В худшем случае такой метод разрешения коллизий может занять до 50 % всей пропускной способности канала. С учетом накладных расходов на протокол предупреждения коллизий реальная скорость передачи данных в канале 802.11b с пропускной способностью 11 Мбит не превышает 5...5,5 Мбит/с.

Методы множественного доступа ALOHA. Система состояла из наземных радиостанций, работающих на одной частоте и связывающих острова между собой. Идея ее конструкции заключалась в том, чтобы позволить в вещательной среде любому количеству пользователей неконтролируемо использовать один и тот же канал.

Чистая ALOHA: любой пользователь, желающий передать сообщение, сразу пытается это сделать. Благодаря тому, что в вещательной среде у него всегда есть обратная связь, т.е. он может определить, пытался ли кто-то еще передавать сообщение на его частоте, отправитель может установить возникновение конфликта при передаче. Обратная связь в среде ЛВС происходит практически мгновенно. Отправитель при этом должен слушать среду передачи до тех пор, пока последний бит его сообщения не достигнет самого отдаленного получателя. Обнаружив конфликт, отправитель ожидает некоторый случайный отрезок времени, после чего повторяет попытку

передачи. Интервал времени на ожидание должен быть случайным, иначе конкуренты, повторяя попытки передачи вызовут коллизию снова. Системы, в которых пользователи конкурируют за получение доступа к общему каналу, называются **системами с состязаниями**.

Неважно, когда произошел конфликт, оба кадра считаются испорченными и должны быть переданы повторно. Контрольная сумма, защищающая данные в кадре, не позволяет различать разные случаи наложения кадров.

Назовем **временем кадра** время, необходимое на передачу кадра стандартной фиксированной длины. Обозначим это время t . Предположим, что число пользователей неограниченно и все они порождают кадры по закону Пуассона со средним числом N кадров за t . Это означает, что вероятность события, при котором будет порождено n кадров за время t , можно записать в виде:

$$P[k] = \frac{G^k e^{-G}}{k!}$$

$$P[n] = \frac{\lambda^n e^{-\lambda}}{n!}, \lambda = N$$

Поскольку при $N > 1$ очередь на передачу будет только расти и все кадры будут страдать от коллизий, предположим, что $0 < N < 1$. Также предположим, что вероятность за время кадра сделать k попыток передачи, как новых, так и ранее не переданных из-за коллизий кадров, распределяется по закону Пуассона со средним значением G . Понятно, что при этом должно выполняться соотношение $G \leq 1$, иначе очередь будет расти бесконечно. При слабой загрузке ($G \sim 0$) будет мало передач, а, следовательно, и коллизий, поэтому $G \approx N$. При высокой загрузке должно выполняться соотношение $G > N$. При этом пропускная способность канала (S) будет равна числу кадров, которые надо передать, умноженному на вероятность успешной передачи. Если обозначить P_0 вероятность отсутствия коллизий при передаче кадра, то можно записать $S = P_0$. Рассмотрим внимательно, сколько времени требуется отправителю, чтобы обнаружить коллизию. Пусть он начал передачу в момент времени t_0 и пусть требуется время t , чтобы кадр достиг самой отдаленной станции. Тогда, если в тот момент, когда кадр почти достиг этой отдаленной станции, она начнет передачу (ведь в системе ALOHA станция сначала передает, а потом слушает), отправитель узнает об этом только через время, равное $t_0 + 2t$. Вероятность появления k кадров при передаче кадра с распределением Пуассона поэтому вероятность, что появится 0 кадров, равна P_0 . За двойное время кадра среднее число кадров равно $2G$, откуда $P_0 = e^{-2G}$, а так как $S = P_0$ то пропускная способность канала $S = e^{-2G}$.

Максимальная пропускная способность достигается при $G = 0,5$ и при $S = \frac{1}{2e}$, что составляет примерно 18% номинальной пропускной способности системы.

Слотированная ALOHA.

Модификация чистой ALOHA, в которой все время работы канала разделяется на слоты. Размер слота при этом должен быть равен максимальному времени кадра. Ясно, что такая организация работы канала требует синхронизации. Кто-то, например одна из станций, испускает сигнал начала очередного слота. Поскольку передачу теперь можно начинать не в любой момент, а только по специальному сигналу, то время на обнаружение коллизии сокращается вдвое. Откуда $S = 2P_0$. Максимум пропускной способности слотированной ALOHA наступает при $G = 1$, где $S = S_{max} = \frac{1}{e}$, т.е. составляет около 37%, что вдвое больше, чем у чистой ALOHA. Рассмотрим как G влияет на пропускную способность системы. Для этого подсчитаем вероятность успешной передачи кадра за k попыток. Так как P_0 – это вероятность отсутствия коллизии при передаче, то вероятность того, что кадр будет передан ровно за k попыток, можно записать в виде $P_k = e^{-G} (1 - e^{-G})^{k-1}$. Среднее ожидаемое число повторных передач: $E = \sum_{k=1}^{\infty} k P_k = \sum_{k=1}^{\infty} k e^{-G} (1 - e^{-G})^{k-1}$. Эта экспоненциальная зависимость показывает, что с ростом G резко возрастает число повторных попыток, поэтому незначительное увеличение загрузки канала ведет к резкому падению его пропускной способности.

18. Коммутация пакетов, устройство пакетов. Как устроен и работает пакетный коммутатор (switch). Виды задержек в компьютерной сети и способы управления ими (приоритеты, веса и гарантированная скорость потока). Управление потоком при пакетной коммутации.

- Пакет – единица данных, несущая достаточно информации, чтобы быть доставленной к месту назначения
- Коммутация пакетов: для каждого поступающего пакета независимо выбирается канал для отправки. Если канал свободен, то пакет отправляют, если нет – буферизуют для более

поздней отправки

В компьютерных сетях пакет — это определённым образом оформленный блок данных, передаваемый по сети в пакетном режиме

Компьютерные линии связи, которые не поддерживают пакетный режим, как, например, традиционная телекоммуникационная связь **точка-точка**, передают данные просто в виде последовательности **байтов**, **символов** или **битов** поодиночке. Если данные сформированы в пакеты, **битрейт** коммуникационной среды можно более эффективно распределить между пользователями, чем в **сети с коммутацией каналов**. При использовании сетей с **коммутацией пакетов** можно надёжно гарантировать пороговый битрейт, ниже которого он опускаться не будет. Сетевой пакет может состоять из стартовых бит (преамбулы), заголовков (headers), нагрузки (payload или body) и прицепа (trailer или footer). Между пакетами, посылаемыми в сеть, обычно соблюдается **межкадровый интервал** (англ. *Interframe gap*). Максимальная длина нагрузки называется **maximum transmission unit (MTU)**
Пример: IP пакет[[править](#) | [править вики-текст](#)]

IP-пакеты состоят из заголовка и полезной нагрузки. Заголовок пакета IPv4 состоит из:

1. 4 бита содержат *версию* пакета: IPv4 или IPv6.
2. 4 бита содержат *длину интернет-заголовка*, которая измеряется отрезками по 4 байта (например, 5 означает 20 байт).
3. 8 бит содержат *тип обслуживания*, известный также как качество обслуживания (**QoS**), описывающее приоритеты пакета.
4. 16 бит содержат *длину* пакета в байтах.
5. 16 бит содержат *тег идентификации*, помогающие восстановить пакет из нескольких фрагментов.
6. 3 бита содержат нуль, флаг разрешения *фрагментации* пакета (DF: не фрагментировать), а также флаг разрешения дальнейшей фрагментации (MF: фрагментировать дальше).
7. 13 бит содержат *смещение фрагмента*, поле для идентификации положение фрагмента в исходном пакете.
8. 8 бит содержат *время жизни* (TTL), которое определяет количество переходов (через маршрутизаторы, компьютеры и сетевые устройства), разрешённых

пройти пакету, прежде чем он исчезнет (например, пакету с TTL 16 разрешено пройти не более 16 маршрутизаторов, чтобы добраться до места назначения).

9.8 бит содержат *протокол* (TCP, UDP, ICMP и т. д.).

10. 16 бит содержат *контрольную сумму заголовка*, используемую при обнаружении ошибок.

11. 32 бит содержат *IP-адрес источника*.

32 бит содержат *адрес назначения*.

Сетевой коммутатор — устройство, предназначенное для соединения нескольких узлов в пределах одного или нескольких сегментов сети. Он работает на канальном уровне модели OSI. Коммутатор передаёт данные непосредственно получателю, кроме случаев широковещательного трафика и трафик для устройств, для которых неизвестен исходящий порт коммутатора. Это повышает производительность и безопасность сети, избавляя остальные сегменты сети от необходимости и возможности обрабатывать данные, которые им не предназначаются.

Принцип работы. Коммутатор хранит в ассоциативной памяти таблицу коммутации, в которой указывается соответствие MAC-адреса узла порту коммутатора. При включении коммутатора эта таблица пуста, и он работает в режиме обучения. В этом режиме поступающие на какой-либо порт данные передаются на все остальные порты коммутатора. При этом коммутатор анализирует фрейм и, определив MAC-адрес хоста-отправителя, заносит его в таблицу на определенное время. Если на один из портов коммутатора поступит фрейм, предназначенный для хоста, MAC-адрес которого уже есть в таблице, то этот фрейм будет передан только через порт, указанный в таблице. Если MAC-адрес хоста-получателя не ассоциирован с каким-либо портом коммутатора, то фрейм будет отправлен на все порты, за исключением того, с которого он был получен. Со временем коммутатор строит таблицу для всех активных MAC-адресов и в результате трафик локализуется.

Существует три способа коммутации, каждый из которых — это комбинация таких параметров, как время ожидания и надёжность передачи:

- **С промежуточным хранением.** Коммутатор читает всю информацию в кадре, проверяет его на отсутствие ошибок, выбирает порт коммутации и после этого посылает в него кадр.
- **Сквозной.** Коммутатор считывает в кадре только адрес назначения и после выполняет коммутацию. Этот режим уменьшает задержки при передаче, но в нём нет метода обнаружения ошибок.
- **Бесфрагментный или гибридный.** Этот режим является модификацией сквозного режима. Передача осуществляется после фильтрации фрагментов коллизий: первые 64 байта кадра анализируются на наличие ошибки и при её отсутствии кадр обрабатывается в сквозном режиме.

Задержка, связанная с принятием коммутатором решения, добавляется к времени, которое требуется фрейму для входа на порт коммутатора и выхода с него, и вместе с ним определяет общую задержку коммутатора.

Для временного хранения фреймов и последующей их отправки по нужному адресу коммутатор может использовать буферизацию. Буферизация может быть также использована в том случае, когда порт пункта назначения занят. Буфером называется область памяти, в которой коммутатор хранит передаваемые данные.

Буфер памяти может использовать два метода хранения и отправки фреймов: буферизация по портам и буферизация с общей памятью. При буферизации по портам пакеты хранятся в очередях, которые связаны с отдельными входными портами. При этом возможна ситуация, когда один фрейм задерживает всю очередь из-за занятости порта его пункта назначения. При буферизации в общей памяти все фреймы хранятся в общем буфере памяти, который используется всеми портами коммутатора.

По лекциям.

Lookup Address – фаза просмотра адреса.

Forwarding Table – анализ заголовка пакета с определением выходной линии.

Update Header – модификация заголовка сетевого уровня (TTL, контрольная сумма).

Ethernet коммутатор.

1. Проверяет заголовок каждого прибывающего кадра.
2. Если адрес DA есть в таблице коммутации, то кадр передают на надлежащий выходной порт.
3. Если адрес DA нет в таблице, кадр рассылается по всем портам, кроме того на который пришел.
4. Когда придет ответ на разосланный пакет, то по его адресу отправителя мы узнаем, куда надо направлять пакеты с такими адресами получателей.

Интернет маршрутизатор

1. Если Ethernet DA поступившего кадра есть Ethernet адрес маршрутизатора, то принять кадр, иначе сбросить его.
2. Просмотреть поля IP version и длина дейтаграммы
3. Сократить поле TTL(время жизни), пересчитать контрольную сумму IP заголовка
4. Проверить TTL на 0
5. Если IP DA есть в таблице маршрутизации, переслать на надлежащий выходной порт для следующего скачка (hop)
6. Найти Ethernet DA для следующего маршрутизатора
7. Построить новый Ethernet кадр и отправить его

Поиск адреса: Ethernet

Таблица Ethernet адресов

Ethernet DA	Действие
0xA8B72340E678	Передать на порт 7
0xB3D22571053B	Передать на порт 3
.....	

Адреса хранятся в хэш-таблице

Ищем в хэш-таблице точное совпадение

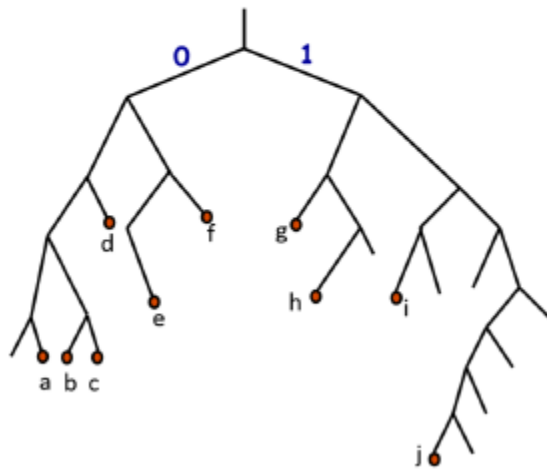
Таблица IP адресов в маршрутизаторе

IP DA	Действие
127.43.57.99	Передать на 56.99.32.16
123.66.44.x	Передать на 22.45.21.126
76.9.x.x	Передать на 56ю99ю32ю16
.....	

Ищут **совпадение по самому длинному префиксу**,
а не точное совпадение.

Поиск совпадения самого длинного префикса

Entry	Prefix
a	00001
b	00010
c	00011
d	001
e	0101
f	011
g	100
h	1010
i	1100
j	11110000



Поиск адреса <совпадение, действие>

Совпадение	Действие
IP DA = X	Передать на порт 7
EthDA=Y & IP DA = Z	Сброс пакета

Обобщение поиска и коммутации в коммутаторах, маршрутизаторах и т.п.

При буферизации на входе один или несколько буферов связываются с каждым входным портом в форме очереди типа FIFO («первым вошел, первым вышел»). Если пакет в начале очереди нельзя передать по причине занятости нужного выходного порта, этот пакет просто ждет своей очереди.

Однако если пакет ожидает, когда освободится выходной порт, то пакет, идущий за ним, тоже не может передаваться, даже если нужный ему порт свободен.

Проблему можно устранить с помощью буферизации на выходе. В этой системе буферы связаны с выходными портами. Биты пакета по мере пребывания сохраняются в буфере, который связан с нужным выходным портом. Поэтому пакеты, направленные в порт т, не могут блокировать пакеты, направленные в порт п.

Организация очередей на входе.

Примером такого варианта размещения может служить организация буферов на входных портах неблокирующей структуры с пространственным разделением. К его недостаткам можно отнести опасность возникновения блокировки в начале очереди. Если две одновременно поступившие ячейки направляются на один и тот же выходной порт, одна из них попадет во входной буфер и будет препятствовать прохождению следующих за ней ячеек, снижая тем самым пропускную способность коммутатора. Решением проблемы является значительное увеличение производительности коммутационного поля с пространственным разделением или замена дисциплины «пришедший первым обслуживается первым» (FIFO) на другую, скажем «пришедший первым обслуживается в случайном порядке» (FIFO).

Организация очередей на выходе.

Этот тип буферизации используется в выходных портах структуры с разделяемой шиной. Он оптимален с точки зрения производительности и задержек, но требует применения дополнительных средств для организации одновременной множественной доставки ячеек на любой выходной порт. Таким образом, либо выходные буферы должны функционировать с достаточно высокой скоростью, либо на каждом выходном порте требуются несколько буферов. Оба решения ограничивают производительность и масштабируемость устройства: в первом случае — из-за необходимости существенно повысить внутреннее быстродействие коммутатора, а во втором — буферов.

Решение проблемы - Виртуальный очереди на выходе(VOQ)
Каждый выход имеет n выходных буферов на порт, чтобы разместить максимальное количество одновременных пакетов данных, которые каждый порт может получить за один раз. Этот механизм буферизации Удаляет узкое место на портах в течении пикового времени и распределяет его в течение определенного периода времени повышая производительность.

Пакетный коммутатор: заключение

- Пакетный коммутатор выполняет две базовые операции:
 - Поиск соответствия в таблице коммутации
 - Передачу на надлежащий выходной порт
- Самый простой и самый медленный коммутатор использует буферизацию на выходе с минимальной задержкой пакета

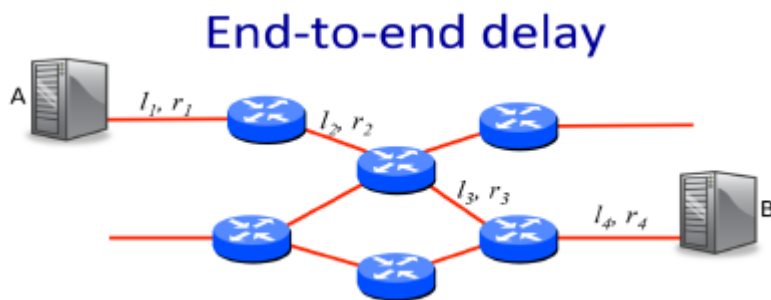
- Высокоскоростные коммутаторы используют буферизацию на входе с виртуальными очередями на выходах для увеличения пропускной способности.

Виды задержек:

- **Задержка распространения** - время распространения одного бита по каналу со скоростью c $t = l/c$ (l - расстояние от точки до точки)

Задержка пакетизации - время, за которое все биты с первого до последнего будут поданы в канал. $t = p/r$ (p - величина пакета, r - пропускная способность канала)

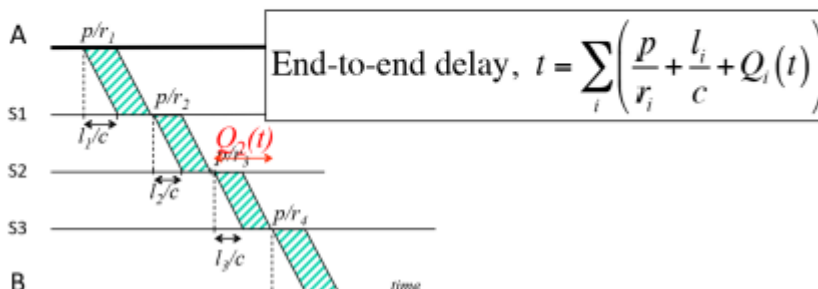
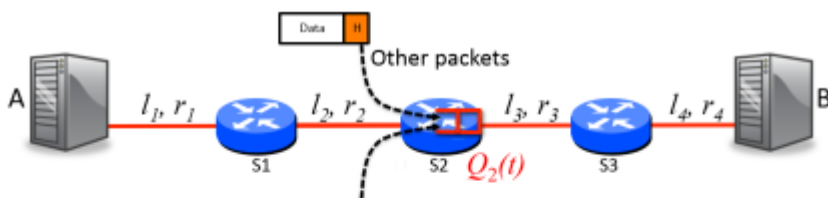
Задержка в очереди в буфере маршрутизатора - задержка от момента поступления пакета в очередь до момента его отправки



Пример: За какое время пакет длины p дойдет от A до B (от момента когда послан первый бит до момента когда придет последний бит). Предполагаем что коммутаторы принимают и сразу передают пакет

$$t = \sum_i \left(\frac{p}{r_i} + \frac{l_i}{c} \right)$$

Задержка в очереди:



Взвешенная справедливая очередь (англ. Weighted Fair Queuing, WFQ) - механизм планирования пакетных потоков данных с различными приоритетами. Его целью является регулировать использования одного канала передачи данных несколькими конкурирующими потоками. В данном случае под потоком понимается очередь пакетов данных.

Это обобщение алгоритма честных планировщиков (англ. Fair Queuing) (FQ). Оба планировщика имеют отдельные FIFO-очереди для каждого потока данных. Так, если канал со скоростью R используется для N потоков, то скорость обработки каждого из них будет R/N при использовании честного планировщика. Честный планировщик с приоритетными коэффициентами позволяет регулировать долю каждого потока. Если имеется N активных потоков, с приоритетами $w_1, w_2 \dots w_N$, то i -ый поток будет иметь скорость

$$\frac{Rw_i}{(w_1 + w_2 + \dots + w_N)}.$$

FIFO очередь – нет приоритетов, не гарантирована скорость

- Строгие приоритеты: высокоприоритетный трафик «не видит» низкоприоритетного трафика в сети.
Полезно, когда высокоприоритетного трафика ограниченное количество
- Waited Fair Queuing (WFQ) позволяет каждому потоку обеспечить гарантированный сервис, планируя их в порядке bit-by-bit finishing time

$$\text{е2е задержка, } \tau = \sum_i \left(\frac{p}{r_i} + \frac{l_i}{c} + Q_i(t) \right)$$

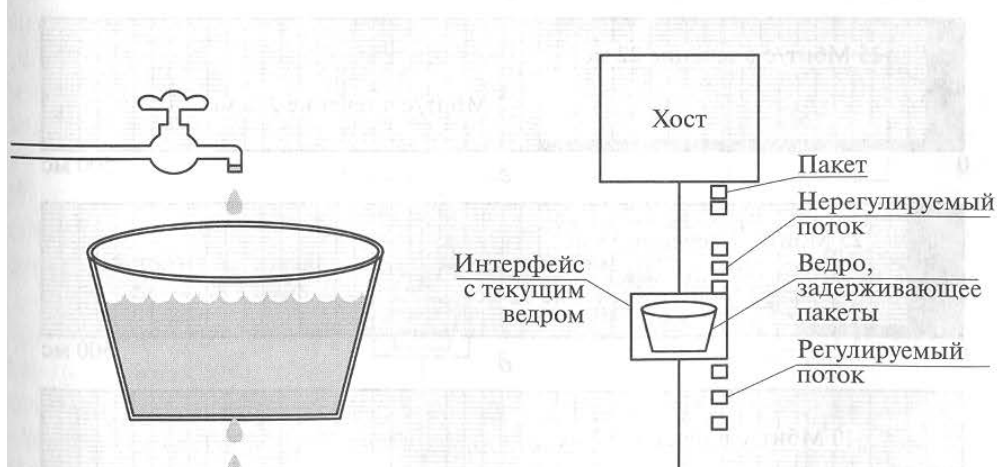
- Если мы знаем длину очереди и дисциплину ее обслуживания, то мы можем ограничить величину задержки в ней.
- Выбрав длину очереди, и, используя WFQ(взвешенная справедливая очередь), мы можем определить скорость обслуживания.

- Поэтому самое главное не допустить сброса пакетов. Для этого можно использовать текущий буфер.
- Таким образом, мы можем ограничить величину e_2e задержки.

Алгоритм текущего ведра

Идею алгоритма текущего ведра иллюстрирует рис. 2.15, а [19]: ведро может наполняться с любой скоростью, но вытекать из него вода будет со строго определенной скоростью, зависящей только от размера отверстия в днище. Если вода будет поступать слишком быстро, то часть ее будет переливаться через край и пропадать.

Этот прием можно применить и к пакетам (рис. 2.15, б). Каждая станция, подключенная к сети, имеет в своем интерфейсе буфер, по-

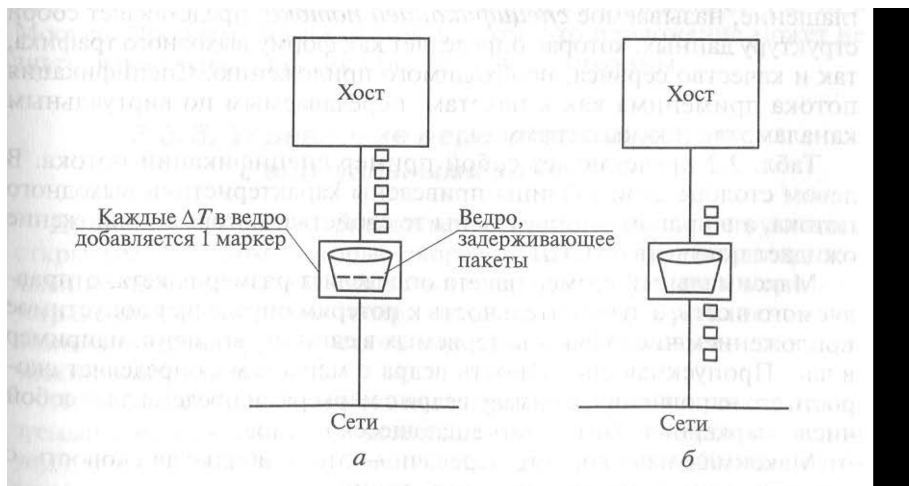


добный текущему ведру. Независимо, сколько процессов посылают пакеты в сеть, если буфер переполнен, то они будут сбрасываться в соответствии с соглашением о трафике.

Алгоритм ведра с маркерами

Алгоритм текущего ведра позволяет сгладить трафик, убрать нерегулярность передачи. Однако в целом ряде приложений бывает полезно при наличии всплеска трафика и необходимых ресурсов разрешить ускорить на некоторое время передачу пакетов в сеть. Одним из алгоритмов, позволяющих это сделать, является алгоритм ведра с маркерами, который иллюстрирует рис. 2.17 [19, 59].

Идея этого алгоритма заключается в том, что вместе с пакетами в ведро поступают маркеры (рис. 2.17, а) и пакеты уходят из ведра в сеть только при наличии соответствующего числа маркеров (рис. 2.17, б). Таким образом можно накапливать маркеры и кратковременно ускорять передачу пакетов в сеть.



Особенность алгоритма ведра с маркером заключается в том, что при переполнении буфера маршрутизатору будет временно запрещено передавать пакеты. Существуют разные варианты этого алгоритма, зависящие от длины пакетов, правил работы со счетчиком маркеров и т.д.

Для реализации алгоритма ведра с маркерами необходима лишь переменная, значение которой увеличивается каждые ΔT секунд и уменьшается с каждым посланным пакетом. В случае использования пакетов с нефиксированной длиной значение этой переменной увеличивается на k байт каждые ΔT секунд и уменьшается на длину каждого посланного пакета. Нетрудно рассчитать длительность всплеска трафика при передаче, осуществляемой на основе уравнения

$$C + \rho S = MS,$$

где C — объем буфера; ρ — скорость поступления маркеров; S — длительность всплеска трафика; M — максимальная скорость «вытекания» данных.

Таким образом, $S = C/(M - \rho)$.

Примеры, приведенные на рис. 2.16, *г...е*, иллюстрируют эту функцию для случая, когда буфер $C = 250$ Кбит, $M = 25$ Мбит/с и $\rho = 2$ Мбит/с.

Недостатком алгоритма маркерного ведра является то, что скорость передачи крупных пачек снижается до постоянного значения R . Часто бывает желательно уменьшить пиковую скорость, не возвращаясь при этом к постоянному значению скорости (но и не увеличивая это значение, чтобы не пропустить в сеть дополнительный трафик).

Управление потоком при пакетной коммутации

В компьютерных сетях неизбежны потери пакетов данных, в частности, из-за переполнения буферной памяти хотя бы одного из узлов, расположенных на пути от источника к приёмнику, включая последний. Такие потери, связанные с переполнениями, в дальнейшем именуется **перегрузками узлов сети**.

Существует множество способов предотвращения и устранения перегрузок; эти способы, в большинстве своем, основаны на управлении потоками данных. Особое место занимает обслуживание пакетов с учетом их приоритетов.

Способ 1. Управление потоком данных регулировкой длительности пауз между пакетами.

Прототип. В процессе передачи данных приемник замечает устойчивую нехватку прибывающих пакетов (например, отслеживая их порядковые номера) и посылает источнику данных управляющий пакет, содержащий команду XOFF приостановки потока данных. Адрес источника данных известен приемнику, так как поступающие к нему пакеты данных содержат информацию об адресах отправителя и адресата. При этом также посылаются запросы на повторную передачу потерянных пакетов.

Получив команду XOFF, источник данных полностью прекращает передачу пакетов и возобновляет ее либо через некоторое время, оговоренное в протоколе обмена данными, либо после получения от приемника команды XON возобновления передачи.

Недостатки:

- 1) Поток либо есть, либо его нет. Запаздывание выполнения команд может привести к неоправданному простаиванию передатчика и периодическому возникновению новых перегрузок, при которых некоторая часть пакетов, в том числе, принадлежащих другим потокам, будет утеряна.
- 2) При длительной перегрузке приемник пересылает передатчику серию одинаковых команд приостановки потока, что засоряет канал связи большим количеством повторяющихся служебных пакетов.
- 3) Команды приостановки работы передатчика формируются приемником только в том случае, когда число отвергнутых в связи с переполнением буфера пакетов достаточно велико.
- 4) Приостановки работы передатчика увеличивают среднюю и максимальную задержки передачи пакетов по трассе, что может снизить заданные в контракте между пользователем и провайдером показатели качества обслуживания канала.

Решение. Предлагаемое решение в значительной степени устраняет перечисленные недостатки благодаря плавной и «опережающей события» регулировке скорости передачи данных источником. Скорость регулируется изменением длительности пауз между пакетами: чем больше паузы, тем ниже скорость передачи данных, и наоборот. Отметим, что наличие паузы не означает, что сигнал в линии связи отсутствует – сигнал присутствует постоянно, но в нем нет флаговых кодов, обозначающих начало пакета, либо наоборот – передается непрерывный поток этих кодов.

Способ 2. Управление потоком данных оповещением источника пакетов о причинах перегрузки.

Прототип. В ответ на каждый пакет или на группу пакетов приемник посылает источнику ответные пакеты ACK или NACK. Ответ ACK подтверждает успешный прием, ответ NACK служит запросом повторной передачи одного пакета или группы пакетов. Если источник данных при повышении скорости передачи пакетов или на некоторой фиксированной скорости начинает получать чрезмерное число запросов повторной передачи, то, вероятнее всего, по крайней мере, один из узлов трассы вошел в режим перегрузки. В этом случае источник данных резко снижает скорость передачи пакетов или (и) увеличивает их длину, чтобы уменьшить долю передаваемых в потоке данных служебных битов, образующих заголовки. В дальнейшем источник данных постепенно либо методом случайных проб и ошибок повышает скорость передачи данных, продвигаясь к допустимой верхней границе с учетом некоторого разрешенного запаса повышения скорости. Такой способ называют «медленным стартом».

Рассмотренный способ управления потоком данных не предотвращает предстоящую потерю пакетов, а позволяет реагировать только на свершившийся факт перегрузки промежуточного узла сети или приемника данных. В этом состоит его **основной недостаток**.

Другой прототип. Идея состоит в том, чтобы вовремя предупредить источник данных А об угрозе перегрузки одного или нескольких узлов вдоль трассы АВ распространения пакетов данных D. Таким предупреждением служит бит Z, входящий в состав заголовка ответного пакета ACK или NACK.

Решение. Поставленная задача решается расширением одноразрядного признака Z до нескольких битов.

Узел может испытывать перегрузку по крайней мере по одной из следующих причин:

- 1) Сужение полосы (пропускной способности) канала АВ из-за появления «узкого места».
 - Увеличился до значительного уровня ранее малозаметный конкурирующий поток данных по маршруту M4M2M3, который использует тот же канал M2M3, что и маршрут АВ. В результате узел M2 перераспределил полосу этого канала в ущерб маршруту АВ.
 - Узел M2 изменил тип модуляции сигнала в канале M2M3, снизив скорость передачи в связи с ухудшением соотношения «сигнал/шум» в этом канале.
- 2) Процессор узла M2 по тем или иным причинам перестал справляться с объемом работы по анализу заголовков пакетов, следующих по маршруту АВ.

Первая и вторая причины надвигающейся перегрузки отображаются соответственно кодами $\square = 01$ и $\square = 10$, отсутствие опасности перегрузки соответствует коду $\square = 00$, обе причины одновременно вызывают формирование кода $\square = 11$.

Способ 3. Управление потоком данных с компенсацией инерционности контура обратной связи. Плавность управления достигается дроблением серий пакетов и более интеллектуальным алгоритмом формирования команд XON и XOFF возобновления и прекращения передачи потока.

19. Алгоритмы маршрутизации в Интернет: основные подходы. Структура сети Интернет, понятие автономной системы, □ протокол внешней маршрутизации BGP. Явление перегрузки и основные методы борьбы с ней. Перегрузка: AIMD в случае одного потока и в случае нескольких потоков.

Общие сведения.

Основной задачей сетевого уровня является маршрутизация пакетов. Пакеты маршрутизируются всегда, т.е. независимо от того, какую внутреннюю организацию имеет транспортная среда: с виртуальными каналами или дейтаграммную. Разница состоит лишь в том, что в первом случае этот маршрут устанавливается один раз для всех пакетов, а во втором – для каждого пакета отдельно. Первый случай называют иногда маршрутизацией сессии, поскольку маршрут устанавливается на все время передачи данных пользователя, т.е. на время сессии.

Алгоритм маршрутизации реализует программное обеспечение маршрутизатора на сетевом уровне, т.е. он отвечает за определение, по какой из линий, доступных маршрутизатору, отправлять пакет дальше. При этом независимо от выбора маршрута (для сессии или для каждого пакета в отдельности) алгоритм маршрутизации должен обладать следующими свойствами: корректностью, простотой, устойчивостью, стабильностью, справедливостью и оптимальностью.

- 1) Корректность – свойство алгоритма маршрутизации, определяющее, что при любых обстоятельствах этот алгоритм либо найдет маршрут для доставки пакета адресату, либо выдаст сообщение о невозможности его доставки. Третьего варианта быть не может. При этом крайне желательно, чтобы алгоритм также сообщил о причинах невозможности доставки пакета.
- 2) Простота – свойство, определяющее вычислительную сложность алгоритма маршрутизации: чем она меньше, тем алгоритм проще, и тем меньше ресурсов маршрутизатора тратится на решение задачи маршрутизации.
- 3) Устойчивость – свойство алгоритма маршрутизации сохранять работоспособность независимо от каких-либо сбоев, отказов в системе передачи данных или транспортной среде, а также изменений топологии (отключения хостов или машин транспортной среды, разрушения каналов и т.п.). Алгоритм маршрутизации должен адаптироваться ко всем таким изменениям, не требуя при этом перезагрузки транспортной среды или остановки абонентских машин.
- 4) Стабильность – весьма важное свойство алгоритма маршрутизации. Существуют алгоритмы, которые никогда не приводят к какому-либо определенному маршруту, как бы долго они ни работали. Это означает, что адаптация алгоритма к изменениям в топологии или конфигурации транспортной среды может оказаться весьма продолжительной, и более того, она может оказаться сколь угодно долгой.
- 5) Справедливость – свойство, означающее, что все пакеты независимо от того, из какого канала они поступили, будут обслуживаться маршрутизатором равномерно, т.е. никакому направлению не будет отдаваться предпочтение, для всех абонентов будет всегда выбираться оптимальный маршрут.

бираться оптимальный маршрут.

Следует отметить, что справедливость и оптимальность часто могут вступать в противоречие друг с другом при неудачном выборе критерия оптимизации.

Прежде чем искать компромисс между оптимальностью и справедливостью, необходимо решить, что является критерием оптимизации маршрута. Один из возможных критериев – средняя задержка пакета (обратите внимание, что именно средняя задержка).

Другой критерий – пропускная способность транспортной среды. Однако эти критерии конфликтуют. Согласно теории массового обслуживания, если система с очередями функционирует близко к своему насыщению, то задержка в очереди увеличивается. Как компромисс во многих сетях минимизируется число переходов между маршрутизаторами. Один такой переход называется скачком, или переходом (hop). Уменьшение числа скачков сокращает маршрут, а следовательно, сокращает задержку и минимизирует необходимую пропускную способность СПД для передачи пакета.

Алгоритмы маршрутизации можно разбить на два больших класса: адаптивные и неадаптивные. Неадаптивные алгоритмы не принимают в расчет текущую загрузку сети и ее текущую топологию. Все возможные маршруты вычисляются заранее и загружаются в маршрутизаторы при загрузке сети. Такая маршрутизация называется статической.

Адаптивные алгоритмы, наоборот, определяют маршрут исходя из текущей загрузки и топологии транспортной среды. Адаптивные алгоритмы различаются способом получения информации (локально от соседних маршрутизаторов или глобально от всех маршрутизаторов), временем изменения маршрута (через каждые T секунд либо только когда изменяется нагрузка, либо когда изменяется топология) и метрикой, используемой при оптимизации (расстояние, число скачков, ожидаемое время передачи и т.н.).

Свойство оптимального пути.

Обоснуем одно важное предположение о свойстве оптимального маршрута, которое будет использоваться в дальнейших рассуждениях. Это свойство состоит в том, что если маршрутизатор J

Поскольку дерево захода – это дерево, то там нет циклов, и каждый пакет будет доставлен за конечное число шагов. На практике же все может оказаться сложнее: маршрутизаторы могут выходить из строя, могут появляться новые маршрутизаторы, каналы могут выходить из строя, разные маршрутизаторы могут узнавать об этих изменениях в разное время и т.д.

Алгоритм маршрутизации по вектору расстояния работает следующим образом: у каждого маршрутизатора в транспортной среде есть таблица расстояний до всех других маршрутизаторов, принадлежащих этой транспортной среде. Периодически каждый маршрутизатор обменивается этой информацией со своими соседями и обновляет информацию в своей таблице. Каждый элемент этой таблицы включает в себя два поля: первое – номер канала, по которому следует отправлять пакеты, чтобы достичь нужного места, второе – значение задержки до места назначения, которое может измеряться в разных единицах: числе скачков, миллисекундах, длине очереди к каналу и т.д. Фактически в протоколе использовалась версия алгоритма, где эту задержку определяли не на основе пропускной способности канала, а на основе длины очереди к каналу.

Каждые T секунд маршрутизатор шлет своим соседям свой вектор задержек до всех маршрутизаторов в транспортной среде. В свою очередь, он получает такие же векторы от своих соседей. Кроме

того, он постоянно замеряет задержки до своих соседей, следовательно, имея векторы расстояний от соседей и зная расстояние до них, маршрутизатор всегда может вычислить наикратчайший маршрут до определенного места в транспортной среде.

На рис. 2.5 приведен пример маршрутизации по вектору расстояния.

Рассмотрим, как маршрутизатор J с помощью итоговой таблицы маршрутизации вычислит маршрут до G . Маршрутизатор J знает, что он может достичь A за 8 мс, маршрутизатор A объявляет, что от него до G 18 мс. Таким образом, J может достичь G за 26 мс через A . Аналогично можно подсчитать, что достичь G через I , H и K можно соответственно за 41 ($31 + 10$), 18 ($6 + 12$) и 37 ($31 + 6$) мс. Наилучшее полученное значение – 18, следовательно, этот маршрут и является наилучшим по критерию скорости доставки.

6

14

Проблема бесконечного счетчика задержки.

Алгоритм маршрутизации по вектору расстояния теоретически работает хорошо, но у него есть один недостаток: он очень медленно реагирует на разрушения каналов в транспортной среде. Информация о появлении хорошего маршрута в транспортной среде распространяется более или менее быстро, а вот данные о потере, разрушении какого-то маршрута распространяются значительно медленнее.

Рассмотрим пример транспортной среды с линейной топологией. Показанный на рис. 2.6, а. Пусть изначально маршрутизатор А не работал, поэтому у всех маршрутизаторов расстояние до него было равно ∞ . Пусть в какой-то момент времени А включился. По истечении определенного времени маршрутизаторы начнут обмениваться векторами, и В узнает об А. Еще через один обмен векторами об А узнает С и т.д. Таким образом, информация о новом маршруте будет распространяться линейно шаг за шагом при каждом обмене векторами. Если самый длинный маршрут в транспортной среде имеет длину N, то потребуется N обменов векторами, пока информация о новом маршруте дойдет до самого удаленного узла.

Теперь рассмотрим обратную ситуацию, показанную на рис. 2.6, б, т.е. когда изначально все маршрутизаторы и каналы работоспособны.

A	B	C	D	E	
∞	∞	∞	∞	∞	Исходное состояние
1	∞	∞	∞	∞	После 1-го обмена
1	2	∞	∞	∞	После двух обменов
1	2	3	∞	∞	После трех обменов
1	2	3	4	∞	После четырех обменов
а					
A	B	C	D	E	
1	2	3	4	4	Исходное состояние
3	2	3	4	4	После 1-го обмена
4	4	3	4	4	После двух обменов
5	5	5	4	4	После трех обменов
5	6	5	6	6	После четырех обменов
7	6	7	6	6	После пяти обменов
7	6	7	8	8	После шести обменов
∞	∞	∞	∞	∞	
б					

Рис. 2.6. Пример топологии транспортной среды с изначально не работающим маршрутизатором А (а) и с полностью работоспособными маршрутизаторами и каналами (б)

Пусть в какой-то момент времени канал между А и В оказался разрушен. В этом случае В перестает видеть А, но С говорит В: «У меня есть маршрут до А». При этом В не подозревает, что маршрут от С до А идет через него же. Маршрутизаторы D и E своих таблиц не изменяют. Их расстояния до А на единицу больше, чем у С. При втором обмене С увидит, что оба его соседа достигают А за 3 единицы. Некоторым случайным образом С выбирает одного соседа, увеличив значение 3 на единицу. Плохая

весть будет распространяться медленно, пока счетчики задержек не примут некоторого очень большого значения, практически бесконечного для данной сети. Только после этого станет ясно, что А недостижим ни через С, ни через D, ни через E. Сколько времени на это потребуется зависит от конкретного значения бесконечности в данной транспортной среде.

Идею алгоритма маршрутизации по состоянию канала можно описать в виде пяти основных шагов, которые должен выполнить каждый маршрутизатор в транспортной среде:

1. Определить своих непосредственных соседей и их сетевые адреса.

2. Измерить задержку или оценить затраты на передачу до каждого соседа.
3. Сформировать пакет, где укатаны все данные, полученные на шаге 2.
4. Послать этот пакет всем другим маршрутизаторам.
5. Вычислить наикратчайший маршрут до каждого маршрутизатора.

Как видно из приведенного описания, топология транспортной среды и все задержки в СПД оцениваются всеми узлами экспериментально. После этого можно использовать, например, алгоритм Дейкстры для вычисления наикратчайшего маршрута. Теперь рассмотрим подробнее эти пять шагов.

Определение соседей.

При загрузке маршрутизатор, прежде всего, определяет, кто его непосредственные соседи. Для этого он рассылает по всем подсоединенным к нему каналам специальный пакет HELLO, и все маршрутизаторы отвечают, указывая свое уникальное имя. Имя маршрутизатора должно быть уникальным в сети, чтобы избежать неоднозначностей.

Если же два и более маршрутизаторов соединены каналом с множественным доступом, как на рис. 2.8 а, то этот канал в графе связей представляют отдельным искусственным узлом (рис. 2.8, б).

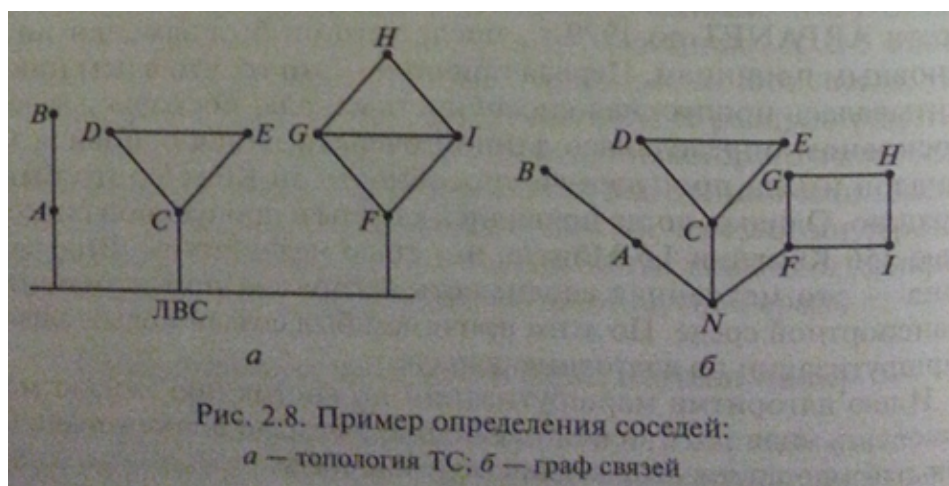
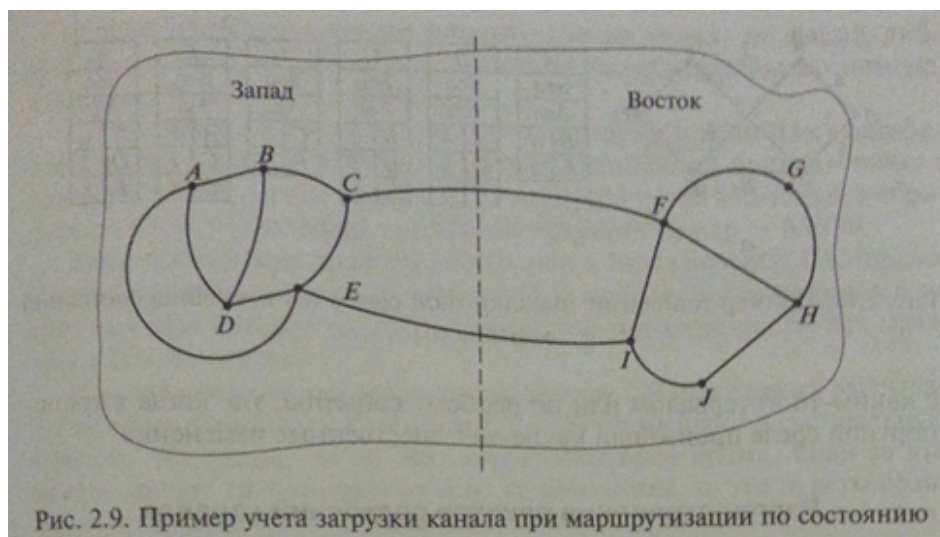


Рис. 2.8. Пример определения соседей:
а — топология ТС; б — граф связей

Оценка затрат.

Оценка затрат до каждого соседа происходит с помощью другого специального пакета ЕСНО, который рассылается **всем соседям**, при этом замеряется задержка от момента отправки этого пакета до момента его возвращения. Все маршрутизаторы, которые получают такой пакет, обязаны отвечать незамедлительно, отправляя пакет обратно. Такие замеры делают несколько раз и вычисляют их среднее значение. Таким образом, длина очереди к каналу не учитывается.

Здесь необходимо понять: следует учитывать загрузку канала или нет? Если учитывать, то задержку надо замерять от момента поступления пакета в очередь к каналу, а если не учитывать, то — от момента когда пакет достиг головы очереди. Есть доводы и за учет загрузки, и против ее учета. При учете загрузки можно выбирать между двумя и более каналами с одинаковой пропускной способностью, обеспечивая лучшую производительность.

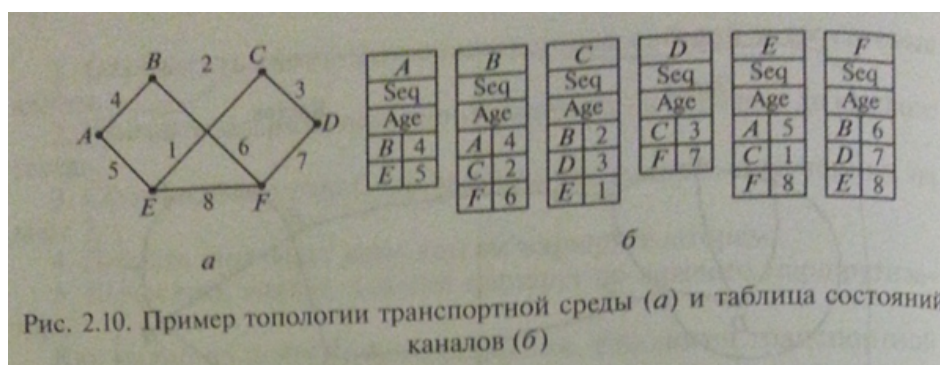


Однако можно привести примеры, когда учет загрузки вызывает проблемы. Например, рассмотрим рис. 2.9, где два одинаковых по производительности канала CF и EI соединяют две транспортные среды. Когда загрузка одного из этих каналов меньше, то он предпочтительнее другого, что через некоторое время приводит к его перегрузке, и предпочтительнее становится другой канал. Если загрузка не учитывается, то такой проблемы не возникает.

Формирование пакетов состояний каналов.

После выполнения всех указанных измерений можно формировать пакеты состояний каналов.

На рис. 2.10, б показаны пакеты состояний каналов для примера транспортной среды, приведенной на рис. 2.10, а.



В пакетах состояний указываются: отправитель, последовательное число, возраст (назначение этих полей станет ясно позднее), список соседей и задержки до них. Формирование таких пакетов не вызывает проблем. Основной вопрос здесь, когда их формировать: периодически с каким-то интервалом или по особому событию, т.е. когда в транспортной среде произошли какие-то существенные изменения?

Распространение пакетов состояния каналов.

Наиболее хитрая часть этого алгоритма – надежное распространение пакетов состояний каналов (далее СК-пакетов). Как только СК-пакет получен и включен в работу, маршрутизатор начинает использовать его при определении маршрута. При неудачном распространении СК-пакетов разные маршрутизаторы могут получить разное представление о топологии транспортной среды, что может привести к возникновению циклов, появлению недостижимых машин и другим проблемам.

Рассмотрим сначала базовый алгоритм, в котором СК-пакеты распространяются методом лавины, т.е. СК-пакет рассылается всем соседям, которые, в свою очередь, пересылают его своим соседям и т.д. Однако чтобы не потерять контроль и не вызвать неограниченного дублирования СК-пакетов, каждый маршрутизатор ведет счетчик последовательных номеров СК-пакетов, которые он сгенерировал. Все маршрутизаторы запоминают пары маршрутизатор – последовательное число, которые они уже встречали среди полученных СК-пакетов. Если поступивший СК-пакет содержит пару, которая еще не встречалась маршрутизатору, то он отправляет этот СК-пакет всем своим соседям, за исключением того соседа, от которого он его получил. Если маршрутизатор уже встречал такой пакет, то этот пакет сбрасывается и нигде не дублируется.

В целях сокращения числа рассылаемых СК-пакетов их рассылают не сразу. Сначала полученный СК-пакет помещают в специальную область задержки, где он находится некоторое время. Если за это время придет другой пакет от того же источника, то эти пакеты сравниваются, и если они одинаковые, то вновь пришедший пакет сбрасывается, если же они различаются, то последний пришедший пакет дублируется и отправляется другим маршрутизаторам, а первый пакет сбрасывается. Все СК-пакеты перепекаются с ввеломлением.

Вычисление нового пути.

Когда маршрутизатор получил полный комплекс СК-пакетов, он может построить топологию транспортной среды и, например, локально запустить алгоритм Дейкстры для вычисления наикратчайшего пути.

В системе, где есть n маршрутизаторов, имеющих по k линий, у каждого из этих маршрутизаторов должно быть достаточно памяти для хранения необходимой информации о сети. При больших значениях n этот объем памяти может стать существенным. Кроме того, в сетях, где число маршрутизаторов достигает десятков или сотен тысяч, проблемы, вызванные сбоем одного из них, могут оказаться весьма серьезными. Например, в случае если из-за сбоя маршрутизатор послал неправильный СК-пакет или неправильно оценил состояние канала, в дальнейшем это может создать большие проблемы.

Маршрутизация по соединяющему дереву (протокол STP).

Основное предназначение STP (Spanning Tree Protocol) – удаление петель в топологии сети и автоматическое управление топологией сети с дублирующими каналами. Действительно, если сетевое оборудование связано для надежности избыточным числом соединений, то без принятия дополнительных мер, кадры будут доставляться получателю в нескольких экземплярах, что приведет к сбоям и возникновению широковещательных штормов (broadcast storm) в случае отправки широковещательных кадров. Следовательно, в каждый момент времени должен быть задействован только один из параллельных каналов, но при этом необходимо иметь возможность переключения при отказах или физическом изменении топологии. С этой задачей может вручную справиться администратор, однако более элегантным и экономичным решением, освобождающим от необходимости круглосуточного мониторинга состояния системы человеком, является использование STP.

Для своей работы STP строит граф, называемый также «деревом», создание которого начинается с корня (root). Корнем становится одно из STP-совместимых устройств, выигравшее выборы. Каждое STP-совместимое устройство (это может быть коммутатор, маршрутизатор или другое оборудование, но для простоты далее мы будем называть такое устройство мостом) при включении считает, что оно является корнем. При этом оно периодически посылает на все свои порты специальные блоки данных – Bridge Protocol Data Units (BPDU). Адрес получателя в пакетах, несущих BPDU, является групповым, что обеспечивает его пропуск неинтеллектуальным оборудованием.

В данном случае под адресом понимается MAC-адрес, так как протокол STP функционирует на уровне управления доступом к среде передачи (Media Access Control, MAC). Из этого также следует, что все дальнейшие рассуждения о STP и его уязвимостях не привязаны к какому-то одному методу передачи, т. е. в равной мере относятся к Ethernet, Token Ring и т. д.

Получив очередной BPDU от другого устройства, мост сравнивает полученные параметры со своими и, в зависимости от результата, перестает или продолжает оспаривать статус корня. В результате корнем становится устройство с наименьшим значением идентификатора моста (Bridge ID). Последний представляет собой комбинацию MAC-адреса и заданного для моста приоритета. Очевидно, что в сети с единственным STP-совместимым устройством оно и будет корнем.

Выбранный корень, или назначенный корневой мост (Designated Root Bridge, в соответствии с терминологией стандарта), не несет никакой дополнительной нагрузки – он всего лишь служит отправной точкой для построения топологии.

Для всех остальных мостов в сети определяется корневой порт (Root Port), т.е. ближайший к корневому мосту порт. От других портов, соединенных с корневым мостом непосредственно или через другие мосты, он отличается своим идентификатором – комбинации из его номера и задаваемым администратором «веса».

На процесс выборов влияет и стоимость пути до корня (Root Path Cost) – она складывается из стоимости пути до корневого порта данного моста и стоимости путей до корневых портов мостов по всему маршруту до корневого моста. Эта стоимость может определяться временем передачи по линку, а также стоимостью портов, через которые происходит передача. Стоимость портов может задаваться вручную администратором.

Помимо выделенного корневого моста в STP вводится логическое понятие назначенного моста (Designated Bridge) – владелец этого статуса считается главным в обслуживании данного сегмента локальной сети. Статус назначенного моста также выборный и может переходить от одного устройства к другому.

Аналогичным образом вводится логическое понятие выборного назначенного порта (Designated Port, он обслуживает данный сегмент сети), а для него – понятие соответствующей стоимости пути (Designated Cost).

После окончания всех выборов наступает фаза стабильности, характеризующаяся следующими условиями.

1. В сети только одно устройство считает себя корнем, а остальные периодически анонсируют его как корень.
2. Корневой мост регулярно посылает на все свои порты пакеты с BPDU. Интервал времени, через который происходит рассылка, называется интервалом приветствия (Hello Time).
3. В каждом сегменте сети имеется единственный назначенный порт, через который происходит обмен трафиком с корневым мостом. Он имеет наименьшее значение стоимости пути до корня по сравнению с другими портами в сегменте. При равенстве этой величины в качестве назначенного выбирается порт с наименьшим идентификатором порта (MAC-адрес порта и его приоритет).
4. BPDU принимаются и отправляются STP-совместимым устройством на всех его портах, даже на тех, которые были «отключены» в результате работы STP. Однако BPDU не принимаются на портах, которые были «отключены» администратором.
5. Каждый мост осуществляет пересылку (Forwarding) пакетов только между корневым портом и назначенными портами соответствующих сегментов. Все остальные находятся в заблокированном состоянии (Blocking).

Как следует из последнего пункта, STP управляет топологией путем изменения состояния портов, которое может принимать следующие значения:

- **Блокирован (Blocking).** Порт заблокирован, однако, в отличие от пользовательских кадров, кадры с пакетами STP (BPDU) принимаются и обрабатываются;
- **Ожидает (Listening).** Первый этап подготовки к состоянию пересылки. В отличие от пользовательских кадров, кадры с пакетами STP (BPDU) принимаются и обрабатываются. Обучения не происходит, так как в этот период в таблицу коммутации может попасть недостоверная информация;
- **Обучается (Learning).** Второй этап подготовки к состоянию пересылки. Кадры с пакетами STP (BPDU) принимаются и обрабатываются, а пользовательские кадры мост принимает для построения таблицы коммутации, но не пересылает данные;
- **Передает (Forwarding).** Рабочее состояние портов, когда передаются как кадры с пакетами STP, так и кадры пользовательских протоколов.
- **Отключен (Disabled).** Порт не предпринимает никакой активности.

Фактически, Интернет состоит из множества локальных и глобальных сетей, принадлежащих различным компаниям и предприятиям, работающих по самым разнообразным протоколам, связанных между собой различными линиями связи, физически передающих данные по телефонным проводам, оптоволокну, через спутники и радиомодемы. Структура Интернет напоминает паутину, в узлах которой находятся компьютеры, связанные между собой линиями связи. Узлы Интернет, связанные высокоскоростными линиями связи, составляют базис Интернет. Оцифрованные данные пересылаются через маршрутизаторы, которые соединяют сети с помощью сложных алгоритмов, выбирая маршруты для информационных потоков.

Каждый компьютер в Интернет имеет свой уникальный адрес – IP-адрес. Этот номер может быть постоянно закреплен за компьютером или же присваиваться динамически – в тот момент, когда пользователь соединился с провайдером, но в любой момент времени в Интернет не существует двух компьютеров с одинаковыми IP-адресами.

Доменное имя – это уникальное имя, которое данный поставщик услуг избрал себе для идентификации. Для преобразования имени в адрес компьютер посылает запрос серверу DNS, начиная с правой части доменного имени и двигаясь влево.

Данные в Интернет пересылаются не целыми файлами, а небольшими блоками, которые называются пакетами. Каждый пакет содержит в себе адреса компьютеров отправителя и получателя, передаваемые данные и порядковый номер пакета в общем потоке данных. Благодаря тому, что каждый пакет содержит все необходимые данные, он может доставляться независимо от других, и довольно часто случается так, что пакеты добираются до места назначения разными путями. А компьютер-получатель затем выбирает из пакетов данные и собирает из них тот файл, который был заказан.

Порт – это число, которое добавляется к адресу компьютера, которое указывает на программу, для которой данные предназначены.

В Интернет используются не просто доменные имена, а универсальные указатели ресурсов URL (Universal Resource Locator).

URL включает в себя:

- метод доступа к ресурсу, т.е. протокол доступа (http, gopher, WAIS, ftp, file, telnet и др.);
- сетевой адрес ресурса (имя хост-машины и домена);
- полный путь к файлу на сервере.

Сервер в сети Интернет – это компьютер, обеспечивающий обслуживание пользователей сети: разделяемый доступ к дискам, файлам, принтеру, системе электронной почты. Обычно сервер – это совокупность аппаратного и программного обеспечения.

Сайт – обобщенное название совокупности документов в Интернет, связанных между собой ссылками.

Шлюз (gateway) – это компьютер или система компьютеров со специальным программным обеспечением, позволяющая связываться двум сетям с разными протоколами.

Домашняя страница – это персональная Web-страница конкретного пользователя или организации.

Автономная система (AS) в интернете – это система IP-сетей и маршрутизаторов, управляемых одним или несколькими операторами, имеющими единую политику маршрутизации с Интернетом.

Протокол BGP используется для передачи информации о внутренних маршрутах между автономными системами. Протокол BGP может быть использован для определения различных типов маршрутов:

- Inter-autonomous system routing – маршруты, которые соединяют данную автономную систему с одной или несколькими другими автономными системами.
- Intra-autonomous system routing – протокол может быть использован для определения маршрута внутри автономной системы, в том случае, когда несколько маршрутизаторов участвуют в процессе определения маршрута BGP.
- Pass-through autonomous system – протокол может быть использован для определения маршрутов, которые проходят через автономную систему, которая не участвует в процессе BGP.

Для обеспечения информационного обмена маршрутизаторы BGP используют сообщения стандартной формы. Для передачи этих сообщений в протоколе BGP предусматривается использование транспортного протокола TCP. Сообщения BGP передаются в следующих случаях:

- Начало сеанса (Open).
- Для периодической проверки состояния соседа (Keep Alive).
- При изменении содержания таблицы маршрутов автономной системы (Update).
- При возникновении аварийной ситуации (Notification).

Формат сообщения BGP. Каждое сообщение BGP состоит из заголовка и последующих специфических полей:

MARKER	
MARKER	
MARKER	
MARKER	
LENGTH	TYPE

В поле LENGTH помещается размер сообщения (вместе с заголовком), выраженный в байтах. В поле TYPE помещается код сообщения в соответствии со следующей таблицей:

TYPE	Сообщение
1	OPEN
2	UPDATE
3	NOTIFICATION
4	KEEPALIVE

В поле маркера может быть помещена информация, которая необходима для выполнения операции аутентификации абонента. Если установление подлинности абонента не требуется, маркер формируется значениями – все «1».

OPEN – первое сообщение, которое должно быть передано маршрутизатором BGP после установления соединения TCP.

UPDATE используется для представления маршрута соседнему маршрутизатору BGP. Это сообщение одновременно может быть использовано для уничтожения маршрутов, которые перестали существовать.

Когда в транспортной среде находится в одно и то же время слишком много пакетов, ее производительность начинает падать. Перегрузка может возникнуть в силу нескольких причин. Например, если сразу несколько потоков, поступающих по нескольким входным линиям, устремятся на одну и ту же выходную линию. Если буфер маршрутизатора переполнится, то пакеты начнут теряться. Перегрузки могут случаться и из-за недостаточной скорости процессора. Если процессор будет не в состоянии справиться своевременно с рутинными задачами (размещения пакета в буфере, корректировка таблиц и т.п.), то даже при наличии линий с достаточной пропускной способностью очередь будет расти. Аналогичная картина может случиться при быстром процессоре, но медленном канале и наоборот. Таким образом, источник проблемы – несбалансированность производительности компонентов системы. Перегрузка – это глобальная проблема в сети.

Управление перегрузками – это такая организация потоков в транспортной среде, при которой потоки соответствуют пропускной способности подсети и не превышают ее.

Основные принципы управления перегрузками.

В терминологии теории управления все методы управления перегрузками в сетях можно разбить на две большие группы: с открытым контуром управления и закрытым контуром управления. Методы с открытым контуром предполагают, что все продумано и предусмотрено заранее в конструкции системы, и если нагрузка находится в заданных пределах, то перегрузки не происходит. Если же нагрузка начинает превышать определенные пределы, то заранее известно, когда и где начнется сброс пакетов, в каких точках сети начнется перепланировка ресурсов, и т.п. Главное, что все эти меры будут приниматься вне зависимости от текущего состояния сети.

Решения, основанные на закрытом контуре, используют обратную связь. Эти решения включают три этапа:

- Наблюдение за системой для определения, где и когда началась перегрузка.
- Передача данных туда, где будут предприняты надлежащие меры.
- Перестройка функционирования системы для устранения проблемы.

При наблюдении за системой используются разные метрики для определения перегрузки.

Основными среди них являются:

- Процент пакетов, сброшенных из-за нехватки памяти в буферах.
- Средняя длина очередей в системе.
- Число пакетов, для которых наступил time_out и для которых были сделаны повторные передачи.
- Средняя задержка пакета при доставке и среднее отклонение задержки при доставке пакета.

Следующий шаг при использовании обратной связи – передать информацию о перегрузке туда, где что-то может быть сделано, чтобы исправить положение. Например, маршрутизатор, обнаруживший перегрузку, может направить сообщение о перегрузке всем источникам сообщений. Ясно, что это увеличит нагрузку в сети, причем именно в тот момент, когда это менее всего желательно. Однако есть и другие возможности. Например, в каждом пакете зарезервировать специальный бит перегрузки, и если какой-то маршрутизатор обнаружил перегрузку, то он устанавливает этот бит, тем самым сообщая другим о ней (вспомним структуру кадра во Frame

Relay). Другое решение напоминает прием, используемый некоторыми радиостанциями: направлять несколько автомашин по дорогам, чтобы обнаруживать пробки, а затем сообщать о них по радиоканалам, предупреждая другие машины, призывая их пользоваться объездными путями. По аналогии с этим решением в сети рассылаются специальные пробные пакеты, которые проверяют нагрузку, и если где-то обнаружена перегрузка, то о ней сообщается всем и происходит перенаправление пакетов так, чтобы обогнуть перегруженные участки.

Методы, предотвращающие перегрузки.

Рассмотрение методов, предотвращающих перегрузки, начнем с методов для систем с открытым контуром. Эти методы ориентированы на минимизацию перегрузок при первых признаках их проявления, а не на борьбу с перегрузками, когда они уже случились. Основные факторы, влияющие на перегрузки на канальном, сетевом и транспортном уровнях, перечислены в таблице:

Уровень	Факторы
Транспортный	Повторная передача Порядок передачи бит Уведомления Управление потоком Значение timeout
Сетевой	Виртуальные каналы vs. дейтаграммы внутри подсети Очередность пакетов и сервисы Сброс пакета Алгоритм маршрутизации Управление временем жизни пакетов
Канальный	Повторная передача Порядок передачи бит Уведомления Управление потоком

Методы:

- 1) Схема управления потоком (небольшое окно) сдерживает нарастание трафика и предотвращает появление перегрузок.
- 2) Методы управления очередями, организация очередей: одна общая на входе или одна общая на выходе; по одной на каждую входную линию или на каждую выходную; по одной очереди на каждую входную и выходную – все это влияет на появление перегрузок.
- 3) Выбор метода сброса пакетов также влияет на перегрузки. Правильная маршрутизация, равномерно использующая каналы в транспортной среде, позволяет избежать перегрузки.
- 4) Методы, регулирующие время жизни пакета в сети, также влияют на образование перегрузок.

Additive-increase/multiplicative-decrease (AIMD) алгоритм является алгоритмом управления перегрузками с обратной связью. AIMD сочетает линейный рост окна с экспоненциальным сокращением, когда происходит сброс пакета.

Пусть W – размер окна, тогда:

- Если пакет получен, то: $W = W + \frac{1}{W}$
- Если пакет сброшен, то: $W = \frac{W}{2}$

AIMD в случае одного потока:

- 1) Окно увеличивают, сокращают в соответствии с AIMD, можно определить как много байт канал еще может вместить.
- 2) Пилообразное поведение графика размера окна от времени – это нормальная форма динамики.
- 3) Скорость отправки постоянная: $R = \frac{W}{RTT}$. (RTT – Round-trip time), если есть достаточно буферного пространства ($R * RTT > W$).

AIMD в случае нескольких потоков:

- 1) Окно увеличивают, сокращают в соответствии с AIMD.
- 2) В «узком месте» будут скапливаться пакеты разных потоков.
- 3) Скорость отправки меняется в зависимости от размера окна.
Доля теряемых пакетов:

$$p = \frac{1}{A}, \text{ где } A = \frac{3}{8} W_{max}^2, R = \frac{A}{\frac{W_{max}}{2} RTT}.$$

$$R = \sqrt{\frac{3}{2}} \frac{1}{RTT \sqrt{p}}$$

- 4) AIMD очень чувствителен к частоте потери пакетов.
- 5) AIMD ущемляет потоки с большим RTT.

20. Качество программного обеспечения и методы его контроля. Тестирование и другие методы верификации.

Качество программного обеспечения определяется в стандарте ISO 9126 как вся совокупность его характеристик, относящихся к возможности удовлетворять высказанные или подразумеваемые потребности всех заинтересованных лиц.

Различаются понятия внутреннего качества, связанного с характеристиками ПО самого по себе, без учета его поведения; внешнего качества характеризующего ПО с точки зрения его поведения; и качества ПО при использовании в различных контекстах— того качества, которое ощущается пользователями при конкретных сценариях работы ПО.

6 основных характеристик качества ПО:

1) Функциональность

Способность ПО в определенных условиях решать задачи, нужные пользователям. Определяет, что именно делает ПО, какие задачи оно решает.

а) Функциональная пригодность

Способность решать нужный набор задач.

б) Точность (accuracy).

Способность выдавать нужные результаты.

в) Способность к взаимодействию (interoperability).

Способность взаимодействовать с нужным набором других систем.

г) Соответствие стандартам и правилам (compliance).

Соответствие ПО имеющимся промышленным стандартам, нормативным и законодательным актам, другим регулирующим нормам.

д) Защищенность (security).

Способность предотвращать неавторизированный, т.е. без указания лица, пытающегося его осуществить, и не разрешенный доступ к данным и программам.

2) Надежность

Способность ПО поддерживать определенную работоспособность в заданных условиях.

а) Зрелость, завершенность

Величина, обратная частоте отказов ПО. Обычно измеряется средним временем работы без сбоев и величиной, обратной вероятности возникновения отказа за данный период времени.

б) Устойчивость к отказам (fault tolerance)

Способность поддерживать заданный уровень работоспособности при отказах и нарушениях правил взаимодействия с окружением.

в) Способность к восстановлению (recoverability).

Способность восстанавливать определенный уровень работоспособности и целостность данных после отказа, необходимые для этого время и ресурсы.

г) Соответствие стандартам надежности (reliability compliance).

Этот атрибут добавлен в 2001 году.

3) Удобство использования (usability) или практичность.

Способность ПО быть удобным в обучении и использовании, а также привлекательным для пользователей.

а) Понятность (understandability).

Показатель, обратный к усилиям, которые затрачиваются пользователями на восприятие основных понятий ПО и осознание их

б) Удобство обучения (learnability).

Показатель, обратный к усилиям, затрачиваемым пользователями на обучение работе с ПО.

в) Удобство работ

Показатель, обратный к усилиям, предпринимаемым пользователями для решения своих задач с помощью ПО.

г) Привлекательность (attractiveness).

Способность ПО быть привлекательным для пользователей. Этот атрибут добавлен в 2001.

д) Соответствие стандартам удобства использования (usability compliance).

Этот атрибут добавлен в 2001.

4) Производительность (efficiency) или эффективность

Способность ПО при заданных условиях обеспечивать необходимую работоспособность по отношению к выделяемым для этого ресурсам. Можно определить ее и как отношение получаемых с помощью ПО результатов к затрачиваемым на это ресурсам.

а) Временная эффективность

Способность ПО выдавать ожидаемые результаты, а также обеспечивать передачу необходимого объема данных за отведенное время.

б) Эффективность использования ресурсов

Способность решать нужные задачи с использованием определенных объемов ресурсов определенных видов. Имеются в виду такие ресурсы, как оперативная и долговременная память, сетевые соединения, устройства ввода и вывода, и пр.

в) Соответствие стандартам производительности (efficiency compliance).

Этот атрибут добавлен в 2001.

5) Удобство сопровождения (maintainability).

Удобство проведения всех видов деятельности, связанных с сопровождением программ

а) Анализируемость (analyzability) или удобство проведения анализа

Удобство проведения анализа ошибок, дефектов и недостатков, а также удобство анализа необходимости изменений и их возможных последствий.

б) Удобство внесения изменений (changeability).

Показатель, обратный трудозатратам на выполнение необходимых изменений.

в) Стабильность (stability).

Показатель, обратный риску возникновения неожиданных эффектов при внесении необходимых изменений.

г) Удобство проверки (testability).

Показатель, обратный трудозатратам на проведение тестирования и других видов проверки того, что внесенные изменения привели к нужным результатам.

д) Соответствие стандартам удобства сопровождения

Этот атрибут добавлен в 2001.

6) Переносимость (portability).

Способность ПО сохранять работоспособность при переносе из одного окружения в другое, включая организационные, аппаратные и программные аспекты окружения. Иногда эта характеристика называется в русскоязычной литературе мобильностью.

а) Адаптируемость (adaptability).

Способность ПО приспосабливаться к различным окружениям без проведения для этого действий, помимо заранее предусмотренных

б) Удобство установки (installability).

Способность ПО быть установленным или развернутым в определенном окружении

в) Способность к сосуществованию (coexistence).

Способность ПО сосуществовать с другими программами в общем окружении, деля с ними ресурсы.

г) Удобство замены (replaceability) другого ПО данным.

Возможность применения данного ПО вместо других программных систем для решения тех же задач в определенном окружении.

д) Соответствие стандартам переносимости (portability compliance).

Этот атрибут добавлен в 2001.

Методы контроля качества

Верификация обозначает проверку того, что ПО разработано в соответствии со всеми требованиями к нему, или что результаты очередного этапа разработки соответствуют ограничениям, сформулированным на предшествующих этапах.

Валидация — это проверка того, что сам продукт правилен, т.е. подтверждение того, что он действительно удовлетворяет потребностям и ожиданиям пользователей, заказчиков и других заинтересованных сторон.

Тестирование — это проверка соответствия ПО требованиям, осуществляемая с помощью наблюдения за его работой в специальных, искусственно построенных ситуациях. Такого рода ситуации называют тестовыми или просто тестами.

валидация подтверждает, что «вы создали правильный продукт», а **верификация** подтверждает, что «вы создали продукт таким, каким и намеревались его сделать».

Тестирование — конечная процедура. Набор ситуаций, в которых будет проверяться тестируемое ПО, всегда конечен. Это означает, что при тестировании всегда проверяется очень небольшая доля всех возможных ситуаций. Тем не менее, тестирование может использоваться для достаточно уверенного вынесения оценок о качестве ПО. Для этого необходимо иметь *критерии полноты тестирования*, описывающие важность различных ситуаций для оценки качества, а также эквивалентности и зависимости между ними.

Часто критерий полноты тестирования задается при помощи определения эквивалентности ситуаций, дающей конечный набор классов ситуаций. В таком случае считают, что все равно какую из ситуаций класса использовать в качестве теста. Такой критерий называют *критерием тестового покрытия*, а процент классов

эквивалентности ситуаций, случившихся во время тестирования, — *достигнутым тестовым покрытием*.

Таким образом, основные задачи тестирования: построить такой набор ситуаций, который достаточно репрезентативен и позволял бы завершить тестирование с достаточной степенью уверенности в правильности ПО вообще, и убедиться, что в конкретной ситуации ПО работает правильно, в соответствии с требованиями.

Кроме того, особо выделяют нагрузочное или стрессовое тестирование, проверяющее работоспособность ПО и показатели его производительности в условиях повышенных нагрузок.

На основе исходных данных, используемых для построения тестов, тестирование делят на следующие виды:

- 1) Тестирование методом черного ящика. (Тесты строятся на основе требований и ограничений, прописанных в нормативных актах)
- 2) Тестирование методом белого ящика. (Тесты создаются на основе знаний о структуре самой системы и о том, как она работает)
- 3) Тестирование, нацеленное на определенные ошибки. (Тесты строятся так, чтобы гарантированно выявить определенные ошибки)

Еще одна классификация видов тестирования основана на том уровне, на который оно нацелено.

- 1) Модульное тестирование (Тестирование корректности отдельного модуля вне зависимости от его окружения. При этом проверяется, что если модуль получает на вход данные, удовлетворяющие определенным критериям корректности, то и результаты его корректны.)
- 2) Интеграционное тестирование. (Предназначено для проверки правильности взаимодействия модулей некоторого набора друг с другом. При этом проверяется, что в ходе совместной работы модули взаимодействуют, не нарушая ограничений на это взаимодействие.)
- 3) Системное тестирование. (Предназначено для проверки правильности работы системы в целом, ее способности правильно решать поставленные пользователем задачи в разных ситуациях.)

Проверка на моделях

Проверка свойств на моделях - проверка соответствия ПО требованиям при помощи формализации проверяемых свойств, построения формальных моделей проверяемого ПО (чаще всего в виде автоматов различных видов) и автоматической проверки выполнения этих свойств на построенных моделях. (Например, отсутствие взаимных блокировок) Свойства безопасности (safety properties) утверждают, что нечто нежелательное никогда не случится в ходе работы ПО. Свойства живучести, наоборот, утверждают, что нечто желательное при любом развитии событий произойдет в ходе его работы. (Например, гарантированная доставка сообщения)

В классическом подходе к проверке на моделях проверяемые свойства формализуются в виде так называемых временных логик. Операторы «всегда в будущем» и «когда-то в будущем».

Программа, как правило, моделируется с помощью конечного автомата. Проверка, выполняемая автоматически, состоит в том, что для всех достижимых при работе системы

состояний этого автомата проверяется нужное свойство. Если оно выполнено – все хорошо, если нет – ошибка.

Основная проблема – большое количество состояний в моделях, хорошо отражающих реальные системы.

21. Виды параллельной обработки данных, их особенности. Компьютеры с общей и распределенной памятью. Вычислительные кластеры: узлы, коммуникационная сеть, способы построения. Производительность вычислительных систем, методы оценки и измерения.

Параллельная обработка данных, воплощая идею одновременного выполнения нескольких действий, имеет две разновидности: конвейерность и собственно параллельность.

Параллельная обработка. Если некое устройство выполняет одну операцию за единицу времени, то тысячу операций оно выполнит за тысячу единиц. Если предположить, что есть пять таких же независимых устройств, способных работать одновременно, то ту же тысячу операций система из пяти устройств может выполнить уже не за тысячу, а за двести единиц времени. Аналогично система из N устройств ту же работу выполнит за $1000/N$ единиц времени. **УВЕЛИЧЕНИЕ КОЛИЧЕСТВА НЕЗАВИСИМО РАБОТАЮЩИХ УСТРОЙСТВ.**

Конвейерная обработка. Что необходимо для сложения двух вещественных чисел, представленных в форме с плавающей запятой? Целое множество мелких операций таких, как сравнение порядков, выравнивание порядков, сложение мантисс, нормализация и т.п.

Процессоры первых компьютеров выполняли все эти "микрооперации" для каждой пары аргументов последовательно одна за одной до тех пор, пока не доходили до окончательного результата, и лишь после этого переходили к обработке следующей пары слагаемых. **УСЛОЖНИТЬ САМО УСТРОЙСТВО, ЧТОБЫ НА РАЗНЫХ ЭТАПАХ МОГЛИ НАХОДИТЬСЯ РАЗНЫЕ ДАННЫЕ.**

Идея конвейерной обработки заключается в выделении отдельных этапов выполнения общей операции, причем каждый этап, выполнив свою работу, передавал бы результат следующему, одновременно принимая новую порцию входных данных. Получаем очевидный выигрыш в скорости обработки за счет совмещения прежде разнесенных во времени операций. Предположим, что в операции можно выделить пять микроопераций, каждая из которых выполняется за одну единицу времени. Если есть одно неделимое последовательное устройство, то 100 пар аргументов оно обработает за 500 единиц. Если каждую микрооперацию выделить в отдельный этап (или иначе говорят - ступень) конвейерного устройства, то на пятой единице времени на разной стадии обработки такого устройства будут находиться первые пять пар аргументов, а весь набор из ста пар будет обработан за $5+99=104$ единицы времени - ускорение по сравнению с последовательным устройством почти в пять раз (по числу ступеней конвейера). **ТЕ СУЩЕСТВУЕТ НЕКОТОРАЯ ЗАДЕРЖКА ДЛЯ ТОГО ЧТОБЫ ЗАПОЛНИТЬ ВСЕ ЭТАПЫ КОНВЕЕРА, НО КОГДА ОНА ЗАПОЛНЕНА ДАЛЬШЕ ПРОИСХОДИТ УСКОРЕНИЕ ОБРАБОТКИ.**

1. Векторно-конвейерные компьютеры. Конвейерные функциональные устройства и набор векторных команд - это две особенности таких машин. В отличие от традиционного подхода, векторные команды оперируют целыми массивами независимых данных, что позволяет эффективно загружать доступные конвейеры, т.е. команда вида $A=B+C$ может означать сложение двух массивов, а не двух чисел. Характерным представителем данного направления является семейство векторно-конвейерных компьютеров CRAY куда входят,

например, CRAY EL, CRAY J90, CRAY T90 (в марте 2000 года американская компания TERA перекупила подразделение CRAY у компании SiliconGraphics, Inc.).

2. Массивно-параллельные компьютеры с распределенной памятью. Идея построения компьютеров этого класса тривиальна: возьмем серийные микропроцессоры, снабдим каждый своей локальной памятью, соединим посредством некоторой коммуникационной среды - вот и все. Достоинств у такой архитектуры масса: если нужна высокая производительность, то можно добавить еще процессоров, если ограничены финансы или заранее известна требуемая вычислительная мощность, то легко подобрать оптимальную конфигурацию и т.п.

Однако есть и решающий "минус", сводящий многие "плюсы" на нет. Дело в том, что межпроцессорное взаимодействие в компьютерах этого класса идет намного медленнее, чем происходит локальная обработка данных самими процессорами. Именно поэтому написать эффективную программу для таких компьютеров очень сложно, а для некоторых алгоритмов иногда просто невозможно. К данному классу можно отнести компьютеры IntelParagon, IBM SP1, Parsytec, в какой-то степени IBM SP2 и CRAY T3D/T3E, хотя в этих компьютерах влияние указанного минуса значительно ослаблено. К этому же классу можно отнести и сети компьютеров, которые все чаще рассматривают как дешевую альтернативу крайне дорогим суперкомпьютерам.

3. Параллельные компьютеры с общей памятью. Вся оперативная память таких компьютеров разделяется несколькими одинаковыми процессорами. Это снимает проблемы предыдущего класса, но добавляет новые - число процессоров, имеющих доступ к общей памяти, по чисто техническим причинам нельзя сделать большим. В данное направление входят многие современные многопроцессорные SMP-компьютеры или, например, отдельные узлы компьютеров HP Exemplar и SunStarFire.

4. Последнее направление, строго говоря, не является самостоятельным, а скорее представляет собой комбинации предыдущих трех. Из нескольких процессоров (традиционных или векторно-конвейерных) и общей для них памяти сформируем вычислительный узел. Если полученной вычислительной мощности не достаточно, то объединим несколько узлов высокоскоростными каналами. Подобную архитектуру называют кластерной, и по такому принципу построены CRAY SV1, HP Exemplar, SunStarFire, NEC SX-5, последние модели IBM SP2 и другие. Именно это направление является в настоящее время наиболее перспективным для конструирования компьютеров с рекордными показателями производительности.

Различных вариантов построения кластеров очень много. Одно из существенных различий лежит в используемой сетевой технологии, выбор которой определяется, прежде всего, классом решаемых задач.

Какими же числовыми характеристиками выражается производительность коммуникационных сетей в кластерных системах? Необходимых пользователю характеристик две: латентность и пропускная способность сети. Латентность — это время начальной задержки при посылке сообщений. Пропускная способность сети определяется скоростью передачи информации по каналам связи. Если в программе много маленьких сообщений, то сильно скажется латентность. Если сообщения передаются большими порциями, то важна высокая пропускная способность

каналов связи. Из-за латентности максимальная скорость передачи по сети не может быть достигнута на сообщениях с небольшой длиной. На практике пользователям не столько важны заявляемые производителем пиковые характеристики, сколько реальные показатели, достигаемые на уровне приложений. После вызова пользователем функции отправки сообщения `Send()` сообщение последовательно проходит через целый набор слоев, определяемых особенностями организации программного обеспечения и аппаратуры. Этим, в частности, определяются и множество вариаций на тему латентности реальных систем. Установили MPI на компьютере плохо, латентность будет большая, купили дешевую сетевую карту от неизвестного производителя, ждите дальнейших сюрпризов.

В заключение параграфа давайте попробуем и для данного класса компьютеров выделить факторы, снижающие производительность вычислительных систем с распределенной памятью на реальных программах. Начнем с уже упоминавшегося ранее закона Амдала. Для компьютеров данного класса он играет очень большую роль. В самом деле, если предположить, что в программе есть лишь 2% последовательных операций, то рассчитывать на более чем 50-кратное ускорение работы программы не приходится. Теперь попробуйте критически взглянуть на свою программу. Скорее всего, в ней есть инициализация, операции ввода/вывода, какие-то сугубо последовательные участки. Оцените их долю на фоне всей программы и на мгновение предположите, что вы получили доступ к вычислительной системе из 1000 процессоров. После вычисления верхней границы для ускорения программы на такой системе, думаем, станет ясно, что недооценивать влияние закона Амдала никак нельзя. Поскольку компьютеры данного класса имеют распределенную память, то взаимодействие процессоров между собой осуществляется с помощью передачи сообщений. Отсюда два других замедляющих фактора — латентность и скорость передачи данных по каналам коммуникационной среды. В зависимости от коммуникационной структуры программы степень влияния этих факторов может сильно меняться.

Если аппаратура или программное обеспечение не поддерживают возможности асинхронной отправки сообщений на фоне вычислений, то возникнут неизбежные накладные расходы, связанные с ожиданием полного завершения взаимодействия параллельных процессов.

Для достижения эффективной параллельной обработки необходимо добиться максимально равномерной загрузки всех процессоров. Если равномерности нет, то часть процессоров неизбежно будет простаивать, ожидая остальных, хотя в это время они вполне могли бы выполнять полезную работу. Данная проблема решается проще, если вычислительная система однородна. Очень большие трудности возникают при переходе на неоднородные системы, в которых есть значительное различие либо между вычислительными узлами, либо между каналами связи. Существенный фактор — это реальная производительность одного процессора вычислительной системы. Разные модели микропроцессоров могут поддерживать несколько уровней кэш-памяти, иметь специализированные функциональные устройства и т. п. Возьмем хотя бы иерархию памяти компьютера Cray T3E: регистры процессора, кэш-память 1-го уровня, кэш-память 2-го уровня, локальная память процессора, удаленная память другого процессора. Эффективное использование такой структуры требует особого внимания при выборе подхода к решению задачи. Дополнительно каждый микропроцессор может иметь элементы векторно-конвейерной архитектуры. К сожалению, как и прежде, на работе каждой конкретной программы в той или иной мере сказываются все эти факторы. Однако в отличие от компьютеров других классов, суммарное воздействие изложенных

здесь факторов может снизить реальную производительность не в десятки, а в сотни и даже тысячи раз по сравнению с пиковой.

Потенциал компьютеров этого класса огромен, добиться на них можно очень многого. Крайняя точка — Интернет. Его тоже можно рассматривать как компьютер с распределенной памятью. Причем, как самый мощный в мире компьютер.

Пиковая производительность.

Главной отличительной особенностью многопроцессорной вычислительной системы является ее производительность, т.е. количество операций, производимых системой за единицу времени. Различают пиковую и реальную производительность.

Подпиковой понимают величину, равную произведению пиковой производительности одного процессора на число таких процессоров в данной машине. При этом предполагается, что все устройства компьютера работают в максимально производительном режиме. Пиковая производительность компьютера вычисляется однозначно, и эта характеристика является базовой, по которой производят сравнение высокопроизводительных вычислительных систем. Чем больше пиковая производительность, тем (теоретически) быстрее пользователь сможет решить свою задачу. Пиковая производительность есть величина теоретическая и, вообще говоря, недостижимая при запуске конкретного приложения. Реальная же производительность, достигаемая на данном приложении, зависит от взаимодействия программной модели, в которой реализовано приложение, с архитектурными особенностями машины, на которой приложение запускается.

Существует два способа оценки пиковой производительности компьютера. Один из них опирается на число команд, выполняемых компьютером за единицу времени. Единицей измерения, как правило, является MIPS (MillionInstructionsPerSecond).

Производительность, выраженная в MIPS, говорит о скорости выполнения компьютером своих же инструкций. Но, во-первых, заранее не ясно, в какое количество инструкций отобразится конкретная программа, а во-вторых, каждая программа обладает своей спецификой, и число команд от программы к программе может меняться очень сильно. В связи с этим данная характеристика дает лишь самое общее представление о производительности компьютера.

Другой способ измерения производительности заключается в определении числа вещественных операций, выполняемых компьютером за единицу времени. Единицей измерения является Flops (Floatingpointoperationspersecond) — число операций с плавающей точкой, производимых компьютером за одну секунду. Такой способ является более приемлемым для пользователя, поскольку ему известна вычислительная сложность программы, и, пользуясь этой характеристикой, пользователь может получить нижнюю оценку времени ее выполнения.

Однако пиковая производительность получается только в идеальных условиях, т.е. при отсутствии конфликтов при обращении к памяти при равномерной загрузке всех устройств.

В реальных условиях на выполнение конкретной программы влияют такие аппаратно-программные особенности данного компьютера как: особенности структуры процессора, системы команд, состав функциональных устройств, реализация ввода/вывода, эффективность работы компиляторов.

Методы оценки производительности.

Основу для сравнения различных типов компьютеров между собой дают стандартные методики измерения производительности. В процессе развития вычислительной техники появилось несколько таких стандартных методик. Они позволяют разработчикам и

пользователям осуществлять выбор между альтернативами на основе количественных показателей, что дает возможность постоянного прогресса в данной области. Единицей измерения производительности компьютера является время: компьютер, выполняющий тот же объем работы за меньшее время является более быстрым. Время выполнения любой программы измеряется в секундах. Часто производительность измеряется как скорость появления некоторого числа событий в секунду, так что меньшее время подразумевает большую производительность.

Однако в зависимости от того, что мы считаем, время может быть определено различными способами. Наиболее простой способ определения времени называется астрономическим временем, временем ответа (responsetime), временем выполнения(executiontime) или прошедшим временем (elapsedtime). Это задержка выполнения задания, включающая буквально все: работу процессора, обращения к диску, обращения к памяти, ввод/вывод и накладные расходы операционной системы. Однако при работе в мультипрограммном режиме во время ожидания ввода/вывода для одной программы, процессор может выполнять другую программу, и система не обязательно будет минимизировать время выполнения данной конкретной программы.

Для измерения времени работы процессора на данной программе используется специальный параметр - время ЦП (CPU time), которое не включает время ожидания ввода/вывода или время выполнения другой программы. Очевидно, что время ответа, видимое пользователем, является полным временем выполнения программы, а не временем ЦП. Время ЦП может далее делиться на время, потраченное ЦП непосредственно на выполнение программы пользователя и называемое пользовательским временем ЦП, и время ЦП, затраченное операционной системой на выполнение заданий, затребованных программой, и называемое системным временем ЦП.

В ряде случаев системное время ЦП игнорируется из-за возможной неточности измерений, выполняемых самой операционной системой, а также из-за проблем, связанных со сравнением производительности машин с разными операционными системами. С другой стороны, системный код на некоторых машинах является пользовательским кодом на других и, кроме того, практически никакая программа не может работать без некоторой операционной системы. Поэтому при измерениях производительности процессора часто используется сумма пользовательского и системного времени ЦП.

В большинстве современных процессоров скорость протекания процессов взаимодействия внутренних функциональных устройств определяется не естественными задержками в этих устройствах, а задается единой системой синхросигналов, вырабатываемых некоторым генератором тактовых импульсов, как правило, работающим с постоянной скоростью.

Дискретные временные события называются тактами синхронизации (clockticks), просто тактами (ticks), периодами синхронизации (clockperiods), циклами (cycles) или циклами синхронизации (clockcycles). Разработчики компьютеров обычно говорят о периоде синхронизации, который определяется либо своей длительностью (например, 10 наносекунд), либо частотой (например, 100 МГц). Длительность периода синхронизации есть величина, обратная к частоте синхронизации.

Таким образом, время ЦП для некоторой программы может быть выражено двумя способами: количеством тактов синхронизации для данной программы, умноженным на длительность такта синхронизации, либо количеством тактов синхронизации для данной программы, деленным на частоту синхронизации.

Важной характеристикой, часто публикуемой в отчетах по процессорам, является среднее количество тактов синхронизации на одну команду - CPI (clockcyclesperinstruction). При известном количестве выполняемых команд в программе этот параметр позволяет быстро оценить время ЦП для данной программы. Таким образом, производительность ЦП зависит от трех параметров: такта (или частоты) синхронизации, среднего количества тактов на команду и количества выполняемых команд. Невозможно изменить ни один из указанных параметров изолированно от другого, поскольку базовые технологии, используемые для изменения каждого из этих параметров, взаимосвязаны: частота синхронизации определяется технологией аппаратных средств и функциональной организацией процессора; среднее количество тактов на команду зависит от функциональной организации и архитектуры системы команд; а количество выполняемых в программе команд определяется архитектурой системы команд и технологией компиляторов. Когда сравниваются две машины, необходимо рассматривать все три компонента, чтобы понять относительную производительность. В процессе поиска стандартной единицы измерения производительности компьютеров было принято несколько популярных единиц измерения, вследствие чего несколько безвредных терминов были искусственно вырваны из их хорошо определенного контекста и использованы там, для чего они никогда не предназначались. В действительности единственной подходящей и надежной единицей измерения производительности является время выполнения реальных программ, и все предлагаемые замены этого времени в качестве единицы измерения или замены реальных программ в качестве объектов измерения на синтетические программы только вводят в заблуждение.

Тесты для оценки производительности систем:

Ливерморский набор циклов : 24 цикла разной степени векторизуемости

В Ливерморских циклах встречаются последовательные, сеточные, конвейерные, волновые вычислительные алгоритмы, что подтверждает их пригодность и для параллельных машин.

Набор тестов LINPACK.

В основе алгоритмов действующего варианта LINPACK лежит метод декомпозиции.

Исходная матрица размером 100x100 элементов (в последнем варианте размером 1000x1000) сначала представляется в виде произведения двух матриц стандартной структуры, над которыми затем выполняется собственно алгоритм нахождения решения.

Тесты SPEC

Набор тестов CINT92, измеряющий производительность процессора при обработке целых чисел, состоит из шести программ, написанных на языке Си и выбранных из различных прикладных областей: теория цепей, интерпретатор языка Лисп, разработка логических схем, упаковка текстовых файлов, электронные таблицы и компиляция программ.

Набор тестов CFP92, измеряющий производительность процессора при обработке чисел с плавающей точкой, состоит из 14 программ, также выбранных из различных прикладных областей: разработка аналоговых схем, моделирование методом Монте-Карло, квантовая химия, оптика, робототехника, квантовая физика, астрофизика, прогноз погоды и другие научные и инженерные задачи. Две программы из этого набора написаны на языке Си, а остальные 12 - на Фортране. В пяти программах используется одинарная, а в остальных - двойная точность.

22. Закон Амдала, его следствия. Граф алгоритма. Критический путь графа алгоритм, ярусно-параллельная форма графа алгоритма. Этапы решения задач на параллельных вычислительных системах.

Предположим, что в вашей программе доля операций, которые нужно выполнять последовательно, равна f , где $0 \leq f \leq 1$ (при этом доля понимается не по статическому числу строк кода, а по числу операций в процессе выполнения). Крайние случаи в значениях f соответствуют полностью параллельным ($f=0$) и полностью последовательным ($f=1$) программам. Так вот, для того, чтобы оценить, какое ускорение S может быть получено на компьютере из p процессоров при данном значении f , можно воспользоваться законом Амдала:

$$S \leq \frac{1}{f + \frac{1-f}{p}}$$

Если 9/10 программы исполняется параллельно, а 1/10 по-прежнему последовательно, то ускорения более, чем в 10 раз получить в принципе невозможно вне зависимости от качества реализации параллельной части кода и числа используемых процессоров (ясно, что 10 получается только в том случае, когда время исполнения параллельной части равно 0).

Посмотрим на проблему с другой стороны: а какую же часть кода надо ускорить (а значит и предварительно исследовать), чтобы получить заданное ускорение? Ответ можно найти в следствии из закона Амдала: для того чтобы ускорить выполнение программы в q раз необходимо ускорить не менее, чем в q раз не менее, чем $(1-1/q)$ -ю часть программы.

Пример 4. 4 раз необходимо
Следовательно, если есть

желание ускорить программу в 100 раз по сравнению с ее последовательным вариантом, то необходимо получить не меньшее ускорение не менее, чем на 99.99% кода, что почти всегда составляет значительную часть программы! Отсюда первый вывод – прежде, чем переделывать код для перехода на параллельный надо подумать. Если оценив заложенный в программе алгоритм вы поняли, что доля последовательных операций велика, то на значительное ускорение рассчитывать не приходится и нужно думать о замене отдельных компонент алгоритма. В ряде случаев последовательный характер алгоритма изменить не так сложно. Допустим, что в программе есть следующий фрагмент для вычисления суммы n чисел:

```
s=0
Do i = 1, n
    s = s + a(i)
EndDo
```

(можно то же самое на любом другом языке) По своей природе он строго последователен, так как на i -й итерации цикла требуется результат с $(i-1)$ -й и все итерации выполняются одна за другой. Имеем 100% последовательных операций, а значит и никакого эффекта от использования параллельных компьютеров. Вместе с тем, выход очевиден. Поскольку в большинстве реальных программ (вопрос: а почему в большинстве, а не во всех?) нет существенной разницы, в каком порядке складывать числа, выберем иную схему сложения. Сначала найдем сумму пар соседних элементов: $a(1)+a(2)$, $a(3)+a(4)$, $a(5)+a(6)$ и т.д. Заметим, что при такой схеме все пары можно складывать одновременно! На следующих шагах будем действовать абсолютно аналогично, получив вариант параллельного алгоритма. Казалось бы в данном случае все проблемы удалось разрешить. Но представьте, что доступные вам процессоры разнородны по своей

производительности. Значит будет такой момент, когда кто-то из них еще трудится, а кто-то уже все сделал и бесполезно простаивает в ожидании. Если разброс в производительности компьютеров большой, то и эффективность всей системы при равномерной загрузке процессоров будет крайне низкой. Но пойдем дальше и предположим, что все процессоры одинаковы. Проблемы кончились? Опять нет! Процессоры выполнили свою работу, но результат-то надо передать другому для продолжения процесса суммирования... а на передачу уходит время... и в это время процессоры опять простаивают...

Словом, заставить параллельную вычислительную систему или супер-ЭВМ работать с максимальной эффективностью на конкретной программе это, прямо скажем, задача не из простых, поскольку **необходимо тщательное согласование структуры программ и алгоритмов с особенностями архитектуры параллельных вычислительных систем.**

Суперлинейное ускорение. При увеличении числа процессоров, например, в 2 раза задача начинает выполняться больше, чем в 2 раза. Такое возможно, если она начинается помещаться в кэш процессоров, при условии, что раньше не помещалась и приходилось обращаться к медленной памяти.

Дадим сказанному следующую трактовку. Пусть при фиксированных входных данных программа описывает некоторый алгоритм. Построим ориентированный граф. В качестве вершин возьмем любое множество, например, множество точек арифметического пространства, на которое взаимнооднозначно отображается множество всех операций алгоритма. Возьмем любую пару вершин u , v . Допустим, что согласно описанному выше частично-порядку операция, соответствующая вершине u , должна поставлять аргумент операции, соответствующей вершине v . Тогда проведем дугу из вершины u в вершину v . Если соответствующие операции могут выполняться независимо друг от друга, дугу проводить не будем. В случае, когда аргументом операции является начальное данное или результат операции нигде не используется, возможны различные договоренности. Например, можно считать, что соответствующие дуги отсутствуют. Или предполагать, что все входные данные и результаты вводятся и выводятся через специальные устройства ввода/вывода. В этом случае вершины графа, отвечающие таким операциям, и только они не будут иметь соответственно входящие и выходящие дуги. Мы будем поступать в зависимости от обстоятельств. Построенный таким образом граф можно было бы назвать графом информационной зависимости реализации алгоритма при фиксированных входных данных. Однако такое название слишком громоздко. Поэтому будем называть его в дальнейшем просто *графом алгоритма*. Независимо от способа построения ориентированного графа, те его вершины, которые не имеют ни одной входящей или выходящей дуги, будем называть соответственно *входными* или *выходными* вершинами графа.

Это понятие требует некоторого пояснения. Граф алгоритма почти всегда зависит от входных данных. Даже если в программе отсутствуют условные операторы, он будет зависеть от размеров массивов, т. к. они определяют общее число выполняемых операций и, следовательно, общее число вершин графа. Так что в действительности граф алгоритма почти всегда есть *параметризованный* граф. От значений параметров зависит, конечно, не только число

вершин, но и вся совокупность дуг. Если программа не имеет условных операторов, то как ее саму, так и описанный ею алгоритм будем называть *детерминированными*. В противном случае будем называть их *недетерминированными*. Имеется одно принципиальное отличие графа детерминированного алгоритма от графа недетерминированного алгоритма. Для детерминированного алгоритма всегда существует взаимно-однозначное соответствие между всеми операциями описывающей его программы и всеми вершинами графа алгоритма. Для недетерминированного алгоритма взаимно-однозначного соответствия при всех значениях параметров, т. е. входных данных, нет. Можно лишь утверждать, что в этом случае при каждом наборе значений параметров существует взаимно-однозначное соответствие между каким-то подмножеством всех операций программы и вершинами графа. Разным значениям параметров могут соответствовать разные подмножества.

Введенный в рассмотрение граф алгоритма есть ориентированный ациклический мультиграф. Его ацикличность следует из того, что в любых программах реализуются только явные вычисления и никакая величина не может определяться через саму себя. Даже если в программе имеются рекурсивные выражения, то это всего лишь удобная форма описания однотипных вычислений. При каждом обращении к рекурсии на самом деле реализуются разные операции. В общем случае граф алгоритма есть мультиграф, т. е. две вершины могут быть связаны несколькими дугами. Это будет тогда, когда в качестве разных аргументов одной и той же операции используется одна и та же величина. Будем применять для обозначения графа стандартную символику $G = (V, E)$, где V — множество вершин, E — множество дуг графа G .

утверждение 4.1

Пусть задан ориентированный ациклический граф, имеющий n вершин. Существует число $s < n$, для которого все вершины графа можно так пометить одним из индексов $1, 2, \dots, s$, что если дуга из вершины с индексом i идет в вершину с индексом j , то $i < j$.

Выберем в графе любое число вершин, не имеющих предшествующих, и пометим их индексом 1. Удалим из графа помеченные вершины и инцидентные им дуги. Оставшийся граф также является ациклическим. Выберем в нем любое число вершин, не имеющих предшествующих, и пометим их индексом 2. Продолжая этот процесс, в конце концов, исчерпаем весь граф. Так как при каждом шаге помечается не менее одной вершины, то число различных индексов не превышает числа вершин графа.

Отсюда следует, что никакие две вершины с одним и тем же индексом не связаны дугой.

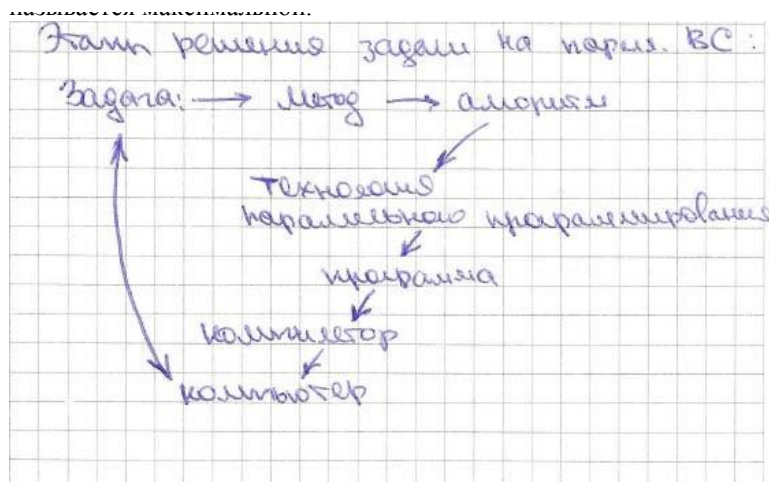
Минимальное число индексов, которым можно пометить все вершины графа, на 1 больше длины его критического пути. И, наконец, для любого целого числа s , не превосходящего общего числа вершин, но большего длины критического пути, существует такая разметка вершин графа, при которой используются все s индексов.

Граф, размеченный в соответствии с утверждением 4.1, называется **строгой параллельной формой графа**. Если в параллельной форме некоторая вершина помечена индексом k , то это означает, что длины всех путей, оканчивающихся в данной вершине, меньше k . Существует строгая параллельная форма, при которой максимальная из длин путей, оканчивающихся в вершине с индексом k , равна $k-1$. Для этой параллельной формы число используемых индексов на 1 больше длины критического пути графа. Среди подобных параллельных форм существует такая, в которой все входные вершины

Критический путь графа — путь максимальной длины в ориентированном ациклическом графе.

Его длина является минимальной из всех возможных высот у ярусно-параллельной формы данного ациклического графа.

находятся в группе с одним индексом, равным 1. Эта строгая параллельная форма называется **канонической**. Для заданного графа его каноническая параллельная форма единственна. Группа вершин, имеющих одинаковые индексы, называется **ярусом** параллельной формы, а число вершин в группе — шириной яруса. Число ярусов в параллельной форме называется **высотой** параллельной формы, а максимальная ширина ярусов — ее **шириной**. Параллельная форма минимальной высоты называется максимальной.



Итак, вы приступаете к созданию параллельной программы. Желание есть, задача ясна, метод выбран, целевой компьютер, скорее всего, тоже определен. Осталось только все мысли выразить в той или иной форме, понятной для этого компьютера. Чем руководствоваться, если собственного опыта пока мало, а априорной информации о доступных технологиях параллельного программирования явно недостаточно? Наводящих соображений может быть много, но в результате вы все равно будете вынуждены пойти на компромисс, делая выбор между временем разработки программы, ее эффективностью и переносимостью, интенсивностью последующего использования программы, необходимостью ее дальнейшего развития. Не вдаваясь в детали выбора, попробуйте для начала оценить, насколько важны для вас следующие три характеристики.

Основное назначение параллельных компьютеров — это быстро решать задачи. Если технология программирования по разным причинам не позволяет в полной мере использовать весь потенциал вычислительной системы, то нужно ли тратить усилия на ее освоение? Не будем сейчас обсуждать причины. Ясно то, что возможность создания эффективных программ является серьезным аргументом в выборе средств программирования.

Технология может давать пользователю полный контроль над использованием ресурсов вычислительной системы и ходом выполнения его программы. Для этого ему предлагается набор из нескольких сотен конструкций и функций, предназначенных "на все случаи жизни". Он может создать действительно эффективную программу, если правильно воспользуется предложенными средствами. Но захочет ли он это делать? Не стоит забывать, что он должен решать свою задачу из своей предметной области, где и своих проблем хватает. Маловероятно, что физик, химик, геолог или эколог с большой радостью захочет осваивать новую специальность, связанную с параллельным программированием. Возможность быстрого создания параллельных программ должна приниматься в расчет наравне с другими факторами.

Вычислительная техника меняется очень быстро. Предположим, что была найдена технология, позволяющая быстро создавать эффективные параллельные программы. Что произойдет через пару лет, когда появится новое поколение компьютеров? Возможных вариантов развития событий два. Первый вариант — разработанные прежде программы были "одноразовыми" и сейчас ими уже никто не интересуется. Бывает и так. Однако, как правило, в параллельные программы вкладывается слишком много средств (времени, усилий, финансовых затрат), чтобы просто так об этом забыть и начать разработку заново. Хочется перенести накопленный багаж на новую компьютерную платформу. Скорее всего, на новом компьютере старая программа рано или поздно работать будет, и даже будет давать правильный результат. Но дает ли выбранная технология гарантии сохранения эффективности параллельной программы при ее переносе с одного компьютера на другой? Скорее всего, нет. Программу для новой платформы нужно оптимизировать заново. А тут еще разработчики новой платформы предлагают вам очередную новую технологию программирования, которая опять позволит создать выдающуюся программу для данного компьютера. Программы переписываются, и так по кругу в течение многих лет.

Выбор технологии параллельного программирования — это и в самом деле вопрос не простой. Если попытаться сделать обзор средств, которые могут помочь в решении задач на параллельном компьютере, то даже поверхностный анализ приведет к списку из более 100 наименований. В некоторых случаях выбор определяется просто. Например, вполне жизненной является ситуация, когда воспользоваться можно только тем, что установлено на доступном вам компьютере. Другой аргумент звучит так: "... все используют MPI, поэтому и я тоже буду...". Если есть возможность и желание сделать осознанный выбор, это обязательно нужно делать. Посоветуйтесь со специалистами. Проблемы в дальнейшем возникнут в любом случае, вопрос только насколько

быстро и в каком объеме. Если выбор будет правильным, проблем будет меньше. Если неправильным, то тоже не отчаивайтесь, будет возможность подумать о выборе еще раз. Сделав выбор несколько раз, вы станете специалистом в данной области, забудете о своих прежних интересах, скажем, о квантовой химии или вычислительной гидродинамике. Не исключено, что в итоге вы сможете предложить свою технологию и найти ответ на центральный вопрос параллельных вычислений: "Как создавать эффективные программы для параллельных компьютеров?"

В данной главе мы рассмотрим различные подходы к программированию параллельных компьютеров. Одни широко используются на практике, другие интересны своей идеей, третьи лаконичны и выразительны. Хотелось показать широкий спектр существующих средств, но и не сводить описание каждой технологии до одного абзаца текста. Противоречивая задача. Однако надеемся, что после изучения каждого раздела вы сможете не только проводить качественное сравнение технологий, но и самостоятельно писать содержательные программы. Знакомясь с различными системами параллельного программирования, обязательно обратите внимание на следующее обстоятельство. Если вы решаете учебные задачи или производственные задачи небольшого размера, вам почти наверняка не придется задумываться об эффективности использования параллельной вычислительной техники. В этом случае выбор системы программирования практически не имеет значения. Используйте то, что вам больше нравится. Но как только вы начнете решать большие задачи и, особенно, предельно большие многовариантные задачи, вопрос эффективности может оказаться ключевым.

Очень скоро станет ясно, что при использовании любой системы параллельного программирования желание повысить производительность вычислительной техники на вашей задаче сопровождается тем, что от вас требуется все больше и больше каких-то новых сведений о структуре задачи, программы или алгоритма. Ни одна система параллельного программирования не гарантирует высокую эффективность вычислительных процессов без предоставления дополнительных сведений.

23. Технологии параллельного программирования MPI и OpenMP.

Одним из наиболее популярных средств программирования компьютеров с общей памятью, базирующихся на традиционных языках программирования и использовании специальных комментариев, в настоящее время является технология OpenMP. За основу берется последовательная программа, а для создания ее параллельной версии пользователю предоставляется набор директив, процедур и переменных окружения. Стандарт OpenMP разработан для языков Фортран, С и С++. Поскольку все основные конструкции для этих языков похожи, то рассказ о данной технологии мы будем вести на примере только одного из них, а именно на примере языка Фортран.



Рис. 1. Процесс исполнения OpenMP-программы

Как, с точки зрения OpenMP, пользователь должен представлять свою параллельную программу? Весь текст программы разбит на последовательные и параллельные области (см. рис.1). В начальный момент времени порождается нить-мастер или "основная" нить, которая начинает выполнение программы со стартовой точки. Здесь следует сразу сказать, почему вместо традиционных для параллельного программирования процессов появился новый термин – нити (threads, легковесные процессы). Технология OpenMP опирается на понятие общей памяти, поэтому она, в значительной степени, ориентирована на SMP-компьютеры. На подобных архитектурах возможна эффективная поддержка нитей, исполняющихся на различных процессорах, что позволяет избежать значительных накладных

расходов на поддержку классических UNIX-процессов.

Основная нить и только она исполняет все последовательные области программы. При входе в параллельную область порождаются дополнительные нити. После порождения каждая нить получает свой уникальный номер, причем нить-мастер всегда имеет номер 0. Все нити исполняют один и тот же код, соответствующий параллельной области. При выходе из параллельной области основная нить дожидается завершения остальных нитей, и дальнейшее выполнение программы продолжает только она.

В параллельной области все переменные программы разделяются на два класса: общие (SHARED) и локальные (PRIVATE). Общая переменная всегда существует лишь в одном экземпляре для всей программы и доступна всем нитям под одним и тем же именем. Объявление же локальной переменной вызывает порождение своего экземпляра данной переменной для каждой нити. Изменение нитью значения своей локальной переменной, естественно, никак не влияет на изменение значения этой же локальной переменной в других нитях.

По сути, только что рассмотренные два понятия: области и классы переменных, и определяют идею написания параллельной программы в рамках OpenMP: некоторые фрагменты текста программы объявляются параллельными областями; именно эти области и только они исполняются набором нитей, которые могут работать как с общими, так и с локальными переменными. Все остальное – это конкретизация деталей и описание особенностей реализации данной идеи на практике.

Рассмотрим базовые положения и основные конструкции OpenMP. Все директивы OpenMP располагаются в комментариях и начинаются с одной из следующих комбинаций: !\$OMP, C\$OMP или *\$OMP (напомним, что строка, начинающаяся с одного из символов '!', 'C' или '*' по правилам языка Фортран считается комментарием). В дальнейшем изложении при описании конкретных директив для сокращения записи мы иногда будем опускать эти префиксы, хотя в реальных программах они, конечно же, всегда должны присутствовать. Все переменные окружения и функции, относящиеся к OpenMP, начинаются с префикса OMP_.

`#pragma omp директива<> опция[ [ [ , ] опция]... ]`

Описание параллельных областей. Для определения параллельных областей программы используется пара директив

Спецификация OpenMP для C/C++, выпущенная на год позже фортранной, содержит в основном аналогичную функциональность.

Необходимо лишь отметить следующие моменты:

- 1) Вместо спецкомментариев используются директивы компилятора "#pragma omp".
- 2) Компилятор с поддержкой OpenMP определяет макрос "_OPENMP", который может использоваться для условной компиляции отдельных блоков, характерных для параллельной версии программы.
- 3) Распараллеливание применяется к for-циклам, для этого используется директива "#pragma omp for". В параллельных циклах запрещается использовать оператор break.
- 4) Статические (static) переменные, определенные в параллельной области программы, являются общими (shared).
- 5) Память, выделенная с помощью malloc(), является общей (однако указатель на нее может быть как общим, так и приватным).
- 6) Типы и функции OpenMP определены во включаемом файле <omp.h>.
- 7) Кроме обычных, возможны также "вложенные" (nested) замки – вместо логических переменных используются целые числа, и нить, уже захватившая замок, при повторном захвате может увеличить это число.

Пример распараллеливания for-цикла в C

```
#pragma omp parallel for private(i)
#pragma omp shared(x, y, n) reduction(+: a, b)
for (i=0; i<n; i++)
{
    a = a + x[i];
    b = b + y[i];
}
```

Опция *schedule*

Управляет распределением работы между нитями в конструкции распределения работы цикла.

schedule тип([, chunk])

Опция задаёт, каким образом итерации цикла распределяются между нитями; Задаётся вид алгоритма планирования и, если необходимо, числовой параметр алгоритма (обычно размер блока пространства итераций).

В опции *schedule* параметр *type* задаёт следующий тип распределения итераций:

1. *static* – блочно-циклическое распределение итераций цикла; размер блока – *chunk*. Первый блок из *chunk* итераций выполняет нулевая нить, второй блок – следующая и т.д. до последней нити, затем распределение снова начинается с нулевой нити. Если значение *chunk* не указано, то всё множество итераций делится на непрерывные куски примерно одинакового размера (конкретный способ зависит от реализации), и полученные порции итераций распределяются между нитями.
2. *dynamic* – динамическое распределение итераций с фиксированным размером блока: сначала каждая нить получает *chunk* итераций (по умолчанию *chunk*=1), та нить, которая заканчивает выполнение своей порции итераций, получает первую свободную порцию из *chunk* итераций. Освободившиеся нити получают новые порции итераций до тех пор, пока все порции не будут исчерпаны. Последняя порция может содержать меньше итераций, чем все остальные.
3. *guided* – динамическое распределение итераций, при котором размер порции уменьшается с некоторого начального значения до величины *chunk* (по умолчанию *chunk*=1) пропорционально количеству ещё не распределённых итераций, делённому на количество нитей, выполняющих цикл. Размер первоначально выделяемого блока зависит от

реализации. В ряде случаев такое распределение позволяет аккуратнее разделить работу и сбалансировать загрузку нитей. Количество итераций в последней порции может оказаться меньше значения `chunk`.

4. `auto` – способ распределения итераций выбирается компилятором и/или системой выполнения.

Параметр `chunk` при этом не задаётся.

`runtime` – способ распределения итераций выбирается во время работы программы по значению переменной среды `OMP_SCHEDULE`. Параметр `chunk` при этом не задаётся.

Директива *parallel*

l. 97

Директива `parallel` создает параллельную область для следующего за ней структурированного блока, параллельная область задаётся при помощи записи: l. 99

```
#pragma omp parallel опция[[[,] опция]...]структурированный  
    блок
```

l. 104

Возможные опции:

12. `if` (условие) – выполнение параллельной области по условию. Вхождение в параллельную область осуществляется только при выполнении некоторого условия. Если условие не выполнено, то директива не срабатывает и продолжается обработка программы в прежнем режиме;
13. `num_threads` (целочисленное выражение) – явное задание количества нитей, которые будут выполнять параллельную область; по умолчанию выбирается последнее значение, установленное с помощью функции `omp_set_num_threads()`, или значение переменной `OMP_NUM_THREADS`;
14. `default(shared|none)` – всем переменным в параллельной области, которым явно не назначен класс, будет назначен класс `shared`; `none` означает, что всем переменным в параллельной области класс должен быть назначен явно;
15. `private` (список) – задаёт список переменных, для которых порождается локальная копия в каждой нити; начальное значение локальных копий переменных из списка не определено;
16. `firstprivate` (список) – задаёт список переменных, для которых порождается локальная копия в каждой нити; локальные копии переменных инициализируются значениями этих переменных в нити-мастере;
17. `shared` (список) – задаёт список переменных, общих для всех нитей;
18. `copyin` (список) – задаёт список переменных, объявленных как `threadprivate`, которые при входе в параллельную область инициализируются значениями соответствующих переменных в нити-мастере;
19. `reduction` (оператор:список) – задаёт оператор и список общих переменных; для каждой переменной создаются локальные копии в каждой нити; локальные копии инициализируются соответственно типу оператора (для аддитивных операций – 0 или его аналоги, для мультипликативных операций – 1 или её аналоги); над локальными копиями переменных после выполнения всех операторов параллельной области выполняется заданный оператор; оператор это: `+`, `*`, `-`, `&`, `|`, `^`, `&&`, `||`; порядок выполнения операторов не определён, поэтому результат может отличаться от запуска к запуску.

l. 116

Эта директива сообщает компилятору, что структурированный блок кода должен быть выполнен параллельно, в нескольких потоках. Каждый поток будет выполнять один и тот же поток команд, но не один и тот же набор команд – все зависит от операторов, управляющих логикой программы, таких как `if-else`. l. 118

При входе в параллельную область порождаются новые `OMP_NUM_THREADS-1` нитей, каждая нить получает свой уникальный номер, причём порождающая нить получает номер 0 и становится основной нитью группы (“мастером”). Остальные нити получают в качестве номера целые числа с 1 до `OMP_NUM_THREADS-1`. Количество нитей, выполняющих данную параллельную область, остаётся неизменным до момента выхода из области. При выходе из параллельной области производится неявная синхронизация и уничтожаются все нити, кроме породившей. l. 120

Если один из потоков параллельной области встречает другую директиву `parallel`, то он создает новую группу потоков, согласно правилам, и становится основной нитью новой группы. l. 122

Если выполнение потока аварийно прерывается внутри параллельной области, то также прерывается выполнение всех потоков во всех группах. Порядок прерывания работы потоков не

определен. Вся работа, проделанная группой до последней барьерной синхронизации, гарантированно будет выполнена. Объем выполненной работы, проделанной каждым потоком после последней барьерной синхронизации, до аварийного завершения работы потоков не определен. l. 124

Все порождённые нити исполняют один и тот же код, соответствующий параллельной области. Предполагается, что в SMP-системе нити будут распределены по различным процессорам (однако это, как правило, находится в ведении операционной системы). l. 126

Во время исполнения любой поток может приостановить выполнение своей неявной задачи в точке планирования задач (task scheduling point) и переключиться на выполнение любой явно-сгенерированной задачи прежде чем возобновить выполнение неявной задачи. l. 128

Нить может узнать свой номер с помощью вызова библиотечной функции

`omp_get_thread_num`. l. 130

Очень часто параллельная область не содержит ничего, кроме конструкции разделения работы (т.е. конструкция разделения работы тесно вложена в параллельную область). В этом случае можно указывать не две директивы, а указать одну комбинированную. l. 132

Например, если внутри параллельной области содержится только Один параллельный цикл или одна конструкция `sections`, то можно использовать укороченную запись: `parallel for` или `parallel sections`. При этом допустимо указание всех опций этих директив, за исключением опции `nowait`. l. 134

Ограничения для директивы *parallel* следующие:

- Программа не должна зависеть от какого-либо порядка определения опций параллельной директивы, или от каких-либо побочных эффектов определения опций;
- Только одна опция `if` может присутствовать в директиве;
- Только одна опция `num_threads` может присутствовать в директиве. Выражение в опции `num_threads` должно быть целочисленным;
- Бросок исключения выполненный внутри параллельной области должен вызывать обработку исключения в рамках одной параллельной области, и той же нити, которая бросила исключение.

l. 143

Директива *for*

l. 145

Если в параллельной области встретился оператор цикла, то, согласно общему правилу, он будет выполнен всеми нитями текущей группы, то есть каждая нить выполнит все итерации данного цикла. Для распределения итераций цикла между различными нитями можно использовать директиву `for`. l. 147

```
#pragma omp for опция[ [, ] опция] ... ]цикл
for
```

l. 152

Эта директива относится к идущему следом за данной директивой блоку, включающему оператор `for`. l. 154

Возможные опции:

- `private` (список) – задаёт список переменных, для которых порождается локальная копия в каждой нити; начальное значение локальных копий переменных из списка не определено;
- `firstprivate` (список) – задаёт список переменных, для которых порождается локальная копия в каждой нити; локальные копии переменных инициализируются значениями этих переменных в нити-мастере;
- `lastprivate` (список) – переменным, перечисленным в списке, присваивается результат с последнего витка цикла;
- `reduction` (оператор:список) – задаёт оператор и список общих переменных; для каждой переменной создаются локальные копии в каждой нити; локальные копии инициализируются соответственно типу оператора (для аддитивных операций – 0 или его аналоги, для мультипликативных операций – 1 или её аналоги); над локальными копиями переменных после завершения всех итераций цикла выполняется заданный оператор; оператор это: `+`, `*`, `-`, `&`, `|`, `^`, `&&`, `|`; порядок выполнения операторов не определён, поэтому результат может отличаться от запуска к запуску;

- `schedule(type[, chunk])` – опция задаёт, каким образом итерации цикла распределяются между нитями;
- `collapse(n)` – опция указывает, что `n` последовательных тесновложенных циклов ассоциируется с данной директивой; для циклов образуется общее пространство итераций, которое делится между нитями; если опция `collapse` не задана, то директива относится только к одному непосредственно следующему за ней циклу;
- `ordered` – опция, говорящая о том, что в цикле могут встречаться директивы `ordered`; в этом случае определяется блок внутри тела цикла, который должен выполняться в том порядке, в котором итерации идут в последовательном цикле;
- `nowait` – в конце параллельного цикла происходит неявная барьерная синхронизация параллельно работающих нитей: их дальнейшее выполнение происходит только тогда, когда все они достигнут данной точки; если в подобной задержке нет необходимости, опция `nowait` позволяет нитям, уже дошедшим до конца цикла, продолжить выполнение без синхронизации с остальными.

l. 167

На вид параллельных циклов накладываются достаточно жёсткие ограничения. В частности, предполагается, что корректная программа не должна зависеть от того, какая именно нить какую итерацию параллельного цикла выполнит. Нельзя использовать побочный выход из параллельного цикла. Размер блока итераций, указанный в опции `schedule`, не должен изменяться в рамках цикла. l. 169

Эти требования введены для того, чтобы OpenMP мог при входе в цикл точно определить число итераций. Если директива параллельного выполнения стоит перед гнездом циклов, завершающихся одним оператором, то директива действует только на самый внешний цикл. Итеративная переменная распределяемого цикла по смыслу должна быть локальной, поэтому в случае, если она специфицирована общей, то она неявно делается локальной при входе в цикл. После завершения цикла значение итеративной переменной цикла не определено, если она не указана в опции `lastprivate`. l. 171

Следующий пример демонстрирует использование директивы `for`. В последовательной области инициализируются три исходных массива `A`, `B`, `C`. В параллельной области данные массивы объявлены общими. Вспомогательные переменные `i` и `n` объявлены локальными. Каждая нить присвоит переменной `n` свой порядковый номер. Далее с помощью директивы `for` определяется цикл, итерации которого будут распределены между существующими нитями. На каждой `i`-ой итерации данный цикл сложит `i`-ые элементы массивов `A` и `B` и результат запишет в `i`-ый элемент массива `C`. Также на каждой итерации будет напечатан номер нити, выполнившей данную итерацию. l. 173

```
#include <stdio.h>
#include <omp.h>
```

```
int main(int argc, char *argv[])
{
    int A[10], B[10], C[10], i, n;
    // Заполним исходные массивы
    for (i = 0; i < 10; i++)
    {
        A[i] = i;
        B[i] = 2 * i;
        C[i] = 0;
    }

    #pragma omp parallel shared(A, B, C) private(i, n)
    {
        // Получим номер текущей нити
        n = omp_get_thread_num();

        #pragma omp for
        for (i = 0; i < 10; i++)
        {
            C[i] = A[i] + B[i];
```

```

        printf("Нить %d сложила элементы с номером %d\n", n, i);
    }
}
}
l. 203

```

Директива *single*

l. 205

Если в параллельной области какой-либо участок кода должен быть выполнен лишь один раз, то его нужно выделить директивой *single*. l. 207

```
#pragma omp single опция[ [[,] опция]...]структурированный
    блок
```

l. 212

Возможные опции:

- **private** (список) – задаёт список переменных, для которых порождается локальная копия в каждой нити; начальное значение локальных копий переменных из списка не определено;
- **firstprivate** (список) – задаёт список переменных, для которых порождается локальная копия в каждой нити; локальные копии переменных инициализируются значениями этих переменных в нити-мастере;
- **copyprivate** (список) – после выполнения нити, содержащей конструкцию *single*, новые значения переменных списка будут доступны всем одноименным частным переменным (*private* и *firstprivate*), описанным в начале параллельной области и используемым всеми её нитями; опция не может использоваться совместно с опцией *nowait*; переменные списка не должны быть перечислены в опциях *private* и *firstprivate* данной директивы *single*;
- **nowait** – после выполнения выделенного участка происходит неявная барьерная синхронизация параллельно работающих нитей: их дальнейшее выполнение происходит только тогда, когда все они достигнут данной точки; если в подобной задержке нет необходимости, опция *nowait* позволяет нитям, уже дошедшим до конца участка, продолжить выполнение без синхронизации с остальными.

l. 220

В параллельных регионах часто встречаются блоки кода, доступ к которым желательно предоставлять только одному потоку, – например, блоки кода, отвечающие за запись данных в файл. Во многих таких ситуациях не имеет значения, какой поток выполнит код, важно лишь, чтобы этот поток был единственным. Для этого в OpenMP служит директива *single*. l. 222

Какая именно нить будет выполнять выделенный участок программы, не специфицируется. Одна нить будет выполнять данный фрагмент, а все остальные нити будут ожидать завершения её работы, если только не указана опция *nowait*. l. 224

Следующий пример иллюстрирует применение опции *copyprivate*. В данном примере переменная *n* объявлена в параллельной области как локальная. Каждая нить присвоит переменной *n* значение, равное своему порядковому номеру, и напечатает данное значение. В области *single* одна из нитей присвоит переменной *n* значение 100, и на выходе из области это значение будет присвоено переменной *n* на всех нитях. В конце параллельной области значение *n* печатается ещё раз и на всех нитях оно равно 100. l. 226

```
#include <stdio.h>
#include <omp.h>
```

```

int main(int argc, char *argv[])
{
    int n;

    #pragma omp parallel private(n)
    {
        n = omp_get_thread_num();
        printf("Значение n начало(): %d\n", n);

        #pragma omp single copyprivate(n)
        {
            n = 100;

```

```

    }

    printf("Значение n конец(): %d\n", n);
}

```

l. 249

Ограничения для директивы *single* следующие:

- Опция `copyprivate` не должна использоваться вместе с опцией `nowait`;
- Только одна опция `nowait` может быть использована в директиве *single*;
- Исключение брошенное в области *single* должно быть обработано в рамках одной области *single*, одной и той же нитью.

l. 257

Директива *sections*

l. 259

Директива *sections* используется для задания конечного (неитеративного) параллелизма. Директива *sections* содержит набор структурированных блоков, которые распределяются по потокам в группе. Каждый структурированный блок выполняется один раз одним из потоков в группе. l. 261

```

#pragma omp sections опция[[[,] опция] ...]
{
#pragma omp sectionструктурированный
    блок
#pragma omp sectionструктурированный
    блок
...
}

```

l. 272

При выполнении этого кода OpenMP сначала создает группу потоков, а затем распределяет между ними обработку итераций цикла, после выполнения которого потоки начинают параллельную обработку оставшихся разделов кода. Если количество разделов программного кода будет больше числа потоков, обработка нескольких разделов будет отложена до тех пор, пока не появятся свободные потоки. В отличие от планирования циклов, распределение нагрузки между потоками при обработке параллельных разделов кода осуществляется и контролируется OpenMP.

Программисту остается только выбрать, какие переменные будут общими, а какие – индивидуальными, и предусмотреть выражения уменьшения аналогично сегменту с организацией циклов. l. 274

Возможные опции:

- `private` (список) – задаёт список переменных, для которых порождается локальная копия в каждой нити; начальное значение локальных копий переменных из списка не определено;
- `firstprivate` (список) – задаёт список переменных, для которых порождается локальная копия в каждой нити; локальные копии переменных инициализируются значениями этих переменных в нити-мастере;
- `lastprivate` (список) – переменным, перечисленным в списке, присваивается результат, полученный в последней секции;
- `reduction` (оператор:список) – задаёт оператор и список общих переменных; для каждой переменной создаются локальные копии в каждой нити; локальные копии инициализируются соответственно типу оператора (для аддитивных операций – 0 или его аналоги, для мультипликативных операций – 1 или её аналоги); над локальными копиями переменных после завершения всех секций выполняется заданный оператор; оператор это: `+`, `*`, `-`, `&`, `|`, `^`, `&&`, `||`; порядок выполнения операторов не определён, поэтому результат может отличаться от запуска к запуску;
- `nowait` – в конце блока секций происходит неявная барьерная синхронизация параллельно работающих нитей: их дальнейшее выполнение происходит только тогда, когда все они достигнут данной точки; если в подобной задержке нет необходимости, опция `nowait` позволяет нитям, уже дошедшим до конца своих секций, продолжить выполнение без синхронизации с остальными.

l. 283

Директива `section` задаёт участок кода внутри секции `sections` для выполнения одной нитью. l. 285

```
#pragma omp section
```

l. 289

Перед первым участком кода в блоке `sections` директива `section` не обязательна. Какие именно нити будут задействованы для выполнения какой секции, не специфицируется. Если количество нитей больше количества секций, то часть нитей для выполнения данного блока секций не будет задействована. Если количество нитей меньше количества секций, то некоторым (или всем) нитям достанется более одной секции. l. 291

Следующий пример демонстрирует использование опции `lastprivate`. В данном примере опция `lastprivate` используется вместе с директивой `sections`. Переменная `n` объявлена как `lastprivate` переменная. Три нити, выполняющие секции `section`, присваивают своей локальной копии `n` разные значения. По выходе из области `sections` значение `n` из последней секции присваивается локальным копиям во всех нитях, поэтому все нити напечатают число 3. Это же значение сохранится для переменной `n` и в последовательной области. l. 293

```
#include <stdio.h>
```

```
#include <omp.h>
```

```
int main(int argc, char *argv[])
{
    int n = 0;

    #pragma omp parallel
    {
        #pragma omp sections lastprivate(n)
        {
            #pragma omp section
            {
                n = 1;
            }

            #pragma omp section
            {
                n = 2;
            }

            #pragma omp section
            {
                n = 3;
            }
        }

        printf("Значение n на нити %d: %d\n",
            omp_get_thread_num(), n);
    }

    printf("Значение n в последовательной области: %d\n", n);
}
```

l. 329

Директива *master*

l. 331

Директивы `master` выделяют участок кода, который будет выполнен только нитью-мастером. Остальные нити просто пропускают данный участок и продолжают работу с оператора, расположенного следом за ним. Неявной синхронизации данная директива не предполагает. l. 333

```
#pragma omp master
```

l. 337

Следующий пример демонстрирует применение директивы `master`. Переменная `n` является локальной, то есть каждая нить работает со своим экземпляром. Сначала все нити присвоят

переменной `n` значение 1. Потом нить-мастер присвоит переменной `n` значение 2, и все нити напечатают значение `n`. Затем нитьмастер присвоит переменной `n` значение 3, и снова все нити напечатают значение `n`. Видно, что директиву `master` всегда выполняет одна и та же нить. В данном примере все нити выведут значение 1, а нить-мастер сначала выведет значение 2, а потом - значение 3.

```
1. 339 #include <stdio.h>

int main(int argc, char *argv[])
{
    int n;

    #pragma omp parallel private(n)
    {
        n = 1;

        #pragma omp master
        {
            n = 2;
        }

        printf("Первое значение n: %d\n", n);

        #pragma omp barrier

        #pragma omp master
        {
            n = 3;
        }

        printf("Второе значение n: %d\n", n);
    }
}
1. 369
```

Директива *critical*

1. 371

С помощью директив `critical` оформляется критическая секция программы. Критическая секция запрещает одновременное исполнение структурированного блока более чем одним потоком.

```
1. 373 #pragma omp critical имя[()]структурированный
      блок
```

1. 378

В каждый момент времени в критической секции может находиться не более одной нити. Если критическая секция уже выполняется какой-либо нитью, то все другие нити, выполнившие директиву для секции с данным именем, будут заблокированы, пока вошедшая нить не закончит выполнение данной критической секции. Как только работавшая нить выйдет из критической секции, одна из заблокированных на входе нитей войдет в неё. Если на входе в критическую секцию стояло несколько нитей, то случайным образом выбирается одна из них, а остальные заблокированные нити продолжают ожидание.

1. 380 Все неименованные критические секции условно ассоциируются с одним и тем же именем. Все критические секции, имеющие одно и то же имя, рассматриваются единой секцией, даже если находятся в разных параллельных областях. Побочные входы и выходы из критической секции запрещены.

1. 382 Следующий пример иллюстрирует применение директивы `critical`. Переменная `n` объявлена вне параллельной области, поэтому по умолчанию является общей. Критическая секция позволяет разграничить доступ к переменной `n`. Каждая нить по очереди присвоит `n` свой номер и затем напечатает полученное значение.

```
1. 384 #include <stdio.h>
#include <omp.h>
```



```

int main(int argc, char *argv[])
{
    int n;

    #pragma omp parallel
    {
        #pragma omp critical
        {
            n = omp_get_thread_num();
            printf("Нить %d\n", n);
        }
    }
}

```

l. 402

Если бы в примере не была указана директива `critical`, результат выполнения программы был бы непредсказуем. С директивой `critical` порядок вывода результатов может быть произвольным, но это всегда будет набор одних и тех же чисел от 0 до `OMP_NUM_THREADS-1`. Конечно, подобного же результата можно было бы добиться другими способами, например, объявив переменную `n` локальной, тогда каждая нить работала бы со своей копией этой переменной. Однако в исполнении этих фрагментов разница существенная. l. 404

Если есть критическая секция, то в каждый момент времени фрагмент будет обрабатываться лишь какой-либо одной нитью. Остальные нити, даже если они уже подошли к данной точке программы и готовы к работе, будут ожидать своей очереди. Если критической секции нет, то все нити могут одновременно выполнить данный участок кода. С одной стороны, критические секции предоставляют удобный механизм для работы с общими переменными. Но с другой стороны, пользоваться им нужно осмотрительно, поскольку критические секции добавляют последовательные участки кода в параллельную программу, что может снизить её эффективность. l. 406

Директива *barrier*

l. 408

Самый распространенный способ синхронизации в OpenMP – барьер. Он оформляется с помощью директивы `barrier`. Директива `barrier` дает всем потокам указание ожидать друг друга перед тем, как они продолжат выполнение за барьером. l. 410

```
#pragma omp barrier
```

l. 414

Нити, выполняющие текущую параллельную область, дойдя до этой директивы, останавливаются и ждут, пока все нити не дойдут до этой точки программы, после чего разблокируются и продолжают работать дальше. Кроме того, для разблокировки необходимо, чтобы все синхронизируемые нити завершили все порождённые ими задачи (task). l. 416

Следующий пример демонстрирует применение директивы `barrier`. Директива `barrier` используется для упорядочивания вывода от работающих нитей. Выдачи с разных нитей "Сообщение 1" и "Сообщение 2" могут перемежаться в произвольном порядке, а выдача "Сообщение 3" со всех нитей придёт строго после двух предыдущих выдач. l. 418

```

#include <stdio.h>
#include <omp.h>
int main(int argc, char *argv[])
{
    #pragma omp parallel
    {
        printf("Сообщение 1\n");

        printf("Сообщение 2\n");

        #pragma omp barrier

        printf("Сообщение 3\n");
    }
}

```

}

Цель MPI

- ▶ Основная цель:
 - Обеспечение переносимости исходных кодов
 - Эффективная реализация
- ▶ Кроме того:
 - Большая функциональность
 - Поддержка неоднородных параллельных архитектур

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.
- ▶ 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.
- ▶ 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

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
- Индексированный 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_Xxxx(parameter,...);  
MPI_Xxxx(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;  
}
```

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();  
}
```

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.

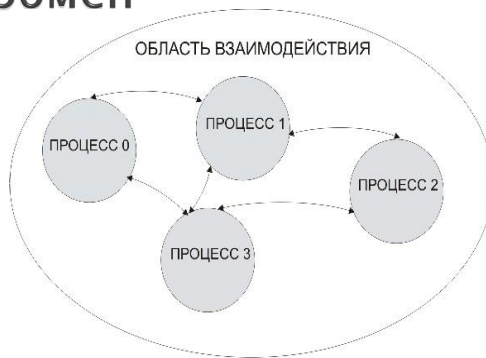
Выполнение MPI-программы

- ▶ 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

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 comm, int*errorcode)**
завершает работу программы

Двухточечный (point-to-point, p2p) обмен



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

Условия успешного взаимодействия точка-точка

- ▶ Отправитель должен указать правильный rank получателя
- ▶ Получатель должен указать верный rank отправителя
- ▶ Одинаковый коммутатор
- ▶ 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)`

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)

- ▶ **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.
- 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

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.

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.
- ▶ 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 большой контроль над распределением памяти. Этот режим удобен и для отладки, поскольку причину переполнения буфера определить легче, чем причину тупика.

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:

После завершения работы с буфером его необходимо отключить:

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


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,
```

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)
```

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,
```

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.



- ▶ 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.

Неблокирующий обмен выполняется в два этапа:

1. инициализация обмена;
2. проверка завершения обмена.

- ▶ 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 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

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:

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.

Неблокирующие процедуры проверки

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

Проверка статуса операции приема сообщения

- ▶ Блокирующая проверка:

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

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

Коллективные операции

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

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)
```

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 recvttype,
int root, MPI_Comm comm)
```

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.

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);
```