ФЕДЕРАЛЬНОЕ АГЕНТСТВО СВЯЗИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ БЮДЖЕТНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ «СИБИРСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ТЕЛЕКОММУНИКАЦИЙ И ИНФОРМАТИКИ»

К.В. Павский

Протоколы TCP/IP и разработка сетевых приложений

Учебное пособие

Новосибирск 2013

УДК 681.3.07

Павский К.В. Протоколы TCP/IP и разработка сетевых приложений: Учебное пособие. – Новосибирск: СибГУТИ, 2013. – 130 с.

Пособие предназначено для студентов, обучающихся по направлению 230100 «Информатика и вычислительная техника» и изучающих дисциплину «Сетевое программное обеспечение». В нём содержится материал, предназначенный для проведения лекционных и практических занятий по указанному учебному курсу с целью изучения протоколов TCP/IP и разработки сетевых приложений.

Кафедра вычислительных систем

Ил.32, табл.13, список лит. – 13 наим.

Рецензенты:
к.т.н., доцент кафедры ПМиК ФГОБУ ВПО «СибГУТИ» Ситняковская Е.И.,
к.ф-м.н., с.н.с. ИНГГ СО РАН Сибиряков Е.Б.,
руководитель отдела ООО «Программные технологии» Канониров А.О.

Для студентов, обучающихся по направлению 230100 «Информатика и вычислительная техника».

Утверждено редакционно-издательским советом ФГОБУ ВПО «СибГУТИ» в качестве учебного пособия.

ОГЛАВЛЕНИЕ

ОГЛАВЛЕНИЕ 3
ВВЕДЕНИЕ 5
ГЛАВА 1. АРХИТЕКТУРА TCP/IP 6
1.1. Многоуровневый подход. Протокол. Интерфейс. Стек протоколов 6
1.2. Модель OSI 9
1.3. Стек TCP/IP 13
Контрольные вопросы 15
ГЛАВА 2. СЕТЕВОЙ УРОВЕНЬ 16
2.1. IP адресация. Маски 16
2.2. IP протокол v.4. Заголовок пакета 19
2.3. ICMP протокол 22
2.3.1. Сообщения об ошибках ICMP 23
2.3.2. Исследование MTU по пути 29
2.3.3. Сообщения запросов ICMP 31
2.4. Маршрутизация в сетях TCP/IP 33
2.5. Протоколы маршрутизации 37
2.5.1. RIP протокол 37
2.5.2. OSPF протокол 41
2.5.3. EGP протокол 44
2.5.4. BGP протокол 47
Контрольные вопросы 50
ГЛАВА 3. ПРОТОКОЛЫ ТРАНСПОРТНОГО УРОВНЯ 51
3.1. Протокол UDP. Заголовок пакета 51
3.2. Протокол TCP 52
3.2.1. Заголовок пакета 52
3.2.2. Вариант максимального размера сегмента 54
3.2.3. Производительность 54
3.2.4. Медленный старт 56
3.2.5. Синдром бестолкового окна 57
3.2.6. Алгоритм Нейгла 58
3.2.7. Задержанный ACK 58
3.2.8. Тайм-аут повторной пересылки 59
3.2.9. Вычисления после повторной отправки 61
3.2.10. Экспоненциальное торможение 61
3.2.11. Снижение перегрузок за счет уменьшения пересылаемых по сети данных 62
3.2.12. Дублированные ACK 63
3.2.13. Барьеры для производительности 63
3.2.14. Функции TCP 65
3.3. Протокол SCTP 65
3.3.1. Заголовок пакета SCTP 66
3.3.2. Множественная адресация 66
3.3.3. Многопотоковая передача данных 67
3.3.4. Безопасность устанавливаемого подключения 69

3.3.5. Формирование кадров сообщения 70
3.3.6. Настраиваемая неупорядоченная передача данных 70
3.3.7. Поэтапное завершение передачи данных 71
Контрольные вопросы 72
ГЛАВА 4. ПРОТОКОЛЫ ПРИКЛАДНОГО УРОВНЯ 73
4.1. Протокол POP3 73
4.2. Протокол SMTP 78
4.3. Протокол FTP 79
4.4. Протокол TFTP 95
Контрольные вопросы 98
ГЛАВА 5. РАЗРАБОТКА СЕТЕВЫХ ПРИЛОЖЕНИЙ 99
5.1. Гнезда и интерфейс транспортного уровня 99
5.1.1. Функция socket 102
5.1.2. Функция bind 103
5.1.3. Функция listen 104
5.1.4. Функция connect 104
5.1.5. Функция accept 105
5.1.6. Функция send 105
5.1.7. Функция sendto 106
5.1.8. Функция recv 106
5.1.9. Функция recvfrom 107
5.1.10. Функция shutdown 107
5.2. Разработка клиент-серверных приложений 108
5.2.1. Однопоточная серверная программа TCP 108
5.2.2. Клиентская программа TCP 113
5.2.3. Пример реализации многопотоковой передачи по протоколу SCTP 115
5.3. Применение процессов для обеспечения параллельной работы сервера 120
5.4. Применение потоков для обеспечения параллельной работы сервера 121
5.5. Однопотоковые псевдопараллельные сервера 126
Контрольные вопросы 128
СПИСОК ЛИТЕРАТУРЫ 130

ВВЕДЕНИЕ

Сегодня компьютерные сети становятся неотъемлемой частью при решении многих задач, как рабочих, так и житейских. Уже становится обыденным иметь при себе сотовый телефон, а информацию получать через Интернет. Более того для некоторых людей это стало неотъемлемой частью, без которой они уже не могут представить свое существование.

Именно поэтому высока актуальность подготовки квалифицированных специалистов понимающих суть организации сетей, способных их администрировать и создавать сетевые приложения.

Учебное пособие содержит материал, составляющий основу курса «Сетевое программное обеспечение» направления «Информатика и вычислительная техника» (квалификация бакалавр, специалист, магистр), который также может быть полезен при изучении сетевых технологий и программирования.

Первый раздел содержит описание основных понятий организации взаимодействия между устройствами в сети.

Второй раздел посвящен сетевому уровню TCP/IP. В нем дается представление об IP-адресации версии 4 и организации подсетей с помощью масок. Так же дано описание протокола интернета IP, протокола ошибок ICMP и протоколов маршрутизации (RIP, OSPF, EGP, BGP).

Третий раздел посвящен протоколам транспортного уровня (UDP, TCP и SCTP). Здесь описаны заголовки их пакетов, алгоритмы, повышающие производительность протокола TCP, достоинства протокола SCTP и сравнение его с протоколами UDP и TCP.

В четвертом разделе приводится описание некоторых протоколов прикладного уровня (POP3, SMTP, FTP, TFTP).

В пятом разделе описаны интерфейсы прикладного программирования (application programming interface — API). Приводятся примеры и разбор кодов программ клиент-сервера. Даны представления об организации параллельных серверов.

ГЛАВА 1. АРХИТЕКТУРА TCP/IP

С самых первых дней использования компьютеров хосты обменивались информацией с непосредственно подключенными к ним устройствами, такими, как устройство чтения перфокарт или устройство печати [1]. Интерактивное использование компьютеров потребовало сначала локального, а затем удаленного подключения терминалов конечных пользователей. Далее последовало объединение нескольких компьютеров в рамках одной организации, что было вызвано необходимостью обеспечения обмена данными между компьютерами или потребностью пользователя одного из компьютеров получить доступ к другому компьютеру.

Разработчики компьютерных систем откликнулись на эти потребности созданием соответствующего аппаратного и программного обеспечения. Однако средства того времени обладали следующими недостатками:

- программные средства были лицензионными (коммерческие и не доступные для бесплатного использования) и работали только с оборудованием того же производителя;
- поддерживался только ограниченный набор локальных и региональных сетей;
- часто программные средства были очень сложными и требовали различных диалектов для работы с разными устройствами;
- отсутствие гибкости не позволяло объединить уже существующие сети недорогими и простыми в работе средствами.

Эта ситуация изменилась с появлением протокола управления передачей/протокола Интернета (Transmission Control Protocol/Internet Protocol — TCP/IP) и порождаемыми им технологиями маршрутизации.

Прежде, чем начнем рассматривать протоколы семейства TCP/IP, рассмотрим некоторые основные понятия организации взаимодействия между устройствами в сети.

1.1. Многоуровневый подход. Протокол. Интерфейс. Стек протоколов

Организация взаимодействия между устройствами в сети является сложной задачей. Для ее решения используется принцип декомпозиции, то есть разбиение одной сложной задачи на несколько более простых задач-модулей. Процедура декомпозиции включает в себя четкое определение функций каждого модуля, решающего отдельную задачу, и интерфейсов между ними. При декомпозиции используют многоуровневый подход, который заключается в следующем [2, 3]:

1) Все множество модулей разбивают на уровни.
2) Уровни образуют иерархию, то есть имеются вышележащие и нижележащие уровни.
3) Множество модулей, составляющих каждый уровень, сформировано таким образом, что для выполнения своих задач они обращаются с запросами

только к модулям непосредственно примыкающего нижележащего уровня.

4) Результаты работы всех модулей, принадлежащих некоторому уровню, могут быть переданы только модулям соседнего вышележащего уровня.
5) Определены функции каждого уровня и интерфейсы между уровнями. Интерфейс определяет набор функций, которые нижележащий уровень предоставляет вышележащему.

В результате иерархической декомпозиции достигается относительная независимость уровней, а значит, и возможность их легкой замены.

На рисунке 1.1 показана модель взаимодействия двух узлов. С каждой стороны средства взаимодействия представлены четырьмя уровнями. Процедура взаимодействия этих двух узлов может быть описана в виде набора правил взаимодействия каждой пары соответствующих уровней обеих участвующих сторон.

Опр. *Формализованные правила, определяющие последовательность и формат сообщений, которыми обмениваются сетевые компоненты, лежащие на одном уровне, но в разных узлах, называются протоколом*.

Модули, реализующие протоколы соседних уровней и находящиеся в одном узле, также взаимодействуют друг с другом в соответствии с четко определенными правилами и с помощью стандартизованных форматов сообщений. Эти правила принято называть интерфейсом. *Интерфейс определяет набор сервисов, предоставляемый данным уровнем соседнему уровню.* В сущности, протокол и интерфейс выражают одно и то же понятие, но традиционно в сетях за ними закрепили разные области действия: *протоколы определяют правила взаимодействия модулей одного уровня в разных узлах, а интерфейсы – модулей соседних уровней в одном узле.*

Опр. *Иерархически организованный набор протоколов, достаточный для организации взаимодействия узлов в сети, называется стеком коммуникационных протоколов.*

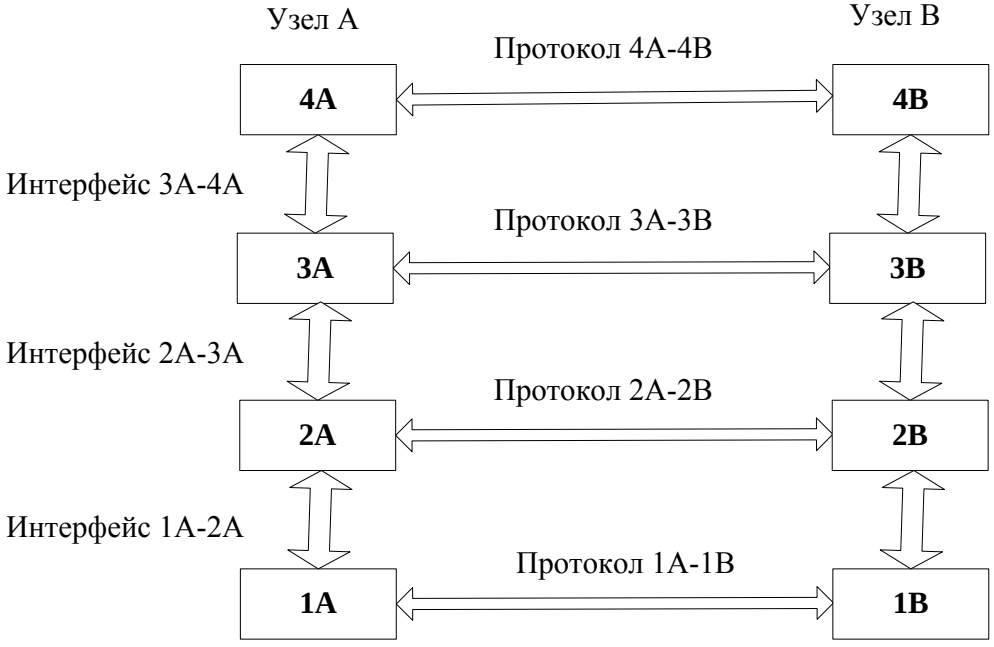


Рис.1.1 Модель взаимодействия двух узлов

Программный модуль, реализующий некоторый протокол, часто для краткости также называют *«протоколом»*. При этом соотношение между протоколом — формально определенной процедурой и протоколом — программным модулем, реализующим эту процедуру, аналогично соотношению между алгоритмом решения некоторой задачи и программой, решающей эту задачу.

Пример. Чтобы еще раз пояснить понятия «протокол» и «интерфейс», рассмотрим пример, не имеющий отношения к вычислительным сетям, а именно обсудим взаимодействие двух предприятий А и В, связанных между собой деловым сотрудничеством. Между предприятиями существуют многочисленные договоренности и соглашения, такие, например, как регулярные поставки продукции одного предприятия другому. В соответствии с этой договоренностью начальник отдела продаж предприятия А регулярно в начале каждого месяца посылает официальное сообщение начальнику отдела закупок предприятия В о том, сколько и какого товара может быть поставлено в этом месяце. В ответ на это сообщение начальник отдела закупок предприятия В посылает в ответ заявку установленного образца на требуемое количество продукции. Возможно, процедура взаимодействия этих начальников включает дополнительные согласования, в любом случае существует установленный порядок взаимодействия, который можно считать «протоколом уровня начальников». Начальники посылают свои сообщения и заявки через своих секретарей. Порядок взаимодействия начальника и секретаря соответствует понятию межуровневого интерфейса «начальник - секретарь». На предприятии А обмен документами между начальником и секретарем идет через специальную папку, а на предприятии В начальник общается с секретарем по факсу. Таким образом, интерфейсы «начальник - секретарь» на этих двух предприятиях отличаются. После того как сообщения переданы секретарям, начальников не волнует, каким образом эти сообщения будут перемещаться дальше — обычной или электронной почтой, факсом или нарочным. Выбор способа передачи — это уровень компетенции секретарей, они могут решать этот вопрос, не уведомляя об этом своих начальников, так как их протокол взаимодействия связан только с передачей сообщений, поступающих сверху, и не касается содержания этих сообщений. При решении других вопросов начальники могут взаимодействовать по другим правилам-протоколам, но это не повлияет на работу секретарей, для которых не важно, какие сообщения отправлять, а важно, чтобы они дошли до адресата. Итак, в данном случае мы имеем дело с двумя уровнями — уровнем начальников и уровнем секретарей, и каждый из них имеет собственный протокол, который может быть изменен независимо от протокола другого уровня. Эта независимость протоколов друг от друга и делает привлекательным многоуровневый подход.

1.2. Модель OSI

В начале 80-х годов ряд международных организаций по стандартизации — ISO, ITU-T и некоторые другие — разработали модель, которая сыграла значительную роль в развитии сетей. Эта модель называется моделью взаимодействия открытых систем (Open System Interconnection, OSI) или моделью OSI. Модель OSI определяет различные уровни взаимодействия систем, дает им стандартные имена и указывает, какие функции должен выполнять каждый уровень. В модели OSI (рис. 1.2) средства взаимодействия делятся на семь уровней: прикладной, представительский, сеансовый, транспортный, сетевой, канальный, физический [2, 3].

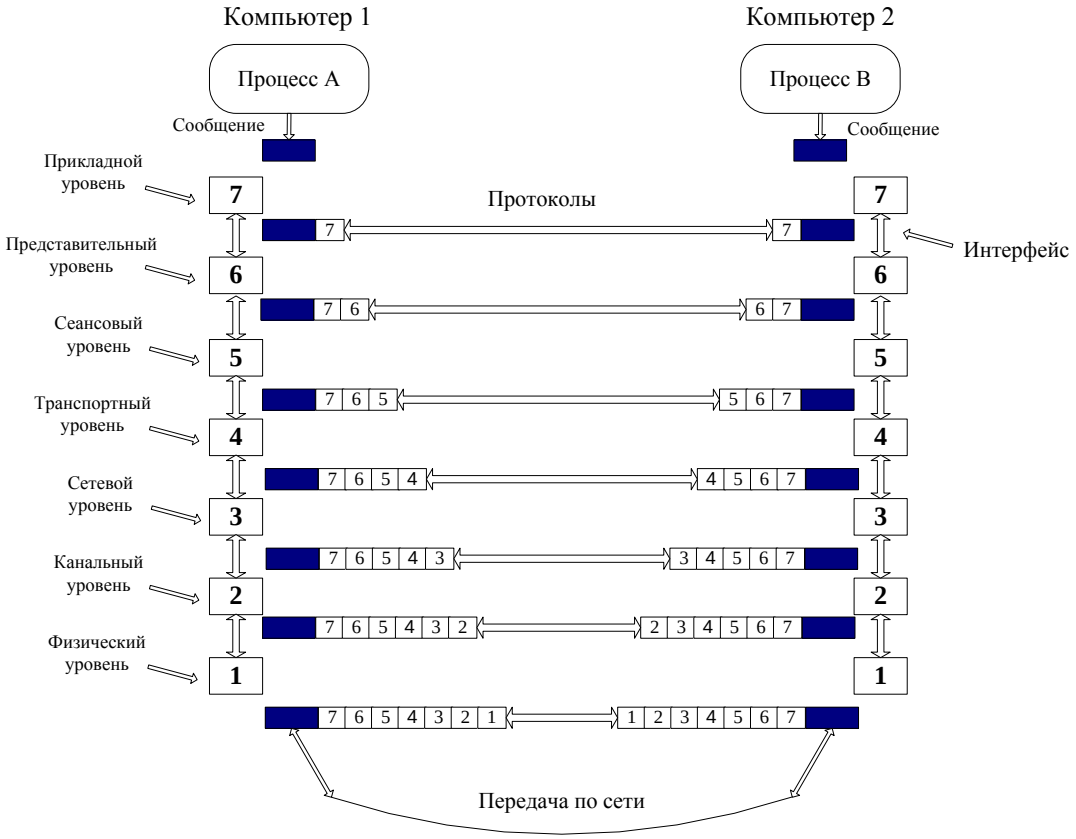


Рис.1.2 Уровни модели OSI

Итак, пусть приложение обращается с запросом к прикладному уровню, например к файловой службе. На основании этого запроса программное обеспечение прикладного уровня формирует сообщение стандартного формата. Обычное сообщение состоит из заголовка и поля данных. Заголовок содержит служебную информацию, которую необходимо передать через сеть прикладному уровню машины-адресата, чтобы сообщить ему, какую работу надо выполнить. После формирования сообщения прикладной уровень направляет его вниз по стеку представительному уровню. Протокол представительного уровня на основании информации, полученной из заголовка прикладного уровня, выполняет требуемые действия и добавляет к сообщению собственную служебную информацию — заголовок представительного уровня, в

котором содержатся указания для протокола представительного уровня машины-адресата. Полученное в результате сообщение передается вниз сеансовому уровню, который в свою очередь добавляет свой заголовок, и т. д. (Некоторые реализации протоколов помещают служебную информацию не только в начале сообщения в виде заголовка, но и в конце, в виде так называемого «концевика»). Наконец, сообщение достигает нижнего, физического уровня, который собственно и передает его по линиям связи машине-адресату. К этому моменту сообщение «обрастает» заголовками всех уровней.

Когда сообщение по сети поступает на машину-адресат, оно принимается ее физическим уровнем и последовательно перемещается вверх с уровня на уровень. Каждый уровень анализирует и обрабатывает заголовок своего уровня, выполняя соответствующие данному уровню функции, а затем удаляет этот заголовок и передает сообщение вышележащему уровню.

Физический уровень

Физический уровень (Physical layer) имеет дело с передачей битов по физическим каналам связи, таким, например, как коаксиальный кабель, витая пара, оптоволоконный кабель или цифровой территориальный канал. К этому уровню имеют отношение характеристики физических сред передачи данных, такие как полоса пропускания, помехозащищенность, волновое сопротивление и другие. Кроме этого, здесь стандартизуются типы разъемов и назначение каждого контакта. Функции физического уровня реализуются во всех устройствах, подключенных к сети. Со стороны компьютера функции физического уровня выполняются сетевым адаптером или последовательным портом.

Пример: Спецификация 10Base-T технологии Ethernet, которая определяет в качестве используемого кабеля неэкранированную витую пару категории 3 с волновым сопротивлением 100 Ом, разъем RJ-45, максимальную длину физического сегмента 100 метров, манчестерский код для представления данных в кабеле, а также некоторые другие характеристики среды и электрических сигналов.

Канальный уровень

На физическом уровне просто пересылаются биты. При этом не учитывается, что в некоторых сетях, в которых линии связи используются (разделяются) попеременно несколькими парами взаимодействующих компьютеров, физическая среда передачи может быть занята. Поэтому одной из задач канального уровня (Data Link layer) является проверка доступности среды передачи. Другой задачей канального уровня является реализация механизмов обнаружения и коррекции ошибок. Для этого на канальном уровне биты группируются в наборы, называемые кадрами (frames).

Состав кадра: первые 8 бит – флаг 01111110 для обнаружения начала кадра; 1-2 байта поле управления – команда или номер кадра; адрес получателя 1 байт; поле данных; поле восстанавливающего кода или контрольная сумма.

Канальный уровень обеспечивает корректность передачи каждого кадра,

помещая специальную последовательность бит в начало и конец каждого кадра, для его выделения, а также вычисляет контрольную сумму, обрабатывая все байты кадра определенным способом и добавляя контрольную сумму к кадру. Когда кадр приходит по сети, получатель снова вычисляет контрольную сумму полученных данных и сравнивает результат с контрольной суммой из кадра. Если они совпадают, кадр считается правильным и принимается. Если же контрольные суммы не совпадают, то фиксируется ошибка. Канальный уровень может не только обнаруживать ошибки, но и исправлять их за счет повторной передачи поврежденных кадров. Необходимо отметить, что функция исправления ошибок не является обязательной для канального уровня, поэтому в некоторых протоколах этого уровня она отсутствует, например, в Ethernet и frame relay.

Пример: Протоколы Ethernet, Token Ring, FDDI, l00VG-AnyLAN.

Сетевой уровень

Сетевой уровень (Network layer) служит для образования единой транспортной системы, объединяющей несколько сетей, причем эти сети могут использовать совершенно различные принципы передачи сообщений между конечными узлами и обладать произвольной структурой связей. На сетевом уровне сам термин сеть наделяют специфическим значением. В данном случае под сетью понимается совокупность компьютеров, соединенных между собой в соответствии с одной из стандартных типовых топологий и использующих для передачи данных один из протоколов канального уровня, определенный для этой топологии.

Внутри сети доставка данных обеспечивается соответствующим канальным уровнем, а вот доставкой данных между сетями занимается сетевой уровень, который и поддерживает возможность правильного выбора маршрута передачи сообщения. Проблема выбора наилучшего пути называется маршрутизацией, и ее решение является одной из главных задач сетевого уровня. Эта проблема осложняется тем, что самый короткий путь не всегда самый лучший. Часто критерием при выборе маршрута является время передачи данных по этому маршруту; оно зависит от пропускной способности каналов связи и интенсивности трафика, которая может изменяться с течением времени.

На сетевом уровне определяются два вида протоколов. Первый вид — сетевые протоколы (routedprotocols) — реализуют продвижение пакетов через сеть. Именно эти протоколы обычно имеют в виду, когда говорят о протоколах сетевого уровня. Однако часто к сетевому уровню относят и другой вид протоколов, называемых протоколами обмена маршрутной информацией или просто протоколами маршрутизации (routingprotocols). С помощью этих протоколов маршрутизаторы собирают информацию о топологии межсетевых соединений. Протоколы сетевого уровня реализуются программными модулями операционной системы, а также программными и аппаратными средствами маршрутизаторов.

На сетевом уровне работают протоколы еще одного типа, которые отвечают за отображение адреса узла, используемого на сетевом уровне, в

локальный адрес сети. Такие протоколы часто называют протоколами разрешения адресов — Address Resolution Protocol, ARP. Иногда их относят не к сетевому уровню, а к канальному, хотя тонкости классификации не изменяют их сути.

Транспортный уровень

Первый уровень, на котором предполагается существование 2-х связанных друг с другом процессов. Транспортный уровень (Transport layer) обеспечивает приложениям или верхним уровням стека — прикладному и сеансовому — передачу данных с той степенью надежности, которая им требуется. Модель OSI определяет пять классов сервиса, предоставляемых транспортным уровнем. Эти виды сервиса отличаются качеством предоставляемых услуг: срочностью, возможностью восстановления прерванной связи, наличием средств мультиплексирования нескольких соединений между различными прикладными протоколами через общий транспортный протокол, способностью к обнаружению и исправлению ошибок передачи, таких как искажение, потеря и дублирование пакетов.

Как правило, все протоколы, начиная с транспортного уровня и выше, реализуются программными средствами конечных узлов сети — компонентами их сетевых операционных систем. Протоколы нижних четырех уровней обобщенно называют сетевым транспортом или транспортной подсистемой, так как они полностью решают задачу транспортировки сообщений с заданным уровнем качества в составных сетях с произвольной топологией и различными технологиями. Остальные три верхних уровня решают задачи предоставления прикладных сервисов на основании имеющейся транспортной подсистемы.

Сеансовый уровень

Сеансовый уровень (Session layer) обеспечивает управление диалогом: фиксирует, какая из сторон является активной в настоящий момент, предоставляет средства синхронизации. Последние позволяют вставлять контрольные точки в длинные передачи, чтобы в случае отказа можно было вернуться назад к последней контрольной точке, а не начинать все с начала. На практике немногие приложения используют сеансовый уровень, и он редко реализуется в виде отдельных протоколов, хотя функции этого уровня часто объединяют с функциями прикладного уровня и реализуют в одном протоколе.

Представительный уровень

Представительный уровень (Presentation layer) имеет дело с формой представления передаваемой по сети информации, не меняя при этом ее содержания. За счет уровня представления информация, передаваемая прикладным уровнем одной системы, всегда понятна прикладному уровню другой системы. На этом уровне может выполняться шифрование и дешифрование данных, благодаря которому секретность обмена данными обеспечивается сразу для всех прикладных служб.

Прикладной уровень

Прикладной уровень (Application layer) — это в действительности просто набор разнообразных протоколов, с помощью которых пользователи сети получают доступ к разделяемым ресурсам, таким как файлы, принтеры или гипертекстовые Web-страницы, а также организуют свою совместную работу, например, с помощью протокола электронной почты.

Существует очень большое разнообразие служб прикладного уровня. Приведем в качестве примера хотя бы несколько наиболее распространенных реализаций файловых служб: NCP в операционной системе Novell NetWare, SMB в Microsoft Windows NT, NFS, FTP и TFTP, входящие в стек TCP/IP.

Сетезависимые и сетенезависимые уровни

Функции всех уровней модели OSI могут быть отнесены к одной из двух групп: либо к функциям, зависящим от конкретной технической реализации сети, либо к функциям, ориентированным на работу с приложениями.

Три нижних уровня — физический, канальный и сетевой — являются сетезависимыми, то есть протоколы этих уровней тесно связаны с технической реализацией сети и используемым коммуникационным оборудованием. Например, переход на оборудование FDDI означает полную смену протоколов физического и канального уровней во всех узлах сети.

Три верхних уровня — прикладной, представительный и сеансовый — ориентированы на приложения и мало зависят от технических особенностей построения сети. На протоколы этих уровней не влияют какие бы то ни было изменения в топологии сети, замена оборудования или переход на другую сетевую технологию. Так, переход от Ethernet на высокоскоростную технологию l00VG-AnyLAN не потребует никаких изменений в программных средствах, реализующих функции прикладного, представительного и сеансового уровней.

Транспортный уровень является промежуточным, он скрывает все детали функционирования нижних уровней от верхних. Это позволяет разрабатывать приложения, не зависящие от технических средств непосредственной транспортировки сообщений.

Модель OSI представляет хотя и очень важную, но только одну из многих моделей коммуникаций. Эти модели и связанные с ними стеки протоколов могут отличаться количеством уровней, их функциями, форматами сообщений, службами, поддерживаемыми на верхних уровнях, и прочими параметрами.

1.3. Стек TCP/IP

Стек TCP/IP был разработан по инициативе Минобороны США более 20 лет назад для связи сети ARPAnet с другими сетями как набор общих протоколов для разнородной вычислительной среды. Большой вклад в развитие стека, внес университет Беркли, реализовав протоколы стека в своей версии ОС UNIX. Популярность этой операционной системы привела к широкому распространению протоколов TCP, IP и других протоколов стека. Сегодня этот

стек используется для связи компьютеров сети Internet, а также в огромном числе корпоративных сетей [1 – 3].

Термин "TCP/IP" обычно обозначает все, что связано с протоколами TCP и IP. Он охватывает целое семейство протоколов, прикладные программы и даже саму сеть. В состав семейства входят протоколы UDP, ARP, ICMP, TELNET, FTP и многие другие. TCP/IP - это технология межсетевого взаимодействия, технология internet. Сеть, которая использует технологию internet, называется "internet". Если речь идет о глобальной сети, объединяющей множество сетей с технологией internet, то ее называют Internet.

Основными протоколами стека, давшими ему название, являются протоколы IP и TCP. К другим протоколам стека относятся: протокол передачи файлов FTP, протокол эмуляции терминала telnet, почтовый протокол SMTP, гипертекстовые сервисы службы WWW (HTTP) и многие другие (см. табл. 1.1).

Причина, по которой TCP/IP столь важен сегодня, заключается в том, что он позволяет самостоятельным сетям подключаться к Internet или объединяться для создания частных интрасетей. Вычислительные сети, составляющие интрасеть, физически подключаются через устройства, называемые маршрутизаторами или IP-маршрутизаторами. Маршрутизатор - это компьютер, который передает пакеты данных из одной сети в другую. В интрасети, работающей на основе TCP/IP, информация передается в виде дискретных блоков, называемых IP-пакетами (IP packets) или IP-дейтаграммами (IP datagrams).

Поскольку стек TCP/IP изначально создавался для глобальной сети Internet, он имеет много особенностей, дающих ему преимущество перед другими протоколами. В частности, TCP/IP способен фрагментировать пакеты, позволяя объединять сети, в которых максимальная длина пакета различна.

Благодаря программному обеспечению TCP/IP все компьютеры, подключенные к вычислительной сети, становятся "близкими родственниками". По существу оно скрывает маршрутизаторы и базовую архитектуру сетей и делает так, что все это выглядит как одна большая сеть. Точно так же, как подключения к сети Ethernet распознаются по 48-разрядным идентификаторам Ethernet, подключения к интрасети идентифицируются 32-разрядными IP-адресами, которые мы выражаем в форме десятичных чисел, разделенных точками (например, 128.10.2.3). Взяв IP-адрес удаленного компьютера, компьютер в интрасети или в Internet может отправить данные на него, как будто они составляют часть одной и той же физической сети.

TCP/IP дает решение проблемы данными между двумя компьютерами, подключенными к одной и той же интрасети, но принадлежащими различным физическим сетям. Решение состоит из нескольких частей, причем каждый член семейства протоколов TCP/IP вносит свою лепту в общее дело. IP - самый фундаментальный протокол из комплекта TCP/IP - передает IP-дейтаграммы по интрасети и выполняет важную функцию, называемую маршрутизацией, по сути дела это выбор маршрута, по которому дейтаграмма будет следовать из пункта А в пункт В, и использование маршрутизаторов для "прыжков" между сетями. В стеке TCP/IP экономно используются возможности широковещательных рассылок.

Таблица 1.1 Стеки протоколов

<table>
<tr><th>Модель OSI</th><th>TCP/IP</th><th>Стек OSI</th></tr>
<tr><td>Прикладной</td><td rowspan="3">Telnet, FTP, POP3, SMTP, IMAP4, ICQ, SNMP, WWW</td><td>X.400, X.500, FTAM</td></tr>
<tr><td>Представительный</td><td>Представительный протокол OSI</td></tr>
<tr><td>Сеансовый</td><td>Сеансовый протокол OSI</td></tr>
<tr><td>Транспортный</td><td>TCP, UDP, SCTP</td><td>Транспортный протокол OSI</td></tr>
<tr><td>Сетевой</td><td>IP, RIP, OSPF, ICMP</td><td>ES-ES, IS-IS</td></tr>
<tr><td>Канальный</td><td colspan="2">802.3 (Ethernet), 802.5 (Token Ring), Fast Ethernet, X.25</td></tr>
<tr><td>Физический</td><td colspan="2">Витая пара, оптоволокно, радиоволны</td></tr>
</table>

Однако, как и всегда, за получаемые преимущества надо платить, и платой здесь оказываются высокие требования к ресурсам и сложность администрирования IP-сетей. Мощные функциональные возможности протоколов стека TCP/IP требуют для своей реализации высоких вычислительных затрат. Гибкая система адресации и отказ от широковещательных рассылок приводят к наличию в IP-сети различных централизованных служб типа DNS, DHCP и т.п.

Контрольные вопросы

1. Дайте определение понятию «интерфейс».
2. Дайте определение понятию «протокол».
3. В чем заключается процедура декомпозиции?
4. На сколько уровней делятся средства взаимодействия модели OSI? Назовите эти уровни?
5. На сколько уровней делятся средства взаимодействия TCP/IP?
6. В чем отличия назначений сетевого и канального уровня?
7. Какие уровни являются сетезависимыми и сетенезависимыми модели OSI?
8. Какие основные назначения транспортного уровня?
9. Приведите примеры протоколов прикладного уровня?
10. К какому уровню относятся протоколы IP и TCP?

ГЛАВА 2. СЕТЕВОЙ УРОВЕНЬ

2.1. IP v.4 адресация. Маски

IP-адресация построена на концепции сети, состоящей из хостов и других сетей [2-4]. Хост представляет собой объект сети, который может передавать и принимать IP-пакеты, например, компьютер или маршрутизатор. Хосты соединены друг с другом через одну или несколько сетей. IP-адрес состоит из двух логических частей – адреса сети и адреса хоста (более точно – адреса интерфейса хоста) в сети и имеет длину 4 байта (IPv4) и обычно записывается в виде четырех чисел, представляющих значения каждого байта в десятичной форме и разделенных точками, например, 62.76.78.110. IP-адреса назначаются администратором при конфигурировании компьютеров и маршрутизаторов. Номер сети может быть выбран администратором произвольно (внутренние закрытые сети), либо выделен поставщиком услуг Internet или назначен подразделением организации InterNIC (Internet Network Information Center), если сеть должна работать как составная часть глобальной сети Internet. Обычно поставщики услуг Internet получают диапазоны адресов у подразделений InterNIC, а затем распределяют их между своими абонентами. Номер узла в протоколе IP назначается независимо от аппаратного адреса интерфейса узла. Маршрутизатор по определению входит сразу в несколько сетей, поэтому каждый порт маршрутизатора имеет свой собственный IP-адрес. Компьютер также может входить в несколько IP-сетей. В этом случае компьютер должен иметь несколько IP-адресов, по числу сетевых связей. Таким образом, IP-адрес характеризует не отдельный компьютер или маршрутизатор, а одно сетевое соединение.

Классы IP-адресов

Какая часть адреса относится к адресу сети, а какая – к адресу хоста, определяется значениями первых бит адреса, которые определяются *классом* IP-сетей.

Существуют 5 классов IP-сетей (рис. 2.1):

- **Класс A**. 0… Сети класса A предназначены главным образом для использования крупными организациями, т.к. они обеспечивают всего 8 бит для поля адреса в сети и 24 бита для адресации хостов.
- **Класс B**. 10… Сети класса B выделяют 16 бит для адресации сети и 16 бит для адресации хостов. Данный класс обеспечивает хороший компромисс между адресным пространством сети и адресным пространством хостов. Используется в сетях среднего размера.
- **Класс C**. 110… Сети класса C выделяют 24 бита для адресации сети. Однако сети класса C обеспечивают только 8 бит для поля адреса хоста, поэтому сети данного класса используются небольшими организациями.
- **Класс D**. 1110… Адреса класса D резервируются для групповой адресации в соответствии с официальным документом RFC-1112.

Назначением данных адресов является распространение информации по схеме «один ко многим».

- **Класс E**. 11110… Адреса класса E зарезервированы для использования в будущем.

Класс	1 байт								2 байт		3 байт		4 байт	
A	0	Сеть							Хост					
B	1	0	Сеть								Хост			
C	1	1	0	Сеть									Хост	
D	1	1	1	0	Групповой адрес									
E	1	1	1	1	0	Зарезервировано								

Рис. 2.1 Классы IP-сетей

Ниже приведена таблица 2.1 диапазонов адресов сетей и их назначение.

Таблица 2.1 Адресное пространство классов IP-сетей

Класс	Наименьший номер сети	Наибольший номер сети	Маска	
A	1.0.0.0	126.0.0.0	255.0.0.0	До 2^{24} узлов
	10.0.0.0	10.0.0.0	255.0.0.0	Внутренние закрытые сети
B	128.0.0.0	191.255.0.0	255.255.0.0	До 2^{16} узлов
	172.16.0.0	172.31.0.0	255.255.0.0	Внутренние закрытые сети
C	192.0.0.0	223.255.255.0	255.255.255.0	До 2^{8} узлов
	192.168.0.0	192.168.255.0	255.255.255.0	Внутренние закрытые сети
D	224.0.0.0	239.255.255.255		Групповой адрес
	224.0.0.1	224.0.0.1		Группа хостов в данной сети
E	240.0.0.0	247.255.255.255		Зарезервирован

Особые значения IP-адресов:

- (все нули) – обозначает адрес данного хоста. В таблице маршрутизации указывает шлюз по умолчанию;
- (все нули).(адрес хоста) – обозначает хост в данной (локальной) сети;
- (адрес сети).(все нули) – обозначает данную IP-сеть;
- (все единицы) – обозначает все хосты в данной локальной сети (подсети). Пакет с указанным адресом должен рассылаться всем хостам в той же сети, что и источник. Такая рассылка называется *ограниченным широковещательным сообщением (limited broadcast)*;
- (номер сети).(все единицы) – обозначает все узлы в указанной IP-сети. Пакет с указанным адресом рассылается всем узлам с указанным адресом сети. Такая рассылка называется *широковещательным сообщением (broadcast)*;

- (все единицы).(все нули) – применяется для обозначения маски сети. *Маска* – это число, которое используется в паре с IP-адресом и в битах, которые обозначают адрес сети, содержит единицы, а в битах номера хоста – нули;
- 127.(адрес) – используется для тестирования программ и взаимодействия процессов в пределах одной машины, создавая петли. Данные не передаются по сети, а возвращаются модулям верхнего уровня, как только что принятые. Эти адреса называются *петлями (loopback).*

Использование масок в IP-адресации

Традиционная схема деления IP-адреса на номер сети и номер узла основана на понятии класса, который определяется значениями нескольких первых бит адреса. Именно потому, что первый байт адреса 185.23.44.206 попадает в диапазон 128-191, мы можем сказать, что этот адрес относится к классу B, а значит, номером сети являются первые два байта, дополненные двумя нулевыми байтами — 185.23.0.0, а номером узла — 0.0.44.206.

А что если использовать какой-либо другой признак, с помощью которого можно было бы более гибко устанавливать границу между номером сети и номером узла? В качестве такого признака сейчас получили широкое распространение маски. Маска — это число, которое используется в паре с IP-адресом; двоичная запись маски содержит единицы в тех разрядах, которые должны в IP-адресе интерпретироваться как номер сети. Поскольку номер сети является цельной частью адреса, единицы в маске также должны представлять непрерывную последовательность.

Для стандартных классов сетей маски имеют следующие значения:

- класс A - 11111111. 00000000. 00000000. 00000000 (255.0.0.0);
- класс B - 11111111. 11111111. 00000000. 00000000 (255.255.0.0);
- класс C - 11111111.11111111.11111111. 00000000 (255.255.255.0).

Замечание. Для записи масок используются и другие форматы, например, удобно интерпретировать значение маски, записанной в шестнадцатеричном коде: FF.FF.00.00 – маска для адресов класса B. Часто встречается и такое обозначение 185.23.44.206/16 - эта запись говорит о том, что маска для этого адреса содержит 16 единиц или что в указанном IP-адресе под номер сети отведено 16 двоичных разрядов.

Снабжая каждый IP-адрес маской, можно отказаться от понятий классов адресов и сделать более гибкой систему адресации. Например, если рассмотренный выше адрес 185.23.44.206 ассоциировать с маской 255.255.255.0, то номером сети будет 185.23.44.0, а не 185.23.0.0, как это определено системой классов.

В масках количество единиц в последовательности, определяющей границу номера сети, не обязательно должно быть кратным 8, чтобы повторять деление адреса на байты. Пусть, например, для IP-адреса 138.64.134.5 указана маска 255.255.128.0, то есть в двоичном виде:

IP-адрес 138.64.134.5 – 10000001. 01000000.10000110. 00000101

Маска 255.255.128.0 – 11111111.11111111.10000000.00000000
Если игнорировать маску, то в соответствии с системой классов адрес 138.64.134.5 относится к классу B, а значит, номером сети являются первые 2 байта — 138.64.0.0, а номером узла — 0.0.134.5.

Если же использовать для определения границы номера сети маску, то 17 последовательных единиц в маске, «наложенные» на IP-адрес, определяют в качестве номера сети в двоичном выражении число: 10000001. 01000000. 10000000. 00000000 или в десятичной форме записи — номер сети 138.64.128.0, а номер узла 0.0.6.5.

Механизм масок широко распространен в IP-маршрутизации, причем маски могут использоваться для самых разных целей. С их помощью администратор может структурировать свою сеть, не требуя от поставщика услуг дополнительных номеров сетей. На основе этого же механизма поставщики услуг могут объединять адресные пространства нескольких сетей путем введения так называемых «префиксов» с целью уменьшения объема таблиц маршрутизации и повышения за счет этого производительности маршрутизаторов.

2.2. IP протокол v.4. Заголовок пакета

Основу транспортных средств стека протоколов TCP/IP составляет *протокол межсетевого взаимодействия (Internet Protocol, IP)*, который обеспечивает передачу дейтаграмм от отправителя к получателям через объединенную систему компьютерных сетей [1 – 3]. Назначением данного протокола является *передача пакетов данных между сетями*.

Протокол IP относится к протоколам без установления соединений. Перед ним не ставится задача надежной доставки сообщений от отправителя к получателю. Протокол IP обрабатывает каждый IP-пакет как независимую единицу, не имеющую связи ни с какими другими пакетами. Все вопросы обеспечения надежности доставки данных по сетям в стеке TCP/IP решает протокол TCP, работающий непосредственно над протоколом IP. Данный протокол организует повторную передачу пакетов в случае недоставки пакета данных по назначению.

Структура IP-пакета

IP-пакет состоит из заголовка и поля данных (рис. 2.2).

<table>
<tr><td>offset</td><td colspan="8">0 34 78 1516 31</td></tr>
<tr><td>0</td><td colspan="2">Version</td><td>IHL</td><td colspan="5">TOS
PR D T R</td><td colspan="3">Total Length</td></tr>
<tr><td>4</td><td colspan="5">Identification</td><td colspan="2">Flags
D M</td><td colspan="2">Fragment Offset</td></tr>
<tr><td>8</td><td colspan="3">TTL</td><td colspan="3">Protocol</td><td colspan="3">Header Checksum</td></tr>
<tr><td>12</td><td colspan="8">Source Address</td></tr>
<tr><td>16</td><td colspan="8">Destination Address</td></tr>
<tr><td>20</td><td colspan="6">Options</td><td colspan="2">Padding</td></tr>
</table>

Рис. 2.2 Формат заголовка IP-пакета

Номер версии (Version) – занимает 4 бита и указывает версию протокола IP. В настоящий момент используется версия 4 (IPv4), и готовится переход на версию 6 (IPv6).

Длина заголовка (IHL – Internet Header Length) – занимает 4 бита и указывает значение длины заголовка, измеряемое в 32-битовых словах (4 байта). Значение данного поля не должно быть меньше 5, т.е. *минимальная* длина заголовка 20 байт. При увеличении объема служебной информации эта длина может быть увеличена за счет использования дополнительных байт поля *Опции (Options).* Наибольший заголовок занимает 60 байт.

Тип сервиса (TOS – Type Of Services) – занимает 1 байт и используется для идентификации используемого дейтаграммой сервиса, определяя вид ее обработки. Значения данного поля задает приоритетность пакета и вид критерия выбора маршрута. Поле используется преимущественно шлюзами, для выбора параметров передачи пакета по данной сети до сети следующего узла или шлюза маршрутизации. Первые 3 бита этого поля образуют подполе *приоритета (Precedence)* пакета. Приоритет может иметь значения от самого низкого – 0 (нормальный пакет) до самого высокого – 7 (пакет управляющей информации). Сначала обрабатываются пакеты с наивысшим приоритетом. Следующие 3 бита критерий выбора маршрута. Реально выбор осуществляется между тремя альтернативами: малой задержкой, высокой достоверностью и высокой пропускной способностью. Установленный бит D (*Delay*) говорит о необходимости выбора маршрута для минимизации задержки доставки данного пакета, бит T (*Throughput*) – для максимизации пропускной способности, а бит R (*Reliability*) – для максимизации надежности доставки. Зарезервированные биты имеют нулевое значение.

Общая длина (Total Length) – занимает 2 байта и означает общую длину пакета с учетом заголовка и поля данных. Максимальная длина пакета ограничена разрядностью поля, определяющего эту величину, и составляет 65535 байт. При передаче по сетям различного типа длина пакета выбирается с учетом максимальной длины пакета протокола нижнего уровня, несущего IP-пакеты. В стандарте предусматривается, что все хосты должны работать с длинами пакета до 576 байт, превышать которую не рекомендуется.

Идентификатор пакета (Identification) – занимает 2 байта и используется для распознавания пакетов, образовавшихся путем фрагментации исходного пакета. Все фрагменты должны иметь одинаковое значение этого поля. Модули обслуживания протокола группируют фрагменты с одинаковым адресом источника, адресом назначения, типом протокола и идентификатором.

Флаги (Flags) – занимают 3 бита и содержат признаки, связанные с фрагментацией. Установленный бит DF (*Do not Fragment*) запрещает маршрутизатору фрагментировать данный пакет, а установленный бит MF (*More Fragments*) говорит о том, что данный пакет является промежуточным (не последним) фрагментом. Оставшийся бит зарезервирован.

Смещение фрагмента (Fragment Offset) – занимает 13 бит и задает смещение в 64-битных словах (8 байт) поля данных этого пакета от начала общего поля данных исходного пакета, подвергнутого фрагментации. Используется при

сборке/разборке фрагментов пакетов при передачах их между сетями с различными величинами MTU.

Время жизни (Time To Live) – занимает 1 байт и означает предельный срок, в течение которого пакет может перемещаться по сети. Время жизни данного пакета измеряется в секундах и задается источником передачи. На маршрутизаторах и в других узлах сети по истечении каждой секунды из текущего времени вычитается единица, даже в том случае, когда время задержки меньше секунды. Если параметр времени жизни станет нулевым до того, как пакет достигнет получателя, этот пакет будет уничтожен. Значение данного поля изменяется при обработке заголовка IP-пакета.

Протокол верхнего уровня (Protocol) – занимает 1 байт и указывает, какому протоколу верхнего уровня принадлежит информация, размещенная в поле данных пакета.

Контрольная сумма заголовка (Header Checksum) – занимает 2 байта и рассчитывается только по заголовку. Поскольку некоторые поля заголовка меняют свое значение (например, TTL), контрольная сумма проверяется и повторно вычисляется при каждой обработке IP-заголовка. Контрольная сумма (16 бит) вычисляется как дополнение к сумме всех 16-битовых слов заголовка. При вычислении контрольной суммы значение самого поля устанавливается в нуль. При обнаружении ошибки контрольной суммы пакет будет отброшен.

Адрес источника (Source Address) – занимает 32 бита и содержит IP-адрес отправителя пакета.

Адрес назначения (Destination Address) – занимает 32 бита и содержит IP-адрес получателя пакета.

Опции (Options) – является не обязательным и используется обычно только при отладке сети. Это поле состоит из нескольких подполей, каждое из которых может быть одного из восьми предопределенных типов. В этих подполях можно указывать точный маршрут прохождения пакетов через маршрутизаторы (структуры *LSRR – Loose Source and Record Route* и *SSRR – Strict Source and Record Route*), регистрировать проходимые пакетом маршрутизаторы (структура *RR – Record Route*), помещать данные системы безопасности, а также временные метки прохождения шлюзов маршрутизации.

Выравнивание (Padding) – имеет переменную длину и используется для выравнивания заголовка пакета по 32-битной границе. Выравнивание осуществляется путем заполнения данного поля нулями.

Функции протокола IP

Основным назначением протокола IP является передача датаграмм через связанные между собой множества сетей. Это достигается передачей датаграмм от одного модуля IP-сервиса к другому до тех пор, пока не будет достигнут хост получателя. Дейтаграммы передаются через интерфейсы сегментов сетей, а выбор пути осуществляется на основе IP-адреса получателя пакета.

Фрагментация

Перед тем как *датаграмма* (IP заголовок + Данные) отправится по сети к участку следующего попадания, она инкапсулируется внутри заголовка (заголовков) второго (канального) уровня [1].

Например, для прохождения в сети 802.3 или 802.5 добавляются: MAC заголовок, заголовок LLC, подзаголовок SNAP (далее идет заголовок IP с данными) и завершающая часть MAC.

Максимальная длина датаграммы для конкретного носителя вычисляется как разность максимального размера кадра, длины заголовка, длины завершающей части кадра и размера заголовка второго уровня.

Максимально возможная длина датаграммы в заданном носителе называется максимальным элементом пересылки (Maximum Transmission Unit - *MTU*).

При передаче сообщения от одного IP-модуля к другому может случиться так, что датаграммы будут передаваться по сети, для которой допустимый размер пакета данных меньше размера датаграммы. Тогда начинает работать механизм *фрагментации* датаграмм.

Фрагментация IP-датаграмм необходима, когда ее размер превышает размер максимально допустимого пакета данных при передаче по сегменту сети.

Однако не все датаграммы могут быть фрагментированы. Если датаграмма помечена как «не фрагментируемая» (т.е. установлен флаг DF), то такая датаграмма ни при каких обстоятельствах не может подвергаться разбиению на пакеты меньшей длины. Если такая датаграмма не может быть доставлена в точку назначения без фрагментации, она будет уничтожена. В остальных случаях датаграмму можно разбить на более мелкие фрагменты.

Процедура фрагментации может разбить датаграмму на пакеты произвольной длины, а затем восстановить ее в первоначальном виде. Каждый фрагмент исходной датаграммы имеет уникальный идентификатор (Identification), который однозначно определяет его принадлежность к исходному фрагментируемому пакету. Это число уникально для пары адресов отправитель-получатель все то время, пока датаграмма находится в сети. Поля смещения фрагмента (Fragment Offset) и длины, заданные для каждого фрагмента исходной датаграммы, полностью определяют положение фрагмента в исходной датаграмме. Флаг «следующего фрагмента» MF (More Fragments) указывает на то, что у данного фрагмента есть еще продолжение или что данный пакет последний. Данной информации достаточно, чтобы IP-модуль, получающий информацию фрагментами, смог собрать исходную датаграмму.

2.3. ICMP протокол

Протокол IP имеет ясную и элегантную структуру. В нормальных ситуациях IP очень эффективно использует для пересылки память и ресурсы. Однако что произойдет в нестандартной ситуации? Что может прервать бесцельное блуждание датаграммы до завершения ее времени жизни после краха маршрутизатора и неисправности в сети? Кто предупредит приложение о

прекращении отправки датаграмм в недостижимую точку назначения?

Средства для лечения таких неисправностей предоставляет *протокол управляющих сообщений Интернета* (Internet Control Message Protocol — ICMP) [1]. Он выполняет роль сетевого помощника, способствуя маршрутизации в хостах и обеспечивая сетевого администратора средствами определения состояния сетевых узлов. Функции ICMP являются важной частью IP. Все хосты и маршрутизаторы должны быть способны генерировать и обрабатывать сообщения ICMP. При правильном использовании эти сообщения могут улучшить выполнение сетевых операций.

Сообщения ICMP пересылаются в датаграммах IP с обычным заголовком IP (см. рис. 2.3), имея в поле *протокола* значение 1.

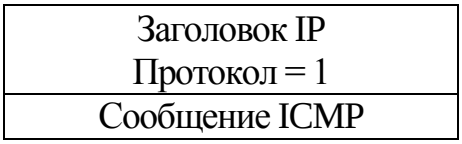


Рис. 2.3 Пакетирование сообщения ICMP

ICMP определен в RFC 792. RFC 1122 (требования к хостам) и RFC 1812 (требования к маршрутизаторам) содержат несколько очень полезных разъяснений. Исследованию маршрутов посвящен RFC 1256. Исследование MTU по пути рассмотрено в RFC 1191, а дополнительные рекомендации представлены в RFC 1435.

2.3.1. Сообщения об ошибках ICMP

Бывают ситуации, приводящие к отбрасыванию (удалению из сети) датаграммы IP. Например, точка назначения может стать недоступной из-за обрыва связи. Или может завершиться время жизни датаграммы. Маршрутизатор не сможет переслать длинную датаграмму при запрещении фрагментации.

При отбрасывании датаграммы по адресу ее источника направляется сообщение ICMP, указывающее на возникшую проблему [1].

ICMP быстро сообщит системе о выявленной проблеме. Это очень надежный протокол, поскольку указание на ошибки не зависит от наличия сетевого центра управления.

Однако в использовании сообщений ICMP имеются некоторые недостатки. Например, если недостижима точка назначения, то сообщение будет распространяться до источника по всей сети, а не на станцию сетевого управления.

Реально ICMP не имеет средств предоставить отчет об ошибках выделенному операционному центру. Для этого служит протокол SNMP.

Типы сообщений об ошибках

В таблице 2.2 перечислены формальные имена сообщений об ошибках ICMP.

Таблица 2.2 Сообщения об ошибках

Сообщение	Описание
Destination Unreachable (недостижимая точка назначения)	Датаграмма не может достичь хоста назначения, утилиты или приложения
Time Exceeded (время закончилось)	Маршрутизатор определил завершение времени жизни, или закончилось время на сборку фрагментов в хосте назначения
Parameter Problem (проблема с параметром)	В заголовке IP неверный параметр
Source Quench (подавление источника)	Перегружен маршрутизатор или система назначения (системам рекомендуется *не отправлять* это сообщение)
Redirect (перенаправление)	Хост направил датаграмму на неверный локальный маршрутизатор

Обязанность по отправке сообщения ICMP

Протокол ICMP определяет, что сообщения *могут* или *должны* быть посланы в каждом случае, но он не требует выдавать сообщения ICMP о *каждой* ошибке.

В этом есть здравый смысл. Основным назначением маршрутизатора в сети является пересылка датаграмм. Перегруженный хост назначения должен уделять больше времени доставке датаграмм в приложения, а не указанию на ошибки удаленному хосту. Именно поэтому не формируются сообщения о случайном отбрасывании датаграммы.

Когда <u>не нужно</u> посылать сообщение ICMP

Напомним, что ICMP-сообщение об ошибке посылается, когда в сети не все благополучно. Важно, обеспечить, чтобы трафик ICMP не перегружал сети, делая ситуацию еще хуже. Для этого протокола, требуется ввести несколько очевидных ограничений. ICMP не должен формировать сообщения о:

- маршрутизации и доставке ICMP-сообщений messages;
- широковещательных и многоадресных датаграммах;
- фрагментах датаграмм, кроме первых;
- сообщениях, чей адрес источника не идентифицирует уникальный хост (например, IP-адреса источников 127.0.0.1 или 0.0.0.0).

Формат сообщения ICMP

Сообщение ICMP переносится в части данных датаграммы IP. Каждое сообщение ICMP начинается тремя одинаковыми полями: полем *типа* (Type),

полем *кода* (Code), обеспечивающим более подробное описание ошибки, и полем *контрольной суммы* (Checksum). Формат оставшейся части сообщения определяется типом сообщения.

Сообщение об ошибке ICMP обрамляется заголовком IP. Добавляются первые 8 октетов датаграммы, которая привела к ошибке. Эти сведения позволяют проанализировать причину ошибки, поскольку содержат информацию о предполагаемом назначении датаграммы и целевом протоколе четвертого уровня. Дополнительные 8 байт позволяют определить коммуникационный элемент приложения (подробнее в [1]).

В сообщение включается и контрольная сумма ICMP, начиная от поля *Type.*

Сообщение Destination Unreachable

Существует много причин прекращения доставки датаграммы. Разорванная связь физически не позволит маршрутизатору достичь подсети назначения или выполнить пересылку в точку следующего попадания. Хост назначения может стать недоступным при отключении его для проведения профилактики.

Современные маршрутизаторы имеют хорошие средства обеспечения безопасности. Они могут быть сконфигурированы для просмотра входящего в сеть трафика. При запрещении сетевым администратором доступа к точке назначения датаграмма также не может быть доставлена.

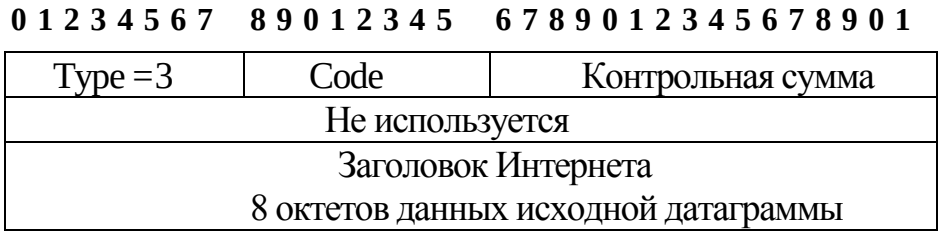


Рис. 2.3 Формат ICMP-сообщения Destination Unreachable

Формат сообщения *Destination Unreachable* показан на рисунке 2.3. Поле Type (в нашем случае 3) идентифицирует именно этот *тип* сообщения. Поле *Code* отражает причину отправки сообщения. Полный список кодов этого поля представлен в таблице 2.3.

Таблица 2.3 Коды ошибок сообщения Destination Unreachable

Код	**Смысл**
0	Сеть недостижима
1	Хост недостижим
2	Запрашиваемый протокол не поддерживается в точке назначения
3	Порт недостижим (недоступно удаленное приложение)
4	Необходима фрагментация, но установлен флаг "Не фрагментировать"
5	Неверен маршрут от источника

6	Неизвестна сеть назначения
7	Неизвестен хост назначения
8	Хост источника изолирован
9	Административно запрещены коммуникации с сетью назначения
10	Административно запрещены коммуникации с хостом назначения
11	Сеть недостижима для заданного типа обслуживания
12	Хост недостижим для заданного типа обслуживания

Сообщение Time Exceeded

Пересылаемая датаграмма может быть отброшена по тайм-ауту при уменьшении до нуля ее времени жизни (TTL). Еще один тайм-аут может возникнуть в хосте назначения, когда завершится время, выделенное на сборку, а прибыли еще не все фрагменты датаграммы. В обоих случаях формируется сообщение *Time Exceeded* для источника датаграммы. Формат этого сообщения показан на рисунке 2.4.

0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1

Type =11	Code	Контрольная сумма
Не используется		
Заголовок Интернета 8 октетов данных исходной датаграммы		

Рис. 2.4 Формат **с**ообщение Time Exceeded

Значения кодов (см. таблицу 2.4) отражают причину тайм-аута.

Таблица 2. 4 Коды сообщения Time Exceeded

Код	**Смысл**
0	Завершилось время жизни датаграммы
1	Завершилось время на сборку фрагментов датаграммы

Сообщение Parameter Problem

ICMP-сообщение *Parameter Problem* используется для отчета об ошибках, не специфицированных в кодах других сообщений. Например, в полях вариантов может появиться неверная информация, не позволяющая правильно обработать датаграмму, в результате чего датаграмма будет отброшена. Более часто проблемы с параметрами возникают из-за ошибок в реализации, когда система пытается записать параметры в заголовок IP.

0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1

Type = 12	Code	Контрольная сумма
Указатель	Не используется	
Заголовок Интернета 8 октетов данных исходной датаграммы		

Рис. 2.5 Формат ICMP-сообщения Parameter Problem

Поле *Pointer* (указатель) сообщения *Parameter Problem* идентифицирует октет, в котором выявлена ошибка. На рисунке 2.5 показан формат сообщения *Parameter Problem,* а в таблице 2.5 — значения кодов ошибок.

Таблица 2.5 Коды сообщения Parameter Problem

Код	**Смысл**
0	Значение в поле указателя специфицирует ошибочный октет
1	Отсутствует требуемый вариант (используется военными для указания на отсутствие параметров безопасности)
2	Неверная длина

Проблемы перегрузок

Протокол IP очень прост: хост или маршрутизатор обрабатывают датаграмму и посылают ее как можно быстрее. Однако доставка не всегда проходит гладко. Могут возникнуть различные проблемы.

Когда один или несколько хостов отправляют трафик UDP на медленный сервер, то на последнем может возникнуть перегрузка, что приведет к отбрасыванию сервером некоторой части этого трафика.

Маршрутизатор может переполнить свои буферы и далее будет вынужден отбрасывать некоторые поступающие датаграммы. Медленное соединение через региональную сеть (например, на скорости 56 Кбит/с) между двумя скоростными локальными сетями (например, в 100 Мбит/с) может создать затор на пути следования датаграмм. Из-за этого в сети возникнут перегрузки, которые также приведут к отбрасыванию датаграммы и, следовательно, к созданию еще большего трафика.

Сообщение Source Quench

Сообщение *Source Quench* (подавление источника) показано на рисунке 2.6. Оно позволяет попытаться решить проблему перегрузок, хотя и не всегда успешно. Механизмы для подавления источника перегрузки сети должны создавать разработчики конкретных продуктов, но остается открытым конкретный вопрос:

0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1

Type =4	Code	Контрольная сумма
Не используется		
Заголовок Интернета 8 октетов данных исходной датаграммы		

Рис. 2.6 Формат ICMP-сообщения Source Quench

Когда и кому маршрутизатор или хост должен отправлять сообщение Source Quench?

Обычно ICMP-сообщение указывает хосту источника на причину отбрасывания посланной им датаграммы. Однако при перегрузке такое сообщение может не дойти до этого хоста, генерирующего очень напряженный сетевой трафик. Кроме того, очень расплывчаты требования к обработке поступающих сообщений *Source Quench.*

Текущий документ по *требованиям к хостам* (RFC 1812) оговаривает в качестве особого пункта, что сообщения *Source Quench* вовсе *не нужно* посылать. Работа должна выполняться более совершенным механизмом управления нагрузкой в сети.

Сообщения Redirect

К локальной сети может быть подключено более одного маршрутизатора. Когда локальный хост посылает датаграмму не на тот маршрутизатор, последний пересылает ее и отправляет хосту источника ICMP-сообщение *Redirect* (перенаправление). Хост должен переключить последующий трафик на более короткий путь.

0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1

Type =5	Code	Контрольная сумма
Не используется		
Заголовок Интернета 8 октетов данных исходной датаграммы		

Рис. 2.7 Формат ICMP-сообщения Redirect

Сообщение *Redirect* используется и для выключения маршрутизатора системным администратором. Хост может быть сконфигурирован с единственным маршрутизатором по умолчанию; при этом он будет динамически определять возможности пересылки через другие маршрутизаторы.

Формат сообщения о перенаправлении показан на рисунке 2.7. Коды этого сообщения перечислены в таблице 2.6. Некоторые протоколы маршрутизации способны выбирать путь доставки на основе содержимого поля *типа обслуживания* (TOS) датаграммы. Коды 2 и 3 предоставляют некоторые сведения для такого выбора.

Таблица 2.6 Коды перенаправления

Код	**Смысл**
0	Перенаправление датаграммы в сеть
1	Перенаправление датаграммы в хост
2	Перенаправление датаграммы в сеть на основе значения из поля типа обслуживания
3	Перенаправление датаграммы в хост на основе значения из поля типа обслуживания

Управление поступающими сообщениями ICMP

Что должен делать хост, получивший сообщение ICMP? Реализации различных разработчиков по-разному отвечают на этот вопрос. В некоторых из них хосты игнорируют все или многие такие сообщения. Стандарты TCP/IP оставляют большую свободу выбора в решении этого вопроса. Для различных типов сообщений ICMP предлагаются следующие рекомендации:

Destination Unreachable	Доставить ICMP-сообщение на транспортный уровень. Выполняемые действия должны зависеть от того, является ли причина вывода сообщения временной или постоянной (например, административный запрет на пересылку).
Redirect	Хост *обязан* обновить таблицу маршрутизации.
Source Quench	Доставить ICMP-сообщение на транспортный уровень или в модуль обработки ICMP.
Time Exceeded	Доставить на транспортный уровень.
Parameter Problem	Доставить ICMP-сообщение на транспортный уровень с необязательным уведомлением пользователя.

Иногда ошибки должны обрабатываться совместно операционной системой, коммуникационным программным обеспечением и сетевым приложением.

2.3.2. Исследование MTU по пути

При пересылке большого объема данных (например, при копировании файлов по сети) с одного хоста на другой размер датаграмм существенно влияет на производительность. Заголовки IP и TCP требуют не менее 40 дополнительных байт [1].

- Если данные пересылаются в 80-байтовых датаграммах, дополнительная нагрузка составит 50%.
- Если данные пересылаются в 400-байтовых датаграммах, дополнительная нагрузка составит 10%.
- Если данные пересылаются в 4000-байтовых датаграммах, дополнительная нагрузка составит 1%.

Для минимизации дополнительной нагрузки лучше отсылать датаграммы наибольшего размера. Однако этот размер ограничивается значением максимального элемента пересылки (Maximum Transmission Unit — MTU) для каждого из носителей. Если датаграмма будет слишком большой, то она будет фрагментирована, а этот процесс снижает производительность. С точки зрения пользователя, качество сети определяется двумя параметрами: интервалом пересылки (от начала пересылки до ее завершения) и временем ожидания (задержкой доступа к сети, занятой другими пользователями). Увеличение размера датаграммы приводит к снижению интервала пересылки, но увеличению ожидания для других пользователей. Грубо говоря, нагрузка на сеть будет выглядеть как пиковые импульсы с очень небольшой нагрузкой

между ними, что считается самым неудачным вариантом загрузки сети. Гораздо лучше, когда сеть нагружается равномерно (*Прим. пер.).*

Многие годы хосты избегали фрагментации, устанавливая эффективное значение MTU для пересылки в 576 октетов для всех нелокальных хостов. Это часто приводило к ненужному снижению производительности.

Гораздо полезнее заранее знать наибольший допустимый размер датаграммы, которую можно переслать по заданному пути. Существует очень простой механизм *исследования MTU по пути* (Path MTU discovery), позволяющий узнать это значение. Для такого исследования:

- Флаг *"Не фрагментировать"* заголовка IP устанавливают в 1.
- Размер MTU по пути первоначально устанавливают в значение MTU для локального интерфейса.
- Если датаграмма будет слишком велика для одного из маршрутизаторов, то он пошлет обратно ICMP-сообщение *Destination Unreachable* с кодом 4.
- Хост источника уменьшит размер датаграммы и повторит попытку.

Какое же значение нужно выбрать для следующей попытки? Спецификация IP предполагает сохранение значения MTU и его доступность для протоколов транспортного уровня. Если маршрутизатор имеет современное программное обеспечение, то он будет включать в пересылаемое дальше по сети сообщение *Destination Unreachable* размер MTU (см. рис. 2.8). Иногда средства защиты конфигурируются на полное исключение *всех* входящих сообщений ICMP, что не позволяет использовать механизм определения MTU по пути следования датаграммы.

0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1

Type = 3	Code	Контрольная сумма
Не используется		MTU для следующего попадания
Заголовок Интернета + 8 октетов данных исходной датаграммы		

Рис. 2.8 Сообщение Destination Unreachable приносит результат исследования размера MTU

Поскольку пути пересылки могут меняться динамически, флажок *"Не фрагментировать"* нужно устанавливать во всех коммуникационных датаграммах. При необходимости маршрутизатор будут посылать сведения об обновлениях.

Если маршрутизатор использует устаревшее программное обеспечение, он не сможет предоставить значение MTU для следующего попадания. В этом случае значение для следующей попытки будет выбираться из списка стандартных размеров MTU с постепенным уменьшением для каждой новой попытки до достижения значения, нужного для коммуникации с удаленным хостом.

Разумеется, изменение пути следования может создать предпосылки для использования большего размера MTU. В этом случае система, согласовавшая небольшой размер MTU, будет пытаться его увеличить, если такое улучшение будет возможно.

2.3.3. Сообщения запросов ICMP

Не все сообщения ICMP сигнализируют об ошибках. Некоторые из них извлекают из сети полезные сведения. Работает ли хост X? Не выключен ли хост Y? Как долго движется датаграмма до хоста Z и обратно? Какова маска подсети хоста источника?

Ответы на эти вопросы дают следующие сообщения ICMP [1]:

- *Эхо-запросы* и *эхо-ответы* обеспечивают обмен информацией между хостами и маршрутизаторами.
- Запросы и ответы о *маске адреса* позволяют системе исследовать присвоенную интерфейсу маску адреса.
- Запросы и ответы *временной метки* служат для извлечения сведений об установке времени на целевой системе. Ответы на такие запросы дают информацию, необходимую для оценки времени обработки датаграмм на хосте.

Эхо-запросы и эхо-ответы

Эхо-запросы (Echo Request) и *эхо-ответы* (Echo Reply) применяются для проверки активности системы. Код типа 8 применяется в запросах, а код 0 — в ответах. Количество октетов в поле данных переменно и может выбираться отправителем.

Отвечающая сторона должна послать обратно те же самые данные, которые были получены. Поле *идентификатора* служит для сравнения ответа с исходным запросом. Последовательный номер эхо-сообщения может применяться для тестирования, на каком участке произошел обрыв сети, и для вычисления приблизительного времени на путь туда и обратно. При этом идентификатор не меняется, а последовательный номер (начиная от 0) увеличивается на единицу для каждого сообщения. Формат эхо-сообщения показан на рисунке 2.9.

<table>
<tr><th colspan="3">0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1</th></tr>
<tr><td>Type = 8 или 0</td><td>Code</td><td>Контрольная сумма</td></tr>
<tr><td colspan="2">Идентификатор</td><td>Последовательный номер</td></tr>
<tr><td colspan="3">Данные</td></tr>
</table>

Рис. 2.9 Формат ICMP-сообщений Echo Request и Echo Reply

Широко известная команда *ping* доступна почти во всех системах TCP/IP, а ее работа основана на ICMP-сообщениях для эхо-запросов и эхо-ответов.

Маска адреса

Напомним, что организация может разделить поле своего локального адреса на часть подсети и часть хоста. Когда включается система, она может быть сконфигурирована так, что не будет заранее знать, сколько бит было присвоено полю адреса подсети. Чтобы выяснить этот вопрос, система посылает широковещательный *запрос на определение маски адреса* (Address Mask Request).

Ответ должен быть получен от сервера, авторизированного для управления маской адреса сервера. Обычно в качестве такого сервера применяется маршрутизатор, но может использоваться и хост. В ответе в полях сети и подсети установлены единицы, определяя 32-разрядное поле маски адреса!

Сервер маски адреса может быть сконфигурирован так, что, даже при отключении от сети на какое-то время, он будет далее передавать широковещательные сообщения *Address Mask Reply,* как только станет активным. Это предоставляет шанс на получение нужной информации системам, которые были запущены в то время, когда сервер был неактивен.

На рисунке 2.10 показан формат *запроса маски адреса* и *ответа* на него. Тип 17 применяется для запроса, а тип 18 —для ответа. В общем случае можно игнорировать идентификатор и последовательный номер.

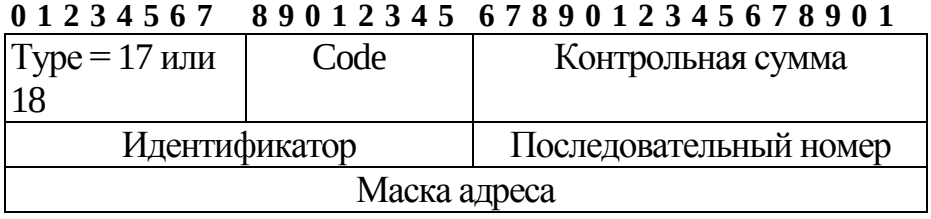


Рис. 2.10 Формат ICMP-сообщений Address Mask

На практике более предпочтительный метод определения маски адреса предоставляют протоколы загрузки, например *Dynamic Host Configuration Protocol* или *BOOTP.* Эти протоколы более эффективны, поскольку обеспечивают полный набор конфигурационных параметров. Кроме того, операции выполняются более точно, в том числе и некорректные.

Временная метка и ответ на Timestamp

Сообщение с ответом на *Timestamp* предоставляет сведения о времени в системе. Оно предназначено для оценки буферизации и обработки датаграммы на удаленной системе. Отметим следующие поля:

Originate timestamp (исходная временная метка)	Время последнего обращения к сообщению в системе-отправителе.
Receive timestamp временная метка получения)	Время первого обращения к сообщению отвечающей системы.
Transmit timestamp (временная метка пересылки)	Время последнего обращения к сообщению отвечающей системы.

По возможности, возвращаемое время должно измеряться в миллисекундах - относительно полуночи по универсальному времени (Universal Time), которое ранее называлось временем по Гринвичу (Greenwich Mean Time). Большинство реализаций реально возвращает одно и то же время в полях *Receive timestamp* и *Transmit timestamp.*

0 1 2 3 4 5 6 7	**8 9 0 1 2 3 4 5**	**6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1**
Type = 13 или 14	Code	Контрольная сумма
Идентификатор		Последовательный номер
Originate timestamp		
Receive timestamp		
Transmit timestamp		

Рис. 2.11 Формат ICMP-сообщений запросов и ответов о временной метке

Протокол ICMP обеспечивает очень простой способ синхронизации систем по времени. Однако это несколько грубая синхронизация, поскольку на нее влияют задержки в сети. Существует более совершенный *протокол сетевого времени* (Network Time Protocol), который был разработан для синхронизации по времени в Интернете.

Тип 13 используется для *запросов,* а 14 — для *ответов.* Формат сообщения представлен на рисунке 2.11.

2.4.Маршрутизация в сетях TCP/IP

Internet изначально строилась как сеть, объединяющая большое количество существующих систем. С самого начала в ее структуре выделяли *магистральную сеть (core backbone network)*, а сети, присоединенные к магистрали, рассматривались как *автономные системы (autonomous systems, AS)* [2, 3]. Магистральная сеть и каждая из автономных систем имели свое собственное административное управление и протоколы маршрутизации. Далее, маршрутизаторы по традиционной терминологии Internet будем называть *шлюзами (gateways).*

Шлюзы, которые используются для образования сетей и подсетей автономной системы, называются *внутренними шлюзами (interior gateways)*, а шлюзы, с помощью которых автономные системы присоединяются к магистральной сети, называются *внешними шлюзами (exterior gateways).* Магистраль сети также является автономной системой. Все автономные системы имеют уникальный 16-ти разрядный номер, который выделяется организацией InterNIC.

Протоколы маршрутизации внутри автономных систем называются *протоколами внутренних шлюзов (interior gateway protocol, IGP)*, а протоколы, определяющие обмен маршрутной информацией между внешними шлюзами и

шлюзами магистральной сети – *протоколами внешних шлюзов (exterior gateway protocol, EGP)*. Внутри магистральной сети также допустим любой собственный внутренний протокол IGP.

Смысл разделения всей сети Internet на автономные системы – в ее многоуровневом модульном представлении, что необходимо для любой крупной системы, способной к расширению в больших масштабах. Изменение протоколов маршрутизации внутри какой-либо автономной системы никак не должно повлиять на работу остальных автономных систем. Кроме того, деление Internet на автономные системы должно способствовать сбору информации в магистральных и внешних шлюзах. Внутренние шлюзы могут использовать для внутренней маршрутизации достаточно подробные графы связей между собой для выбора наиболее рационального маршрута. Детальная топологическая информация остается внутри автономной системы, а автономную систему как единое целое для остальной части Internet представляют внешние шлюзы, которые сообщают о внутреннем составе автономной системы минимально необходимые сведения: количество IP-сетей, их адреса и внутреннее расстояние до этих сетей от данного внешнего шлюза.

Виды маршрутизации

Существуют два вида маршрутизации: *прямая (direct routing)* и *косвенная (indirect routing)*.

Вид маршрутизации определяется следующими шагами.

1. Выделяется адрес сети (подсети) получателя пакета путем наложения маски на его IP-адрес.
2. Хост-отправитель сравнивает номер сети назначения пакета со своим номером сети. При совпадении номера сети IP-адреса получателя с адресом сети хоста-отправителя определяется, что пакет может быть прямо передан внутри локального сегмента сети без помощи маршрутизатора (прямая). Иначе, будет производиться поиск по таблице маршрутизации шлюза (маршрутизатора) через который данный пакет будет отправлен в сеть назначения (косвенная).

При косвенной маршрутизации, если сеть назначения пакета принятого маршрутизатором, не подсоединена к нему непосредственно, то данный маршрутизатор должен воспользоваться услугами другого маршрутизатора, который и будет определять следующий пункт назначения (когда не указан точный маршрут следования в заголовке пакета). Последний маршрутизатор будет использовать прямую маршрутизацию для доставки пакета в сеть назначения.

Структура таблицы маршрутизации

Таблица маршрутизации состоит из следующих основных полей:

- **Destination** (Адрес назначения) – содержит адрес сети или хоста назначения.

- **Gateway** (Шлюз) – содержит адрес шлюза (маршрутизатора), через который будут отправляться пакеты по адресу, указанному в поле Destination.
- **Mask** (Маска IP-адреса) – содержит маску для определения того, что обозначает IP-адрес поля назначения – хост или сеть определенной размерности.
- **Flags** (Флаги) – содержит информацию о состоянии указанной записи маршрутизации.
  U (Up) – показывает, что маршрут активен;
  H (Host) – указывает, что адресом назначения является адрес хоста;
  G (Gateway) – указывает, что маршрут пакета проходит через промежуточный маршрутизатор;
  S (Static) – указывает, что запись была введена вручную (статически);
  D (Dynamic) – указывает, что запись была создана динамически протоколом маршрутизации;
  M (Modified) – указывает, что запись была изменена динамически протоколом маршрутизации;
  R или ! (Reject) – указывает, что адрес назначения не доступен.
- **Metric** (Метрика) – расстояние до указаной сети (хоста).
- **Refcnt** (Счетчик ссылок) – сколько раз на данный маршрут ссылались при движении пакетов.
- **Use** – количество пакетов, переданных по данному маршруту.
- **Interface** – имя интерфейса в Unix-маршрутизаторах.

Источники записей в таблице маршрутизации

- Первым источником является ПО стека TCP/IP. При инициализации маршрутизатора это ПО автоматически заносит в таблицу несколько записей о локальных сетях и маршрутизаторе по умолчанию, о внутренних специальных адресах. Создается так называемая *минимальная таблица маршрутизации*.
- Вторым источником записей является администратор системы, непосредственно формирующий запись с помощью системной утилиты (например, route). Эти записи могут быть как постоянными, так и временными, т.е. хранящимися до перезагрузки системы.
- Третьим источником являются протоколы маршрутизации (RIP, OSPF). Такие записи всегда являются динамическими и имеют ограниченный срок жизни.

Маршрутизация с использованием масок

Допустим, администратор получил в свое распоряжение адрес класса B: 138.44.0.0 [2, 3]. Он может организовать сеть с большим числом узлов, номера которых он может брать из диапазона 0.0.0.1-0.0.255.254. Требуется, чтобы сеть была разделена на три отдельных подсети, при этом трафик в каждой подсети должен быть локализован. Это позволит легче диагностировать сеть и

проводить в каждой из подсетей свою политику безопасности.

В качестве маски было выбрано значение 255.255.192.0. После наложения маски на этот адрес число разрядов, интерпретируемых как номер сети, увеличилось с 16 (стандартная маска сети класса B) до 18. Это позволяет сделать из одного, централизованно заданного ему номера сети, четыре: 138.44.0.0; 138.44.64.0; 138.44.128.0; 138.44.192.0 (см. рис. 2.12).

Замечание. Некоторые программные и аппаратные маршрутизаторы не поддерживают номера подсетей, которые состоят либо только из одних нулей, либо только из одних единиц. Например, для некоторых типов оборудования номер сети 138.44.0.0 с маской 255.255.192.0, использованный в нашем примере, окажется недопустимым, поскольку в этом случае разряды в поле номера подсети имеют значение 00.

Рассмотрим, как изменяется работа модуля IP, когда становится необходимым учитывать наличие масок. Во-первых, в каждой записи таблицы маршрутизации появляется новое поле — поле маски. Во-вторых, меняется алгоритм определения маршрута по таблице маршрутизации. После того как IP-адрес извлекается из очередного полученного IP-пакета, необходимо определить адрес следующего маршрутизатора, на который надо передать пакет с этим адресом. Модуль IP последовательно просматривает все записи таблицы маршрутизации. С каждой записью производятся следующие действия.

- Маска, содержащаяся в данной записи, накладывается на IP-адрес узла назначения, извлеченный из пакета.
- Полученное в результате число является номером сети назначения обрабатываемого пакета. Оно сравнивается с номером сети, который помещен в данной записи таблицы маршрутизации.
- Если номера сетей совпадают, то пакет передается маршрутизатору, адрес которого помещен в соответствующем поле данной записи.

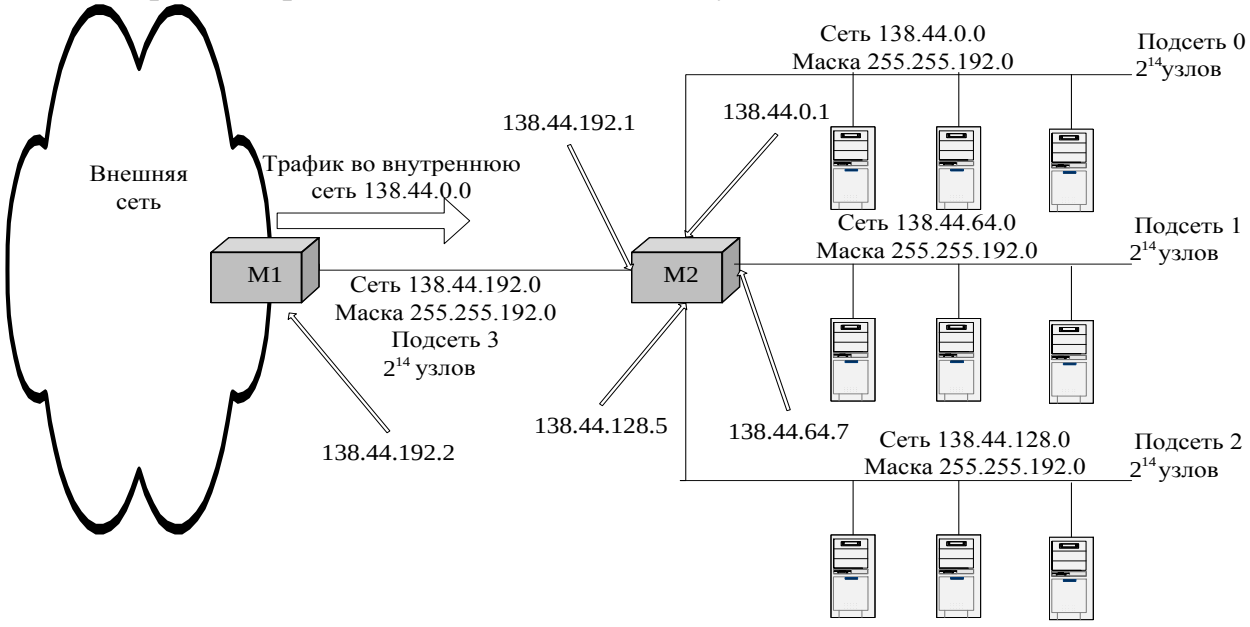


Рис. 2.12 Пример организации сети

Таблица 2.7 Таблица маршрутизатора М2 в сети с масками одинаковой длины

Номер сети	Маска	Адрес следующего маршрутизатора	Адрес порта	Расстояние
138.44.0.0	255.255.192.0	138.44.0.1	138.44.0.1	Подключена
138.44.64.0	255.255.192.0	138.44.64.7	138.44.64.7	Подключена
138.44.128.0	255.255.192.0	138.44.128.5	138.44.128.5	Подключена
138.44.192.0	255.255.192.0	138.44.192.1	138.44.192.1	Подключена
0.0.0.0	0.0.0.0	138.44.192.2	138.44.192.1	—

Пусть, например, с маршрутизатора Ml на порт 138.44.192.1 маршрутизатора М2 поступает пакет с адресом назначения 138.44.78.200. Модуль IP начинает последовательно просматривать все строки таблицы, до тех пор пока не найдет совпадения номера сети в адресе назначения и в строке таблицы. Маска из первой строки 255.255.192.0 накладывается на адрес 138.44.78.200, в результате чего получается номер сети 138.44.64.0. Полученный номер 138.44.64.0 совпадает с номером сети во второй строке таблицы, а значит, пакет должен быть отправлен на порт маршрутизатора 138.44.64.7 в сеть непосредственно подключенную к данному маршрутизатору.

2.5.Протоколы маршрутизации

2.5.1.RIP протокол

Алгоритм маршрутизации протокола RIP (Roster Image Processor) принадлежит к классу дистанционно-векторных алгоритмов [1 – 3]. Данный класс алгоритмов также известен по имени автора алгоритма Форда-Фолкерсона (Ford-Folkerson). Помимо того, для этого класса также используется название «алгоритмы Белмана-Форда» (Bellman-Ford), которое появилось после окончательной формализации алгоритма, которая была сделана на основании основного уравнения динамического программирования Белмана.

Данный алгоритм маршрутизации применяется внутри автономных систем и относится к классу протоколов IGP – Interior Gateway Protocol. Он построен на основе механизма обмена небольшими блоками информации таблиц маршрутизации между соседними маршрутизаторами сети. Т.о., каждый шлюз или хост, участвующий в работе протокола, хранит у себя информацию о всех членах сети в виде базы данных маршрутизации (таблицы).

Оптимальный путь – это путь с наименьшей «длиной» («metric»), которая может определяться исходя из требований алгоритма. Метрика может определяться либо на основе количества шлюзов, которые необходимо преодолеть по пути к получателю, либо на основе таких параметров, как суммарное время задержки пересылки пакета, стоимость канала связи и т.п.

Маршрутизатор работающий с протоколом RIP должен:

1. Хранить таблицу маршрутизации с записями каждого потенциального получателя пакета в системе. Запись должна содержать расстояние до объекта (D) и адрес первого шлюза (G) на пути к этому объекту.
2. Периодически отправлять информационные сообщения, содержащие всю информацию своей таблицы, каждому из своих соседей по сети (объектам, находящимся в области прямой видимости).
3. При получении информационного сообщения от соседа G^*, которое содержит его таблицу маршрутизации считать из полученного сообщения метрики сети и добавить к ним метрику сети до соседа G^* (по этой сети поступило данное сообщение). Сравнить результаты с результатами собственной таблицы маршрутизации. Если какая-либо метрика D^* до объекта N меньше существующей метрики до объекта N в собственной таблице, изменить запись в собственной базе данных маршрутизации для этого объекта (метрику на D^*, а шлюз на G^*).

Описанный выше алгоритм подразумевает, что топология сети не меняется. При выходе каналов из строя, протоколы дистанционно-векторной маршрутизации должны применять меры по синхронизации установленных маршрутов.

Характеристики протокола RIP

1. Протокол не может использоваться в сетях, где количество пересылок (промежуточных шлюзов на пути пакетов) превышает 15. Если необходимы маршруты с большим количеством пересылок, то следует использовать более мощный протокол маршрутизации или разбить сеть на более мелкие самостоятельные сегменты.
2. RIP-протокол имеет ограничение на время восстановления путей маршрутизации. Поэтому если сеть состоит из большого количества подсетей и цикл обновления маршрутов касается всех этих подсетей, то необходимо либо увеличить пропускную способность каналов, либо увеличить ограничение на время восстановления путей маршрутизации.
3. RIP-протокол использует статические «метрики» для сравнения различных маршрутов. Это неудобно в тех случаях, когда маршрут необходимо выбрать на основании временных характеристик, например, загрузки канала, надежности, задержки. При оценки маршрутов по данным параметрам протокол RIP не используется.

Механизмы работы протокола RIP

1. *Механизм ограничения числа пересылок* и феномен «счета до бесконечности».

Образование маршрутных петель при выходе каналов из строя и циклическом увеличении метрики за счет информации, поступающей из маршрутных таблиц соседних маршрутизаторов, является ограничением области применения протокола, позволяя сделать его работу более устойчивой.

Рассмотрим, что произойдет, если канал маршрутизатора «А», связывающий его с сетью «N», откажет. «А» проверяет свою информацию и

обнаруживает, что маршрутизатор «B» связан с сетью «N» каналом длиной в одну пересылку. Т.к. «A» знает, что он напрямую соединен с «B», то он объявляет о маршруте из двух пересылок до сети «N» и начинает направлять весь трафик в сеть «N» через «B». Это приводит к образованию маршрутной петли. Т.е. когда «B» обнаруживает, что «A» может теперь достичь сеть «N» за две пересылки, он изменяет запись своих собственных данных в таблице маршрутизации, чтобы показать, что он имеет канал длиной в 3 пересылки до сети «N» и т.д. Количество пересылок на данном маршруте в таблицах маршрутизаторов будет расти до бесконечности.

Так будет продолжаться до тех пор, пока не будет навязано какое-нибудь внешнее граничное условие. Этим граничным условием является максимальное число пересылок RIP. Когда число пересылок превысит 15, данный маршрут маркируется как недосягаемый. Он помечается числом 16, точно так же, как и маршрут, отсекаемый при работе основного алгоритма, если он проходил через вышедший из строя шлюз. Через некоторое время, этот маршрут удаляется из таблицы.

2. *Механизм временного удерживания изменений* (hold-downs).

Данный механизм используется для того, чтобы помешать регулярным сообщениям о корректировке незаконно восстановить в правах маршрут, который оказался испорченным. При отказе какого-либо маршрута соседние маршрутизаторы обнаруживают это, вычисляют новые маршруты и отправляют сообщения об обновлении маршрутизации, чтобы информировать своих соседей. Эта деятельность приводит к появлению целой волны коррекций маршрутизации через сеть. Команды о временном удерживании указывают маршрутизаторам, чтобы они на некоторое время придержали любые изменения, которые могут оказать влияние на только что удаленные маршруты. Данный период удерживания обычно рассчитывается т.о., чтобы он был больше периода времени, необходимого для внесения какого-либо изменения о маршрутизации во всю сеть. Именно этот механизм определяет промежуток времени восстановления маршрутизации и синхронизации маршрутов при сбое в какой-либо из компонент сети.

3. *Расщепленные горизонты* (split-horizons).

Механизм, в некотором роде, определяет объем передаваемой шлюзами друг другу информации о состоянии маршрутов сети, используя тот факт, что не следует отправлять информацию о каком-нибудь маршруте обратно в том же направлении, из которого пришла эта информация. Данный механизм как бы ограничивает поле видимости (горизонт) маршрутизатора, не позволяя появляться петлям маршрутизации между двумя соседними хостами при изменениях топологии сети, касающихся одного из соседей.

Также существует механизм «расщепленных горизонтов с испорченной обратной связью» (split horizons with poisoned reverse). Он отличается тем, что обратный маршрут не исключается из таблицы маршрутизации вовсе, а ему присваивается статус 16 (недостижимый) и он участвует во всех операциях алгоритмов построения оптимального маршрута.

4. *Механизм триггерных изменений* (triggered updates).

Механизм триггерных изменений позволяет увеличить скорость сходимости алгоритмов маршрутизации за счет локализации рассылаемых изменений.

Триггерные изменения представляют собой команды изменения метрики маршрута, которые принимаются во внимание только теми объектами сети, которые работают с сетью через этот шлюз. Механизм рассылает предполагаемые новые маршруты, которые в процессе обработки либо принимаются – «закрываются», либо не принимаются хостом – остаются «мнимыми». Т.о. происходит направленное или локализованное изменение метрик только определенных шлюзов. Триггерные сообщения имеют приоритет значительно больший чем обычные сообщения об изменении маршрута.

5. Таймеры.

Обновление маршрутов протокол RIP синхронизирует по таймеру. Каждые 30 сек. маршрутизаторы рассылают своим соседям состояние своих таблиц маршрутизации.

Помимо этого, каждая запись синхронизируется по таймеру. Если существование маршрута не подтверждается в течение 180 сек. (ни от одного из соседей не приходит ответ на запрос о таком маршруте), то данный маршрут помечается как "нерабочий". Он уже не используется хостом и его соседями, но еще некоторое время сохраняется в таблице.

Один раз в 120 сек. система производит операцию "сборки мусора". В этом процессе в таблицах маршрутизации удаляются записи, которые либо помечены как "нерабочие", либо метрика которых равна 16, что в спецификации RIP означает – "недостижимый маршрут", либо запись помечена как "незакрытая триггером", что в спецификации RIP означает – "мнимая".

Формат RIP-пакета

RIP работает на основе UDP-протокола и использует порт 520. На рисунке 2.13 представлен формат заголовка RIP-пакета.

Offset	0 7	8 15	16 31
0	Command	Version	Zero
4	Address Family Identifier		Zero
8	Address		
12	Zero		
16	Zero		
20	Metric		

Рис. 2.13 Формат RIP-пакета

Команда – Command (8 бит) – содержит число, обозначающее либо запрос, либо ответ. Команда-запрос запрашивает хост или маршрутизатор об отправке всей таблицы маршрутизации или ее части. Пункты назначения, для которых запрашивается ответ, перечисляются далее в данном пакете. Ответная команда представляет собой ответ на запрос или какую-нибудь

незатребованную регулярную корректировку маршрутизации. Отвечающая система включает в ответный пакет всю таблицу маршрутизации или ее часть. Регулярные сообщения о корректировке маршрутизации включают в себя всю таблицу маршрутизации.

Версия – Version (8 бит) – определяет реализуемую версию RIP.

Нули – Zero – заполнено нулями.

Идентификатор семейства адресов – Address Family Identifier (16 бит) – определяет конкретное семейство адресов. Для сети Internet и протокола IP это значение равно 2.

Адрес – Address (32 бита) – для сети Internet содержит какой-либо IP-адрес хоста, сети либо подсети.

Метрика – Metric (32 бита) – представляет собой число пересылок (hop count) или транзитных участков (маршрутизаторов) сети, прежде чем можно будет добраться до пункта назначения.

В каждом отдельном пакете RIP может быть перечислено до 25-ти пунктов назначения. Для передачи информации из более крупных маршрутных таблиц используется множество пакетов RIP.

2.5.2.OSPF протокол

OSPF (Open Shortest Path First) – это протокол маршрутизации, базирующийся на алгоритме поиска наикратчайшего пути SPF (Shortest Path First) [1 – 3]. Алгоритм SPF иногда называют алгоритмом Дейкстра по имени его автора. Основанием для разработки OSPF была очевидная непригодность RIP для обслуживания крупных гетерогенных систем.

OSPF является иерархическим протоколом маршрутизации с объявлением состояния о канале соединения (link-state). Он был спроектирован как протокол работы внутри сетевой области – AS (Autonomous System), которая представляет собой группу маршрутизаторов и сетей, объединенных по иерархическому принципу и находящихся под единым управлением и совместно использующих общую стратегию маршрутизации. В качестве транспортного протокола для маршрутизации внутри AS OSPF использует протокол IP.

Обмен информацией о маршрутах внутри AS протокол OSPF осуществляет посредством обмена сообщениями о состояниях канала соединений между маршрутизаторами и сетями области (link-state advertisment – LSA). Эти сообщения передаются между объектами сети, находящимися в пределах одной и той же иерархической области – это может быть как вся AS, так и некоторая группа сетей внутри данной AS. В LSA-сообщения протокола OSPF включается информация о подключенных интерфейсах, о параметрах маршрутов и других переменных. По мере накопления маршрутизаторами OSPF информации о состоянии маршрутов области, они рассчитывают наикратчайший путь к каждому узлу, используя алгоритм SPF. Причем расчет оптимального маршрута осуществляется динамически в соответствии с изменениями топологии сети.

Для различных типов IP-сервиса (видов услуг высшего уровня, которые определяются значением поля TOS IP-пакета), OSPF может рассчитывать свои оптимальные маршруты на основании параметров, наиболее критичных для данного вида сервиса. Например, какая-нибудь прикладная программа может включить требование о том, что определенная информация является срочной. Если OSPF имеет в своем распоряжении каналы с высоким приоритетом, то они могут быть использованы для транспортировки срочных датаграмм.

OSPF поддерживает механизм, позволяющий работать с несколькими равноправными маршрутами между двумя объектами сети. Это позволяет существенно уменьшить время передачи данных и более эффективно использовать каналы связи.

Кроме того, протокол OSPF поддерживает аутентификацию изменений маршрутов. Это означает, что только те маршрутизаторы, которые имеют определенные права, могут осуществлять маршрутизацию пакетов. Это позволяет, при соответствующей настройке прав системы маршрутизаторов, передавать по сети конфиденциальные сообщения, зная заранее, что они проходят только по определенным маршрутам.

Принцип работы

Алгоритм маршрутизации SPF является основой для операции OSPF. В каждой области работает отдельная копия алгоритма маршрутизации. Маршрутизаторы, которые имеют интерфейсы к нескольким областям, работают с несколькими копиями алгоритма SPF.

После включения и инициализации своих структур данных о протоколе маршрутизации, маршрутизатор ожидает уведомления от протоколов низшего уровня о том, что его интерфейсы работоспособны.

После получения подтверждения о работоспособности своих интерфейсов, маршрутизатор использует протокол приветствия (hello protocol) OSPF, чтобы приобрести соседей (neighbor). Соседи – это объекты сети с интерфейсами, предназначенными для работы в общей с данным маршрутизатором сети. Описываемый маршрутизатор отправляет своим соседям приветственные пакеты и получает от них такие же пакеты. Помимо оказания помощи в приобретении соседей, приветственные пакеты также действуют как подтверждение работоспособности, позволяя маршрутизаторам узнавать о том, что другие маршрутизаторы функционируют.

Каждый маршрутизатор периодически, в зависимости от настройки системы, отправляет сообщение о состоянии канала (LSA-message). Эти сообщения содержат информацию о состоянии интерфейса данного маршрутизатора и смежных с ним объектов сети. Каждое такое сообщение рассылается маршрутизаторам всей области. Из LSA-сообщений всех объектов формируется топологическая база данных (дерево маршрутов). Сообщения LSA также отправляются в том случае, когда изменяется состояние какого-нибудь маршрутизатора.

После построения дерева маршрутов внутри AS, протокол проверяет информацию внешних маршрутов по отношению к данной AS. Эта информация

может быть получена с помощью протоколов, обеспечивающих, взаимодействие OSPF с другими областями или другими типами сетей, например с помощью протоколов EGP и BGP.

Все маршрутизаторы данной AS используют один и тот же алгоритм построения маршрута на основе топологической базы данных. Маршрутизатор строит граф оптимальных маршрутов, в котором он сам является корнем. На основании этого графа маршрутизаторы и производят свои расчеты для каждого маршрута информационного пакета. В свою очередь, по дереву оптимальных маршрутов строится маршрутная таблица, которая служит основой оценок и выбора маршрутов.

Формат пакета OSPF

Существует пять типов OSPF-пакетов. Все пакеты начинаются со стандартного 24-байтного заголовка (рис. 2.14).

Offset	0 7	8 15	16 31
0	Version	Type	Packet Length
4	Router ID		
8	Area ID		
12	Checksum		Autype
16	Authentication		
20	Authentication Data		

Рис. 2.14 Формат OSPF-пакета

Версия – Version – (1 байт) – содержит номер версии OSPF пакета протокола, использующего данный пакет.
Тип – Type – (1 байт) – содержит тип сообщения
1 – Hello
2 – Database Description
3 – Link-State Request
4 – Link-State Update
5 – Link-State Acknowledgement
Длина пакета – Packet Length – (16 бит) – содержит длину пакета (в байтах) вместе со стандартным заголовком.
Идентификатор маршрутизатора – RouterID – (32 бита) – содержит идентификатор маршрутизатора.
Идентификатор области – AreaID – (32 бита) – идентифицирует область, к которой принадлежит данный пакет.
Контрольная сумма – Checksum – (16 бит) – поле контрольной суммы пакета.
Тип аутентификации – Authentication – (16 бит) – поле типа аутентификации. Например, «простой пароль». Все обмены протокола OSPF проводятся с аутентификацией отправителя и его прав. Тип аутентификации устанавливается по принципу «отдельный для каждой области».
Информация аутентификации – Authentication Data – (64 бита) – содержит информацию аутентификации.

2.5.3.EGP протокол

Основной функцией EGP (Exterior Gateway Protocol) является обеспечение взаимодействия различных автономных систем – AS, так что для конечного пользователя все разнородные группы и домены сетей Internet будут казаться единым плоским пространством [1 – 3].

EGP не является протоколом маршрутизации пакетов данных. Он предназначен для обеспечения взаимодействия между шлюзами различных AS, для обмена информацией согласования алгоритмов маршрутизации между ними, а не для управления перемещением самой информации. Иначе говоря, каждая из AS может работать со своим IGP (RIP или OSPF), а EGP осуществляет управление маршрутизацией между AS. Все изменения, обеспечивающие доступ AS к главной магистрали – backbone, должны быть также сделаны самой AS. Кроме того, EGP не строит схем и алгоритмов маршрутизации данных.

Любая часть EGP-сети Internet должна представлять собой структуру дерева, у которой стержневой маршрутизатор является корнем, и в пределах которого отсутствуют петли между другими AS. Это ограничение является основным ограничением EGP. Оно стало причиной его постепенного вытеснения другими, более совершенными протоколами маршрутизации.

EGP первоначально предназначался для передачи информации о достижимости узлов в стержневые маршрутизаторы ARPANET и получения от них этой информации. Информация передавалась из отдельных узлов источника, находящихся в различных административных доменах – AS, вверх в стержневые маршрутизаторы, которые передавали эту информацию через стержневую область до тех пор, пока ее можно было передать вниз к сети пункта назначения, находящейся в пределах другой AS.

Схема работы

Схема работы протокола EGP очень проста. Он не может принимать интеллектуальных решений о маршрутизации. Корректировки маршрутизации EGP содержат информацию только о достижимости сетей. Другими словами, они указывают, что в определенные сети информационные пакеты попадают через определенные маршрутизаторы.

EGP выполняет три основные функции:

1. Маршрутизаторы, работающие с EGP, организуют для себя определенный набор соседей. Соседи – это просто другие маршрутизаторы, с которыми какой-нибудь маршрутизатор хочет коллективно пользоваться информацией о достижимости сетей (какие-либо указания о географическом соседстве не включаются).
2. Маршрутизаторы EGP опрашивают своих соседей для того, чтобы убедиться в их работоспособности.
3. Маршрутизаторы EGP отправляют сообщения о корректировках, содержащих информацию о достижимости сетей в пределах своих AS.

Для реализации этих функции протокол использует систему следующих сообщений:

- *Приобретение соседа (Neighbor acquisition)*. Прежде чем начать получать информацию от внешних маршрутизаторов, необходимо установить, какой маршрутизатор является соседним. Эта операция состоит из обмена сообщениями типа «приобретение соседа» (соответственно запрос/ответ/отказ и др.) через стандартный механизм трехходового квитирования. Маршрутизатор предполагаемого соседа также должен поддерживать механизм сообщений типа «приобретение соседа». Данное сообщение включает в себя поле интервала приветствия (hello interval) и поле интервала опроса (poll interval). Поле интервала приветствия определяет период интервала проверки работоспособности соседей. Поле интервала опроса определяет частоту корректировки маршрутизации.
- Достижимость соседа (Neighbor reachability). Для маршрутизаторов, выполняющих функции связи различных доменов сетей, важно располагать самой последней информацией о работе своих соседей. Если маршрутизатор обнаруживает, что какой-либо шлюз не функционирует, ему необходимо немедленно приостановить поток данных к этому шлюзу. Для этих целей и используется данный вид сообщений.
- EGP-протокол поддерживает два вида сообщений этого типа – сообщение приветствия (hello) и ответа на приветствие (I heard you). Выделение типа сообщений оценки достижимости из общего потока корректирующих сообщений позволяет уменьшать сетевой трафик, т.к. изменения о достижимости сетей обычно появляются чаще, чем изменения параметров маршрутизации. Любой узел EGP заявляет об отказе одного из своих соседей только после того, как от него не был получен определенный процент сообщений о достижимости.
- Опроса (Poll). Для обеспечения правильной маршрутизации между AS, EGP должен иметь информацию об относительном местоположении отдаленных хостов. Данные сообщения позволяют маршрутизаторам EGP получать информацию о достижимости сетей, в которых находятся эти машины. Такие сообщения имеют помимо обычного заголовка только одно поле – поле сети источника IP (source network). Это поле определяет сеть, которая должна использоваться в качестве контрольной точки опроса.
- Корректировки маршрутизации (Routing update). Сообщения о корректировке маршрутизации дают маршрутизаторам EGP возможность указывать местоположение различных сетей в пределах своих AS.
- В дополнение к обычному заголовку эти сообщения включают несколько дополнительных полей. Поле числа внутренних маршрутизаторов (number of interior gateways) указывает на число внутренних маршрутизаторов, появляющихся в сообщении. Поле числа внешних маршрутизаторов (number of exterior gateways) указывает на число внешних маршрутизаторов, появляющихся в сообщении. Поле сети источника IP (IP source network)

указывает IP-адрес той сети, до которой измерена достижимость. За этим полем идет последовательность блоков маршрутизаторов (gateway blocks). Каждый блок маршрутизаторов предоставляет IP-адрес какого-нибудь маршрутизатора и перечень сетей, а также расстояний, связанных с достижением этих сетей.

- EGP фактически использует поле расстояния только для указания существования какого-либо маршрута. Значение расстояния может быть использовано только для сравнения трактов, если эти тракты полностью находятся в пределах одной конкретной AS. По этой причине EGP является скорее протоколом достижимости, чем протоколом маршрутизации.
- О неисправностях (Error). Сообщения о неисправностях указывают на различные сбойные ситуации. В дополнение к общему заголовку EGP сообщения о неисправностях обеспечивают поле причины (reason), за которым следует заголовок сообщения о неисправностях (message header). В число типичных неисправностей EGP входят неправильный формат заголовка EGP (bad EGP header format), неправильный формат поля данных EGP (bad EGP data field format), чрезмерная скорость опроса (excessive polling rate) и невозможность достижимости информации (unavailability of reachability information).

Формат заголовка EGP-пакета

На рисунке 2.15 представлен формат заголовка EGP-пакета.

Offset	0 7	8 15	16 31
0	EGP Version	Type	Code	Status
4	Checksum		Autonomous System
8	Sequence		Data …

Рис. 2.15 Формат заголовка EGP-пакета

Номер версии EGP (EGP Version) – 8 бит – обозначает текущую версию EGP и проверяется приемными устройствами для определения соответствия между номерами версий отправителя и получателя.

Тип (Type) – 8 бит – обозначает один из 5 типов сообщений EGP.

Код (Code) – 8 бит – определяет различие между подтипами сообщений.

Статус (Status) – 8 бит – содержит информацию о состоянии, зависящую от сообщения. В число кодов состояния входят коды недостатка ресурсов (insufficient resources), неисправных параметров (parameter problem), нарушений протокола (protocol violation) и др.

Контрольная сумма (Checksum) – 16 бит – используется для обнаружения возможных проблем, которые могли появиться в пакете в результате его транспортировки.

Номер автономной системы (Autonomous System Number) – 16 бит – обозначает AS, к которой принадлежит шлюз отправитель.

Номер последовательности (Sequence Number) – 16 бит – позволяет двум маршрутизаторам EGP, которые обмениваются сообщениями, соотносить сообщения запросов с сообщениями ответов. Когда определен какой-нибудь

новый сосед, номер последовательности устанавливается в исходное нулевое значение и увеличивается на единицу с каждой новой транзакцией запрос-ответ.

За заголовком EGP идут дополнительные поля, содержимое которых различается в зависимости от типа сообщения.

2.5.4.BGP протокол

Протокол граничных маршрутизаторов BGP (Border Gateway Protocol) является протоколом маршрутизации между AS и в отличие от EGP предназначен для обнаружения маршрутных петель. Как и EGP, протокол BGP относится к классу «междоменных протоколов» [1, 2, 4].

Основным предназначением BGP является обеспечение обмена информацией с другими BGP-системами о достижимости определенных сетей или хостов. Эта информация должна содержать набор маршрутов к данной сети, т.е. должны быть указаны все промежуточные AS. Такой информации вполне достаточно для построения графа соединений между AS и контроля на возможные маршрутные петли. На основании этих данных BGP выбирает оптимальный маршрут и передает эту информацию своим соседям.

Хосты, работающие с BGP, не принимают участие в процедуре маршрутизации информационных пакетов. Они предназначены только для обмена информацией с маршрутизаторами других AS. Два соседних маршрутизатора BGP, из различных AS, для открытия соединения должны находится в одной и той же физической сети. Маршрутизаторы BGP, находящиеся в пределах одной и той же AS, сообщаются друг с другом, чтобы обеспечить согласование представлений о данной AS и определить, какой из маршрутизаторов BGP данной AS будет служить в качестве точки соединения при передаче или приеме сообщений во внешние AS. BGP должен взаимодействовать с любыми протоколами маршрутизации внутри AS.

Между двумя AS BGP передает информацию самостоятельно. По AS передача информации осуществляется вместе с каким-либо протоколом класса IGP.

Два хоста устанавливают друг с другом соединение, обмениваясь сообщениями открытия и согласования параметров соединения. Инициализация потока данных включает в себя передачу всей таблицы маршрутов BGP. С изменением маршрутной таблицы отправляются соответствующие корректировки.

Периодически хосты отправляют друг другу сообщения подтверждения своей работоспособности, например, при возникновении ошибочных ситуаций передаются сообщения об ошибках. BGP не требует периодического обновления всей маршрутной таблицы, хотя BGP поддерживает маршрутную таблицу всех возможных трактов к какой-нибудь конкретной сети, в своих сообщениях о корректировке он объявляет только об основных – оптимальных маршрутах.

Показатели оптимальности – «метрики» BGP представляют собой числа, характеризующие степень предпочтения какого-нибудь конкретного маршрута.

Эти показатели обычно определяются администратором сети с помощью конфигурационных файлов. Степень предпочтения может базироваться на любом числе критериев, включая число промежуточных AS (например, тракты с меньшим числом AS, как правило, лучше), тип канала, стабильность, быстродействие, надежность канала и другие факторы.

BGP работает поверх протокола транспортного уровня. Например, при работе поверх TCP BGP использует порт 179. Это позволяет не нагружать сервисы обработки протокола BGP механизмами фрагментации или обеспечения достоверности доставки пакетов. Схемы аутентификации протоколов транспортного уровня также могут быть использованы BGP в дополнение к собственной схеме аутентификации. Кроме того, хотя BGP разработан как протокол маршрутизации между AS, он может использоваться для маршрутизации и внутри AS.

Формат заголовка BGP-пакета

На рисунке 2.16 представлен формат заголовка BGP-пакета.

Offset	0 15 16 23 24 31		
0	Marker		
4	Length	Type	

Рис. 2.16 Формат заголовка BGP-пакета

Маркер – Marker (16 бит) – содержит величину, которую получатель сообщения может прогнозировать (например, при использовании механизма аутентификации). Это поле также может использовано для обнаружения потерянной синхронизации между парой BGP-маршрутизаторов.

Длина – Length (16 бит) – содержит полную длину сообщения в байтах, включая заголовок и данные.

Тип – Type (8 бит) – указывает тип сообщения (Open, Notification, Update, Keepalive, Errors).

За основным заголовком следует структура данных, определяемая типом сообщения.

Сообщения BGP

Для установления соединения, коррекции маршрутов, уведомления друг друга BGP-маршрутизаторы используют следующую систему сообщений:

- Открывающие сообщения (open message). После того, как организовано соединение протокола транспортного уровня, первым сообщением, отправляемым каждой стороной, является открывающее сообщение. Пакет открывающего сообщения содержит, в дополнение к основному заголовку BGP, несколько дополнительных параметров. Параметр версия (version) – номер версии BGP дает возможность получателю проверять, совпадает ли его версия с версией отправителя. Параметр автономная система (autonomous system) содержит номер AS отправителя. Параметр время удержания (hold time) указывает максимальное число секунд, которые могут пройти без получения

какого-либо сообщения от передающего устройства, прежде чем считать его отказавшим. Параметр код аутентификации (authentication code) указывает на используемый код удостоверения, если конечно, таковой имеется. Параметр данных аутентификации (authentication data) содержит возможные данные аутентификации.

- Сообщения о корректировке (update message). Сообщения о корректировках BGP обеспечивают корректировки маршрутизации для других систем BGP. Информация этих сообщений используется для построения графа, описывающего взаимоотношения между различными AS.
- В дополнение к стандартному заголовку BGP сообщения о корректировках содержат несколько дополнительных параметров. Эти параметры обеспечивают маршрутную информацию атрибутов трактов, соответствующих каждой сети. BGP определяет 5 атрибутов трактов.
- Источник (origin). Может иметь одно из трех значений: IGP, EGP и incomplete (незавершенный). Атрибут IGP означает, что данная сеть является частью данной AS. Атрибут EGP означает, что первоначальные сведения о данной информации получены от протокола EGP. Реализации BGP склонны отдавать предпочтение маршрутам IGP перед маршрутами EGP, так как маршрут EGP отказывает при наличии маршрутных петель. Атрибут incomplete используется для указания того, что о данной сети известно через какие-то другие средства.
- Путь (path). Путь AS. Обеспечивает фактический перечень AS на пути к пункту назначения.
- Следующая пересылка (next hop). Содержит адрес IP-маршрутизатора, который должен быть использован в качестве следующей пересылки к сетям, перечисленным в сообщении о корректировке.
- Недосягаемый (unreachable). Указывает, что какой-нибудь маршрут больше не является досягаемым.
- Показатель оптимальности маршрута между AS (inter-AS metric). Обеспечивает для какого-нибудь маршрутизатора BGP возможность тиражировать свои затраты на маршруты к пунктам назначения, находящимся в пределах его AS. Эта информация может использована маршрутизаторами, которые являются внешними по отношению к AS тиражирующего маршрутизатора, для выбора оптимального маршрута к конкретному пункту назначения, находящемуся в пределах данной AS.
- Сообщения «продолжай действовать» (keep alive message). Эти сообщения не содержат каких-либо дополнительных полей помимо тех, которые содержатся в заголовке BGP.
- Уведомления об ошибках (error message). Уведомления отправляются в том случае, если была обнаружена сбойная ситуация и один маршрутизатор хочет сообщить другому, почему он закрывает соединение между ними. В сообщении содержится код ошибки (error code), подкод ошибки (error subcode) и данные ошибки (error data).

Контрольные вопросы

1. Сколько бит занимает адрес IP v.4?
2. Для чего вводится классификация IP адресов?
3. Приведите пример IP-адресов класса A?
4. К какому классу IP адресов относится широковещательные адреса?
5. Для чего предназначена Маска адреса?
6. Имеется сеть с идентификатором 154.121.0.0. Требуется поддержка 2040 узлов в подразделении. Какую маску Вы бы посоветовали использовать?
7. Какой максимальный и минимальный размер заголовка протокола IP?
8. Какой максимально возможный срок жизни IP пакета в сети?
9. В чем заключается механизм фрагментации IP?
10. Сколько типов сообщений об ошибках представлено в протоколе ICMP?
11. Что такое MTU?
12. Как исследовать MTU на пути прохождения пакета?
13. Чем отличаются дистанционно-векторные протоколы от протоколов состояния связей?
14. Как образуются петли при использовании протокола?
15. В чем недостатки протокола EGP?
16. В чем достоинства протокола BGP?

ГЛАВА 3. ПРОТОКОЛЫ ТРАНСПОРТНОГО УРОВНЯ

Протоколы транспортного уровня осуществляют передачу данных между «прикладными процессами», выполняющимися на машинах, подключенных к сети [1 - 3]. Данные с сетевого уровня направляются сетевым программным обеспечением (ПО) конкретному процессу получателю и наоборот. На каждом компьютере может выполняться множество процессов, более того, прикладной процесс может иметь несколько точек входа, выступающих в качестве адреса назначения для пакетов данных.

Для однозначной идентификации сетевого приложения, работающего на машине сети, для протоколов транспортного уровня реализована концепция т.н. *портов*. Порт вместе с IP-адресом однозначно определяют прикладной процесс на любой машине сети. Данный набор идентификационных параметров называется *сокетом (socket)*. Порты задаются 16-битным числом от 0 до 65535.

Существует три типа номеров портов: *назначенные (assigned), зарегистрированные (registered)* и *динамические (dynamic)*. Назначенные номера портов располагаются в диапазоне 0 – 1023 и полностью контролируются Комиссией по константам Internet. Они применяются для общеизвестных и стандартизированных сетевых служб. Зарегистрированные номера портов от 1024 до 65535 предназначены для регистрации производителями сетевого ПО своих приложений, работающих с данными портами. Динамические номера портов присваиваются сетевым ПО на локальной машине и могут повторяться от станции к станции для различных приложений.

3.1.Протокол UDP. Заголовок пакета

Протокол UDP (User Datagram Protocol – протокол дейтаграмм пользователя) является не ориентированным на соединение транспортным протоколом с ненадежной доставкой данных [1 - 3]. Т.е. он не обеспечивает подтверждение доставки пакетов, не сохраняет порядок входящих пакетов, может терять пакеты или дублировать их. Функционирование UDP похоже на IP, за исключением введения понятия портов. UDP обычно работает быстрее TCP за счет меньших «накладных расходов». Он применяется приложениями, которые не нуждаются в надежной доставке, либо реализуют их сами. Например, сервера имен (Name Servers), служба TFTP (Trivial File Transfer Protocol, тривиальный протокол передачи данных), SNMP (Simple Network Management Protocol, простой протокол управления сетью), системы аутентификации. Идентификатор UDP протокола в поле *Protocol* заголовка IP – число 17.

Любая прикладная программа, использующая UDP в качестве своей службы транспортного уровня, должна сама обеспечить механизмы подтверждения и систему последовательной нумерации, чтобы гарантировать доставку пакетов в том же порядке, в котором они были высланы.

На рисунке 3.1 представлен формат заголовка UDP-пакета.

Offset	0 15 16 31	
0	Source Port	Destination Port
4	Length	Checksum

Рис. 3.1 Формат заголовка UDP-пакета

Назначение полей UDP пакета

Номер порта отправителя – Source Port (16 бит) – содержит номер порта, с которого был отправлен пакет, когда это имеет значение (например, отправитель ожидает ответа). Если это поле не используется, оно заполняется нулями.

Номер порта назначения – Destination Port (16 бит) – содержит номер порта, на который будет доставлен пакет.

Длина – Length (16 бит) – содержит длину данной дейтаграммы в байтах, включая заголовок и данные.

Поле контрольной суммы – Checksum (16 бит) – представляет собой побитное дополнение 16-битной суммы 16-битных слов. В вычислении суммы участвуют: данные пакета с полями выравнивания по 16-битной границе (нулевые), заголовок UDP-пакета, псевдозаголовок.

3.2. Протокол TCP

3.2.1.Заголовок пакета

Протокол TCP (Transmission Control Protocol – протокол управления передачей) является ориентированным на соединение транспортным протоколом с надежной доставкой данных [1 - 3]. Поэтому он имеет жесткие алгоритмы обнаружения ошибок, разработанные для обеспечения целостности передаваемых данных.

Для обеспечения надежной доставки применяется последовательная нумерация и подтверждение. С помощью последовательной нумерации определяется порядок следования данных в пакетах и выявляются пропущенные пакеты. Последовательная нумерация с подтверждением позволяет организовать надежную связь, которая называется *полным дуплексом* (full duplex). Каждая сторона соединения обеспечивает собственную нумерацию для другой стороны.

TCP – является *байтовым последовательным протоколом*. В отличие от пакетных последовательных протоколов, он присваивает последовательный номер каждому передаваемому байту пакета, а не каждому пакету в отдельности.

На рисунке 3.2 представлен формат заголовка TCP-пакета.

Offset	0	7 8	15 16	31
0	Source Port		Destination Port	
2	Sequence Number			
8	Acknowledgement Number			
1	Data Offset	Reserved	URG ACK PSH RST SYN FIN	Window
1	Checksum		Urgent Pointer	
2	Options			Padding

Рис. 3.2 Формат заголовка TCP-пакета

Назначение полей TCP пакета:

Номер порта отправителя – Source Port (16 бит) – содержит номер порта, с которого был отправлен пакет, когда это имеет значение (например, отправитель ожидает ответа). Если это поле не используется, оно заполняется нулями.

Номер порта назначения – Destination Port (16 бит) – содержит номер порта, на который будет доставлен пакет.

Порядковый номер – Sequence number (32 бита) – значение, присвоенное пакету TCP, определяющее номер стартового байта пакета, если не установлен бит SYN. Если установлен указанный бит, то порядковый номер является начальным порядковым номером (ISN) и первый байт данных равен ISN + 1.

Номер подтверждения – Acknowledgment Number (32 бита) – значение, отсылаемое принимающей станцией отправителю, подтверждающее прием переданного ранее пакета (пакетов). Оно задает следующий порядковый номер, который целевая станция ожидает получить при установленном бите ACK. При установленном соединении подтверждение отправляется всегда.

Смещение данных – Data Offset (4 бита) – задает длину заголовка TCP (количество 32-битовых слов в заголовке TCP).

Резервное поле – Reserved (6 бит) – зарезервировано.

Флаги управления:

URG – флаг срочности, применяется при посылке сообщения получателю, ожидающему приема экстренной информации;

ACK – флаг пакета, содержащего подтверждение получения;

PSH – флаг выталкивания, немедленная отсылка данных после считывания данных этого пакета;

RST – флаг переустановки соединения;

SYN – флаг синхронизации чисел последовательности;

FIN – флаг окончания передачи со стороны отправителя.

Окно – Window (16 бит) – содержит количество байт данных, которое отправитель данного сегмента может принять, отсчитанное от номера байта (*без подтверждения*), указанного в поле Acknowledgment Number.

Поле контрольной суммы – Checksum (16 бит) – представляет собой побитное дополнение 16-битной суммы 16-битных слов заголовка и данных дополненных нулевым байтом, если сегмент содержит нечетное число байт

заголовка и данных. При вычислении контрольной суммы поле контрольной суммы полагается равным нулю.

Указатель срочных данных – Urgent Pointer (16 бит) – содержит значение счетчика байтов, начиная с которого следуют данные повышенной срочности. Данное поле интерпретируется только в пакетах с установленным флагом URG;

Опции – Options – имеет переменную длину и содержит дополнительные параметры.

Заполнение – Padding – имеет переменную длину и используется для выравнивания заголовка по 32-битному слову нулевыми значениями.

3.2.2.Вариант максимального размера сегмента

Параметр *"максимальный размер сегмента"* (maximum segment size — MSS) применяется для объявления о наибольшем куске данных, который может быть принят и обработан системой [1]. Однако название несколько неточно. Обычно в TCP *сегмент* рассматривается как заголовок плюс данные. Однако *максимальный размер сегмента* определяется как:

Размер наибольшей датаграммы, которую можно принять — 40

Другими словами, MSS отражает наибольшую *полезную нагрузку* в приемнике при длине заголовков TCP и IP по 20 байт. Если имеются дополнительные параметры, их длину следует вычесть из общего размера. Следовательно, количество данных, которые можно переслать в сегменте, определяется как:

Заявленное значение MSS + 40 — (сумма длин заголовков TCP и IP)

Обычно партнеры обмениваются значениями MSS в начальных сообщениях SYN при открытии соединения. Если система не анонсирует величину максимального размера сегмента, используется значение по умолчанию в 536 байт.

Размер максимального сегмента кодируется 2-байтовой вводной частью со следующим далее 2-байтовым значением, т.е. наибольшая величина будет составлять 2^{16} - 1 (65 535 байт).

MSS накладывает жесткое ограничение на пересылаемые в TCP данные: приемник не сможет обработать большие значения. Однако отправитель использует сегменты *меньшего размера,* поскольку для соединения определяется еще размер MTU по пути следования.

3.2.3.Производительность

На производительность ресурсов влияют многие факторы, из которых основными являются память и полоса пропускания.

Полоса пропускания и задержки в используемой физической сети существенно ограничивают пропускную способность. Плохое качество пересылки данных приводит к большому объему отброшенных датаграмм, что

вызывает повторную пересылку и, как следствие, снижает эффективность полосы пропускания.

Приемная сторона должна обеспечить достаточное буферное пространство, позволяющее отправителю выполнять пересылку данных без пауз в работе. Это особенно важно для сетей с большими задержками, в которых между отправкой данных и получением АСК (а также при согласовании размера окна) проходит большой интервал времени.

Для поддержания устойчивого потока данных от источника принимающая сторона должна иметь окно размером не менее чем произведение полосы пропускания на задержку.

Например, если источник может отсылать данные со скоростью 10 000 байт/с, а на возврат АСК тратится 2 с, то на другой стороне нужно обеспечить приемное окно размером не менее 20 000 байт, иначе поток данных не будет непрерывным. Приемный буфер в 10 000 байт вполовину снизит пропускную способность.

Еще одним важным фактором для производительности является способность хоста реагировать на высокоприоритетные события и быстро выполнять *контекстное переключение,* т.е. завершать одни операции и переключаться на другие. Хост может интерактивно поддерживать множество локальных пользователей, пакетные фоновые процессы и десятки одновременных коммуникационных соединений. Контекстное переключение позволяет обслуживать все эти операции, скрывая нагрузки на систему. Реализации, в которых проведена интеграция TCP/IP с ядром операционной системы, могут существенно снизить нагрузки от использования контекстного переключения.

Ресурсы центрального процессора компьютера необходимы для операций по обработке заголовков TCP. Если процессор не может быстро вычислять контрольные суммы, это приводит к снижению скорости пересылки данных по сети.

Кроме того, разработчикам следует обращать внимание на упрощение конфигурирования параметров TCP, чтобы сетевой администратор мог настроить их в соответствии со своими локальными требованиями. Например, возможность настройки размера буфера на полосу пропускания и задержку сети существенно улучшит производительность. К сожалению, многие реализации не уделяют этому вопросу должного внимания и жестко программируют коммуникационные параметры.

Предположим, что сетевое окружение совершенно: имеются достаточные ресурсы и контекстное переключение выполняется быстрее, чем ковбои выхватывают свои револьверы. Будет ли получена прекрасная производительность?

Не всегда. Имеет значение и качество разработки программного обеспечения TCP. За долгие годы в различных реализациях TCP были диагностированы и решены многие проблемы производительности. Можно считать, что наилучшим будет программное обеспечение, соответствующее

документу RFC 1122, в котором определены требования к коммуникационному уровню хостов Интернета.

Не менее важно исключение *синдрома "бестолкового окна"* и применение алгоритмов Джекобсона, Керна и Партриджа (эти интересные алгоритмы будут рассмотрены ниже).

Разработчики программного обеспечения могут получить существенные выгоды, создавая программы, исключающие ненужные пересылки небольших объемов данных и имеющие встроенные таймеры для освобождения сетевых ресурсов, не используемых в текущий момент.

Алгоритмы повышения производительности

Переходя к знакомству с достаточно сложной частью TCP, мы рассмотрим механизмы повышения производительности и решения проблем снижений пропускной способности. В этом разделе обсуждаются нижеследующие проблемы [1].

- *Медленный старт* (slow start) мешает использованию большой доли сетевого трафика для нового сеанса, что может привести к непроизводительным потерям.
- Излечение от *синдрома "бестолкового окна"* (silly window syndrome) предохраняет плохо разработанные приложения от перегрузки сети сообщениями.
- *Задержанный АСК* (delayed АСК) снижает перегрузку посредством сокращения количества независимых сообщений подтверждения пересылки данных.
- *Вычисляемый тайм-аут повторной пересылки* (computing retransmission timeout) основывается на согласовании реального времени сеанса, уменьшая объем ненужных повторных пересылок, но при этом не вызывает больших задержек для реально необходимых обменов данными.
- *Торможение пересылки TCP* при перегрузках в сети позволяет маршрутизаторам вернуться в исходный режим и совместно использовать сетевые ресурсы для всех сеансов.
- Отправка *дублированных АСК* (duplicate АСК) при получении сегмента вне последовательности отправки позволяет партнерам выполнить повторную пересылку до наступления тайм-аута.

3.2.4.Медленный старт

Если дома одновременно включить все бытовые электроприборы, произойдет перегрузка электрической сети. В компьютерных сетях *медленный старт* не позволяет перегореть сетевым предохранителям. Новое соединение, мгновенно запускающее пересылку большого объема данных в уже и так нагруженной сети, может привести к проблемам. Идея медленного старта заключается в обеспечении новому соединению успешного запуска с медленным увеличением скорости пересылки данных в соответствии с

реальной нагрузкой на сеть. Отправитель ограничивается размером нагрузочного окна, а не большим по размеру приемным окном.

Нагрузочное окно (congestion window) начинается с размера в 1 сегмент. Для каждого сегмента с успешно полученным АСК размер нагрузочного окна увеличивается на 1 сегмент, пока оно остается меньше, чем приемное окно. Если сеть не перегружена, нагрузочное окно постепенно достигнет размера приемного окна. При нормальном состоянии пересылки размеры этих окон будут совпадать.

Отметим, что медленный старт — не такой уж и медленный. После первого АСК размер нагрузочного окна равен 2-м сегментам, а после успешного получения АСК для двух сегментов размер может увеличиться до 8 сегментов. Другими словами, размер окна увеличивается экспоненциально.

Предположим, что вместо получения АСК возникла ситуация тайм-аута. Поведение нагрузочного окна в таком случае рассматривается ниже.

3.2.5.Синдром бестолкового окна

В первых реализациях TCP/IP разработчики столкнулись с феноменом *синдрома «бестолкового окна»*, который проявлялся достаточно часто [1]. Для понимания происходящих событий рассмотрим следующий сценарий, приводящий к нежелательным последствиям, но вполне возможный:

1. Передающее приложение быстро отсылает данные.
2. Принимающее приложение читает из входного буфера по 1 байту данных (т.е. медленно).
3. Входной буфер после чтения быстро заполняется.
4. Принимающее приложение читает 1 байт, и TCP отправляет АСК, означающий "Я имею свободное место для 1 байта данных".
5. Передающее приложение отправляет по сети пакет TCP из 1 байта.
6. Принимающий TCP посылает АСК означающий "Спасибо. Я получил пакет и не имею больше свободного места".
7. Принимающее приложение опять читает 1 байт и отправляет АСК и весь процесс повторяется.

Медленное принимающее приложение долго ожидает поступления данных и постоянно подталкивает полученную информацию к левому краю окна, выполняя совершенно бесполезную операцию, порождающую дополнительный трафик в сети.

Реальные ситуации, конечно, не столь экстремальны. Быстрый отправитель и медленный получатель будут обмениваться небольшими (относительно максимального размера сегмента) кусками данных и переключаться по почти заполненному приемному окну.

Решить эту проблему не сложно. Как только приемное окно сокращается на длину меньшую, чем данный целевой размер, TCP начинает обманывать отправителя. В этой ситуации TCP не должен указывать отправителю на дополнительное пространство в окне, когда принимающее приложение читает данные из буфера небольшими порциями. Вместо этого нужно держать освобождающиеся ресурсы в секрете от отправителя до тех пор, пока их не

будет достаточное количество. Рекомендуется размер в один сегмент, кроме случаев, когда весь входной буфер хранит единственный сегмент (в последнем случае используется размер, равный половине буфера). Целевой размер, о котором должен сообщать TCP, можно выразить как:

minimum (1/2 входного буфера, Максимальный размер сегмента)

TCP начинает обманывать, когда размер окна станет меньше этого размера, и скажет правду, когда размер окна не меньше, чем получаемая по формуле величина. Отметим, что для отправителя нет никакого ущерба, поскольку принимающее приложение все равно не смогло бы обработать большую часть данных, которых оно ожидает.

Предложенное решение легко проверить в рассмотренном выше случае с выводом АСК для каждого из полученных байтов. Этот же способ пригоден и для случая, когда входной буфер может хранить несколько сегментов (как часто бывает на практике). Быстрый отправитель заполнит входной буфер, но приемник укажет, что не имеет свободного места для размещения информации, и не откроет этот ресурс, пока его размер не достигнет целого сегмента.

3.2.6.Алгоритм Нейгла

Отправитель должен независимо от получателя исключить пересылку очень коротких сегментов, аккумулируя данные перед отправлением [1]. Алгоритм Нейгла (*Nagle*) реализует очень простую идею, позволяющую снизить количество пересылаемых по сети коротких датаграмм.

Алгоритм рекомендует задержать пересылку данных (и их выталкивание) на время ожидания АСК от ранее переданных данных. Аккумулируемые данные пересылаются после получения АСК на ранее отправленную порцию информации, либо после получения для отправки данных в размере полного сегмента или по завершении тайм-аута. Этот алгоритм не следует применять для приложений реального времени, которые должны отправлять данные как можно быстрее.

3.2.7.Задержанный АСК

Еще одним механизмом повышения производительности является способ задержки АСК [1]. Сокращение числа АСК снижает полосу пропускания, которую можно использовать для пересылки другого трафика. Если партнер по TCP чуть задержит отправку АСК, то:

- можно подтвердить прием нескольких сегментов одним АСК;
- принимающее приложение способно получить некоторый объем данных в пределах интервала тайм-аута, т.е. в АСК может попасть выходной заголовок, и не потребуется формирование отдельного сообщения.

С целью исключения задержек при пересылке потока полноразмерных сегментов (например, при обмене файлами) АСК должен отсылаться, по крайней мере, для каждого второго полного сегмента.

Многие реализации используют тайм-аут в 200 мс. Но задержанный АСК не снижает скорость обмена. При поступлении короткого сегмента во входном буфере остается еще достаточно свободного места для получения новых данных, а отправитель может продолжить пересылку (кроме того, повторная пересылка обычно выполняется гораздо медленнее). Если же поступает целый сегмент, нужно в ту же секунду ответить на него сообщением АСК.

3.2.8.Тайм-аут повторной пересылки

После отправки сегмента TCP устанавливает таймер и отслеживает поступление АСК [1]. Если АСК не получен в течение периода тайм-аута, TCP выполняет повторную пересылку сегмента (ретрансляцию). Однако каким должен быть период тайм-аута?

Если он слишком короткий, отправитель заполнит сеть пересылкой ненужных сегментов, дублирующих уже отправленную информацию. Слишком же большой тайм-аут будет препятствовать быстрому исправлению действительно разрушенных при пересылке сегментов, что снизит пропускную способность.

Как выбрать правильный промежуток для тайм-аута? Значение, пригодное для высокоскоростной локальной сети, не подойдет для удаленного соединения с множеством попаданий. Значит, принцип "одно значение для любых условий" явно непригоден. Более того, даже для существующего конкретного соединения могут измениться сетевые условия, а задержки — увеличиться или снизиться.

Алгоритмы Джекобсона, Керна и Партриджа (описанные в статьях Congestion Avoidance and Control**,** Van Jacobson, и Improving Round-Trip Time Estimates in Reliable Transport Protocols, Karn и Partridge) позволяют адаптировать TCP к изменению сетевых условий. Эти алгоритмы рекомендованы к использованию в новых реализациях. Мы кратко рассмотрим их ниже.

Здравый смысл подсказывает, что наилучшей основой оценки правильного времени тайм-аута для конкретного соединения может быть отслеживание *времени цикла* (round-trip time) как промежутка между отправкой данных и получением подтверждения об их приеме.

Хорошие решения для следующих величин можно получить на основе элементарных статистических сведений, которые помогут вычислить время тайм-аута. Однако не нужно полагаться на усредненные величины, поскольку более половины оценок будет больше, чем среднестатистическая величина. Рассмотрев пару отклонений, можно получить более правильные оценки, учитывающие нормальное распределение и снижающие слишком долгое время ожидания повторной пересылки.

Нет необходимости в большом объеме вычислений для получения формальных математических оценок отклонений. Можно использовать достаточно грубые оценки на основе абсолютной величины разницы между последним значением и среднестатистической оценкой:

Последнее отклонение = |*Последний цикл* - *Средняя величина*|

Для вычисления правильного значения тайм-аута нужно учитывать еще один фактор — изменение времени цикла из-за текущих сетевых условий. Происходившее в сети в последнюю минуту более важно, чем то, что было час назад.

Предположим, что вычисляется среднее значение цикла для очень длинного по времени сеанса. Пусть вначале сеть была мало загружена, и мы определили 1000 небольших значений, однако, далее произошло увеличение трафика с существенным увеличением времени задержки.

Например, если 1000 значений дали среднестатистическую величину в 170 единиц, но далее были замерены 50 значений со средним в 282, то текущее среднее будет:

$$170 \times 1000/1050 + 282 \times 50/1050 = 175$$

Более резонной будет величина сглаженного времени цикла (*Smoothed Round-Trip Time — SRTT*), которая учитывает приоритет более поздних значений:

Новое SRTT = (1 — a) x *(старое SRTT) + a* x *Последнее значение цикла*

Значение *a* находится между 0 и 1. Увеличение *a* приводит к большему влиянию текущего времени цикла на сглаженное среднее значение. Поскольку компьютеры быстро могут выполнять деление на степени числа 2, сдвигая двоичные числа направо, для *a* всегда выбирается значение 1/2 в степени (обычно 1/8), поэтому:

Новое SRTT - 7/8 x *старое SRTT + 1/8* x *Последнее время цикла*

Теперь возникает вопрос о выборе значения для тайм-аута повторной пересылки. Анализ величин времени цикла показывает существенное отклонение этих значений от текущей средней величины. Имеет смысл установить границу для величины отклонений (девиаций). Хорошие величины для тайм-аута повторной пересылки (в стандартах RFC эту величину именуют Retransmission TimeOut — RTO) дает следующая формула с ограничением сглаженного отклонения (SDEV):

T = Тайм-аут повторной пересылки = SRTT + 2 x *SDEV*

Именно эта формула рекомендована в RFC 1122. Однако некоторые реализации используют другую:

T = SRTT + 4 x SDEV

Для вычисления *SDEV* сначала определяют абсолютное значение текущего отклонения:

DEV = | *Последнее время цикла — старое SRTT*

Затем используют формулу сглаживания, чтобы учесть последнее значение:

Новое SDEV = 3/4 x *старое SDEV + 1/4* x *DEV*

Начальный тайм-аут = 3 с
Начальный SRTT = 0
Начальный SDEV = 1,5 с

Ван Джекобсон определил быстрый алгоритм, который очень эффективно вычисляет тайм-аут повторной пересылки данных.

3.2.9.Вычисления после повторной отправки

В представленных выше формулах используется значение времени цикла как интервала между отправкой сегмента и получением подтверждения его приема [1]. Однако предположим, что в течение периода тайм-аута подтверждение не получено и данные должны быть оправлены повторно.

Алгоритм Керна предполагает, что в этом случае не следует изменять время цикла. Текущее сглаженное значение времени цикла и *сглаженное отклонение* сохраняют свои значения, пока не будет получено подтверждение на пересылку некоторого сегмента без его повторной отправки. С этого момента возобновляются вычисления на основе сохраненных величин и новых замеров.

Но что происходит до получения подтверждения? После повторной пересылки поведение TCP радикально меняется в основном из-за потери данных от перегрузки в сети. Следовательно, реакцией на повторную отправку данных будет:

- Снижение скорости повторной пересылки.
- Борьба с перегрузкой сети с помощью сокращения общего трафика

3.2.10.Экспоненциальное торможение

После повторной пересылки удваивается интервал тайм-аута. Однако что произойдет при повторном переполнении таймера? Данные будут оправлены еще раз, а период повторной пересылки снова удвоится. Этот процесс называется *экспоненциальным торможением* (exponential backoff) [1].

Если продолжает проявляться неисправность сети, период тайм-аута будет удваиваться до достижения предустановленного максимального значения (обычно — 1 мин). После тайм-аута может быть отправлен только один сегмент. Тайм-аут наступает и при превышении заранее установленного значения для количества пересылок данных без получения АСК.

3.2.11.Снижение перегрузок за счет уменьшения пересылаемых по сети данных

Сокращение объема пересылаемых данных несколько сложнее, чем рассмотренные выше механизмы [1]. Оно начинает работать, как и уже упомянутый медленный старт. Но, поскольку устанавливается граница для уровня трафика, который может в начальный момент привести к проблемам, будет реально замедляться скорость обмена вследствие увеличения размера нагрузочного окна по одному сегменту. Нужно установить значения границы для реального сокращения скорости отправки. Сначала вычисляется граница Опасности (danger threshold):

Граница = 1/2 minimum (текущее нагрузочное окно, приемное окно партнера)

Если полученная величина будет более двух сегментов, ее используют как границу. Иначе размер границы устанавливается равным двум сегментам. Полный алгоритм восстановления требует:

- Установить размер нагрузочного окна в один сегмент.
- Для каждого полученного АСК увеличивать размер нагрузочного окна на один сегмент, пока не будет достигнута граница (что очень напоминает механизм медленного старта).
- После этого с каждым полученным АСК к нагрузочному окну добавлять меньшее значение, которое выбирается на основе скорости увеличения по одному сегменту для времени цикла (увеличение вычисляется как *MSS/N*, где *N* — размер нагрузочного окна в сегментах).

Сценарий для идеального варианта может упрощенно представить работу механизма восстановления. Предположим, что приемное окно партнера (и текущее нагрузочное окно) имело до выявления тайм-аута размер в 8 сегментов, а граница определена в 4 сегмента. Если принимающее приложение мгновенно читает данные из буфера, размер приемного окна останется равным 8 сегментам.

1. Отправляется 1 сегмент (нагрузочное окно = 1 сегмент).
2. Получен АСК — отправляется 2 сегмента.
3. Получен АСК для 2 сегментов — посылается 4 сегмента, (достигается граница).
4. Получен АСК для 4 сегментов. Посылается 5 сегментов.
5. Получен АСК для 5 сегментов. Посылается 6 сегментов.
6. Получен АСК для 6 сегментов. Посылается 7 сегментов.
7. Получен АСК для 7 сегментов. Посылается 8 сегментов (нагрузочное окно по размеру снова стало равно приемному окну).

Поскольку во время повторной пересылки по тайм-ауту требуется подтверждение приема всех отправленных данных, процесс продолжается до достижения нагрузочным окном размера приемного окна. Размер окна

увеличивается экспоненциально, удваиваясь во время периода медленного старта, а по достижении границы увеличение происходит по линейному закону.

3.2.12.Дублированные АСК

В некоторых реализациях применяется необязательная возможность — так называемая *быстрая повторная пересылка* (fast retransmit) — с целью ускорить повторную отправку данных при определенных условиях [1]. Ее основная идея связана с отправкой получателем дополнительных АСК, указывающих на пробел в принятых данных.

Принимая сегмент, поступивший не по порядку, получатель отсылает обратно АСК, указывающий на первый байт *потерянных* данных

Отправитель не выполняет мгновенной повторной пересылки данных, поскольку IP может и в нормальном режиме доставлять данные получателю без последовательности отправки. Но когда получено несколько дополнительных АСК на дублирование данных (например, три), то отсутствующий сегмент будет отправлен, не дожидаясь завершения тайм-аута.

Отметим, что каждый дублирующий АСК указывает на получение сегмента данных. Несколько дублирующих АСК позволяют понять, что сеть способна доставлять достаточный объем данных, следовательно, не слишком сильно нагружена. Как часть общего алгоритма выполняется небольшое сокращение размера нагрузочного окна при реальном увеличении сетевого трафика. В данном случае процесс радикального изменения размера при восстановлении работы не применяется.

3.2.13.Барьеры для производительности

TCP доказал свою гибкость, работая в сетях со скоростью обмена в сотни или миллионы бит за секунду. Этот протокол позволил достичь хороших результатов в современных локальных сетях с топологиями Ethernet, Token-Ring и Fiber Distributed Data Interlace (FDDI), а также для низкоскоростных линий связи или соединений на дальние расстояния (подобных спутниковым связям).

TCP разработан так, чтобы реагировать на экстремальные условия, например на перегрузки в сети. Однако в текущей версии протокола имеются особенности, ограничивающие производительность в перспективных технологиях, которые предлагают полосу пропускания в сотни и тысячи мегабайт [1]. Чтобы понять возникающие проблемы, рассмотрим простой (хотя и нереалистичный) пример.

Предположим, что при перемещении файла между двумя системами необходимо выполнять обмен непрерывным потоком настолько эффективно, насколько это возможно. Допустим, что:

- максимальный размер сегмента приемника — 1 Кбайт;
- приемное окно — 4 Кбайт;
- полоса пропускания позволяет пересылать по два сегмента за 1 с.;

- принимающее приложение поглощает данные по мере их поступления;
- сообщения АСК прибывают через 2 с.

Отправитель способен посылать данные непрерывно. Ведь когда выделенный для окна объем будет заполнен, прибывает АСК, разрешающий отправку другого сегмента:

SEND SEGMENT 1.
SEND SEGMENT 2.
SEND SEGMENT 3.
SEND SEGMENT 4.

Через 2 с:
RECEIVE ACK OF SEGMENT 1, CAN SEND SEGMENT 5.
RECEIVE ACK OF SEGMENT 2, CAN SEND SEGMENT 6.
RECEIVE ACK OF SEGMENT 3, CAN SEND SEGMENT 7.
RECEIVE ACK OF SEGMENT 4, CAN SEND SEGMENT 8.

Еще через 2 с:
RECEIVE ACK OF SEGMENT 5, CAN SEND SEGMENT 9.

Если приемное окно было только в 2 Кбайт, отправитель будет вынужден ждать одну секунду из каждых двух перед отправкой следующих данных. Фактически, чтобы удержать непрерывный поток данных, приемное окно должно иметь размер не менее:

Окно = Полоса пропускания x *Время цикла*

Хотя пример несколько преувеличен (для обеспечения более простых чисел), малое окно может привести к проблемам при соединениях через спутниковые связи с большой задержкой.

Теперь рассмотрим, что происходит с высокоскоростными соединениями. Например, если полоса пропускания и скорость пересылки измеряются 10 млн. бит в секунду, но время цикла составляет 100 мс (1/10 секунды), то для непрерывного потока приемное окно должно хранить, по крайней мере, 1 000 000 бит, т.е. 125 000 байт. Но наибольшее число, которое можно записать в поле заголовка для приемного окна TCP, равно 65 536.

Другая проблема возникает при высоких скоростях обмена, поскольку порядковые номера очень быстро закончатся. Если по соединению можно будет пересылать данные со скоростью 4 Гбайт/с, то порядковые номера должны обновляться в течение каждой секунды. Не будет возможности различить старые дублирующие датаграммы, которые были отсрочены более чем на секунду при перемещении по Интернету, от свежих новых данных.

Сейчас активно проводятся новые исследования для улучшения TCP/IP и устранения упомянутых выше препятствий.

3.2.14.Функции TCP

Ниже перечислены основные из функций TCP:

- Связывание портов с соединениями.
- Инициализация соединений посредством трехшагового подтверждения.
- Выполнение медленного старта, исключающего перегрузку сети.
- Сегментация данных при пересылке.
- Нумерация данных.
- Обработка поступающих дублированных сегментов.
- Вычисление контрольных сумм.
- Регулирование потока данных через приемное окно и окно отправки.
- Завершение соединения установленным способом.
- Прерывание соединения.
- Пересылка срочных данных.
- Положительное подтверждение повторной пересылки.
- Вычисление тайм-аута повторной пересылки.
- Снижение обратного трафика при перегрузке сети.
- Сигнализация поступления сегментов не по порядку.
- Зондирование закрытия приемного окна.

3.3.Протокол SCTP

Протокол передачи с управлением потоком (Stream Control Transmission Protocol, SCTP) − это надежный транспортный протокол, который обеспечивает стабильную, упорядоченную (с сохранением порядка следования пакетов) передачу данных между двумя конечными точками (подобно TCP) [5-7]. Кроме того, протокол обеспечивает сохранение границ отдельных сообщений (подобно UDP). Однако в отличие от протоколов TCP и UDP протокол SCTP имеет дополнительные преимущества, такие как поддержка множественной адресации (multihoming) и многопоточности (multi-streaming) - каждая из этих возможностей увеличивает доступность узла передачи данных. В этой статье мы познакомимся с основными характеристиками протокола SCTP ядра Linux® 2.6 и рассмотрим исходный текст программ сервера и клиента, демонстрирующий возможности протокола по многопоточной передаче данных.

Протокол SCTP представляет собой надежный универсальный протокол транспортного уровня для сетей IP. Несмотря на то, что протокол изначально разрабатывался для передачи телефонных сигналов (RFC 2960), SCTP имеет дополнительные преимущества − он лишен некоторых ограничений протокола TCP, обладая при этом возможностями протокола UDP. Протокол SCTP предоставляет возможности, обеспечивающие высокую доступность, повышенную надежность и улучшенную безопасность сокетов.

3.3.1. Заголовок пакета SCTP

SCTP пакеты имеют более простую структуру, чем пакеты TCP (см. рис. 3.3). Каждый пакет состоит из двух основных разделов [6, 7]:

1. Общий заголовок занимает первые 12 байт.
2. Блоки данных занимают оставшуюся часть пакета.

Пакет делится на блоки. Каждый блок имеет идентификатор типа, размером один байт. RFC 4960 определяет список видов блоков, всего на данный момент определено 15 типов. Остальная часть блока состоит из двух байтов длиной (максимальный размер 65535 байт) и данных. Если длина блока не кратно 4, то оставшееся пространство (блок выравнивается до кратности 4) заполняется нулями.

Биты	**Биты 0–7**	**8–15**	**16–23**	**24–31**
+0	Порт источника		Порт назначения	
32	Тэг проверки			
64	Контрольная сумма			
96	Тип 1 блока	Флаги 1 блока	Длина 1 блока	
128	Данные 1 блока			
…	…			
…	Тип N блока	Флаги N блока	Длина N блока	
…	Данные N блока			

Рис. 3.3 Формат заголовка SCTP-пакета

3.3.2. Множественная адресация

По сравнению с протоколом TCP поддержка протоколом SCTP *множественной адресации* обеспечивает приложениям повышенную готовность. Хост, подключенный к нескольким сетевым интерфейсам и потому имеющий несколько IP адресов, называется multi-homed хост. В протоколе TCP *соединением* называется канал между двумя конечными точками (в данном случае сокет между интерфейсами двух хостов). Протокол SCTP вводит понятие *ассоциации*, которая устанавливается между двумя хостами и в рамках которой возможна организация взаимодействия между несколькими интерфейсами каждого хоста.

На рисунке 3.4 показано отличие соединения протокола TCP и ассоциации протокола SCTP.

В верхней части рисунка показано соединение протокола TCP. Каждый хост имеет один сетевой интерфейс, соединение устанавливается между одним интерфейсом на клиенте и одним интерфейсом на сервере. Установленное соединение привязано к конкретному интерфейсу.

В нижней части рисунка представлена архитектура, в которой каждый хост имеет два сетевых интерфейса. Обеспечивается два маршрута по независимым сетям: один от интерфейса C0 к интерфейсу S0, другой — от

интерфейса C1 к интерфейсу S1. В протоколе SCTP два этих маршрута объединяются в ассоциацию.

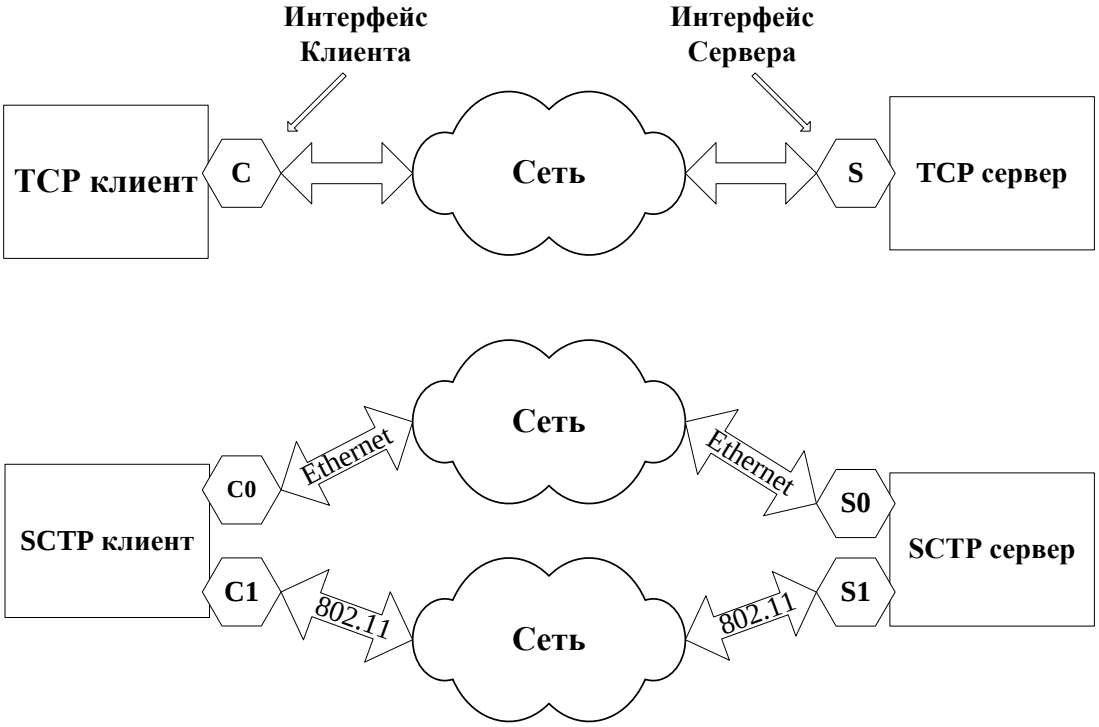


Рис. 3.4 Отличие между соединением протокола TCP и ассоциацией протокола SCTP

Протокол SCTP отслеживает состояние маршрутов в ассоциации с помощью встроенного механизма контрольных сообщений; при нарушении маршрута передача данных продолжается по альтернативному маршруту. При этом приложению даже не обязательно знать о фактах нарушения и восстановления маршрута.

Переключение на резервный канал также может обеспечивать непрерывность связи для сетевых приложений. Для примера рассмотрим ноутбук, который имеет беспроводный интерфейс 802.11 и интерфейс Ethernet. Пока ноутбук подключен к док-станции, предпочтительнее использовать более скоростной интерфейс Ethernet (в протоколе SCTP используется термин *основной адрес*); при нарушении этого соединения (в случае отключения от док-станции) будет использоваться соединение по беспроводному интерфейсу. При повторном подключении к док-станции будет обнаружено соединение по интерфейсу Ethernet, через который будет продолжен обмен данными. Таким образом, протокол SCTP реализует эффективный механизм, обеспечивающий высокую готовность и повышенную надежность.

3.3.3. Многопотоковая передача данных

В некоторой степени ассоциация протокола SCTP похожа на соединение протокола TCP [6, 7]. Отличие состоит в том, что протокол SCTP поддерживает несколько потоков в рамках одной ассоциации. Все потоки являются независимыми, но принадлежат одной ассоциации (см. рис. 3.5).

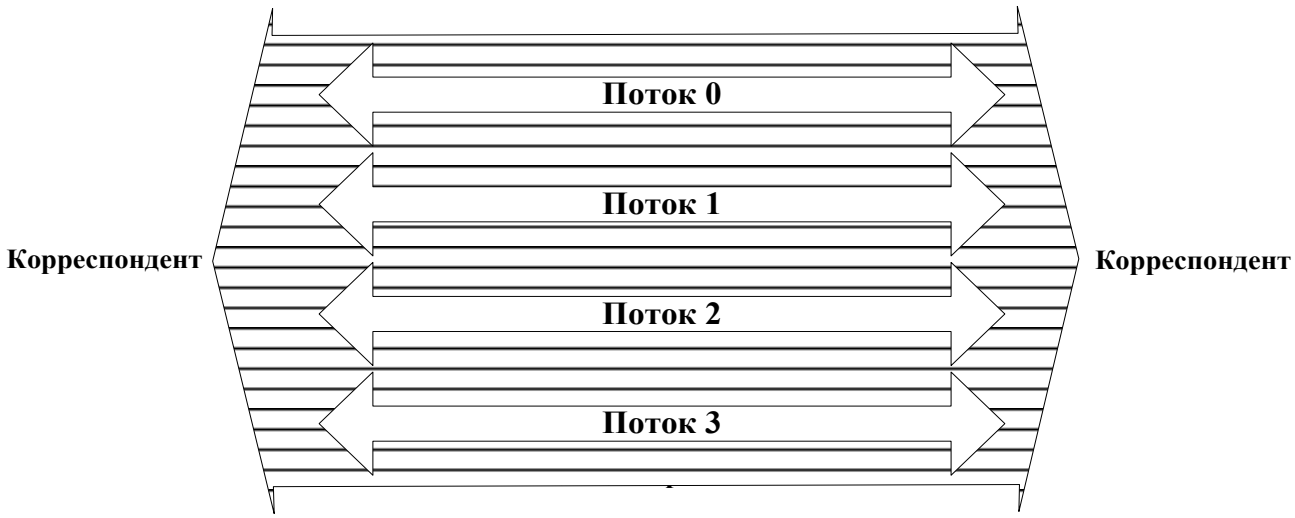


Рис. 3.5 Взаимосвязь между ассоциацией протокола SCTP и потоками

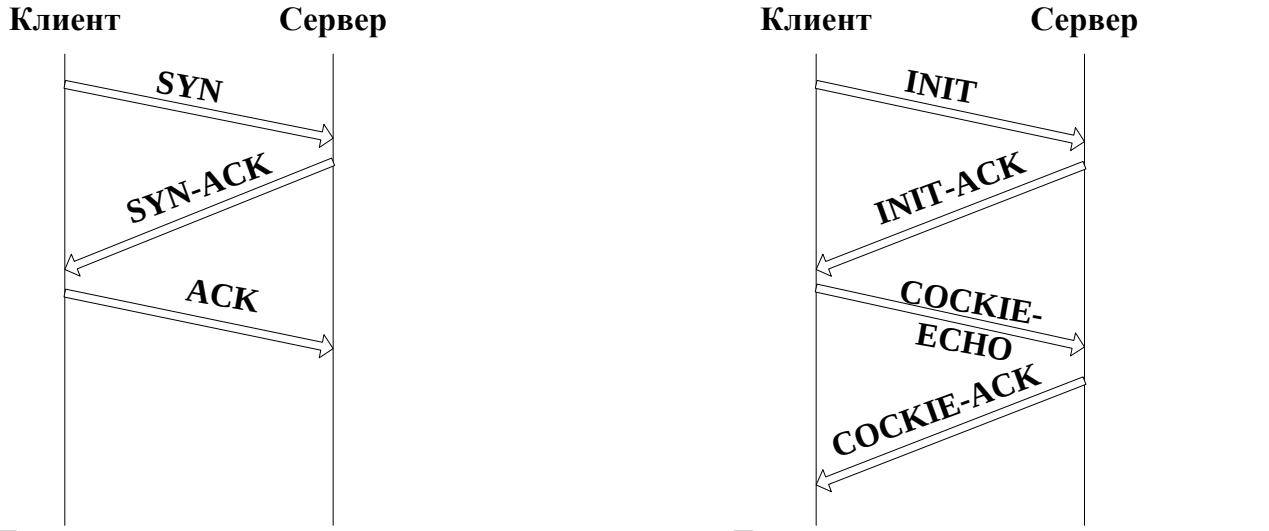
Каждому потоку ассоциации присваивается номер, который включается в передающиеся пакеты SCTP. Важность многопотоковой передачи обусловлена тем, что блокировка какого-либо потока (например, из-за ожидания повторной передачи при потере пакета) не оказывает влияния на другие потоки в ассоциации. В общем случае данная проблема получила название *блокировка головы очереди (head-of-line blocking*). Протокол TCP уязвим для подобных блокировок.

Каким образом множество потоков обеспечивают лучшую оперативность при передаче данных? Например, в протоколе HTTP данные и служебная информация передаются по одному и тому же сокету. Web-клиент запрашивает файл, и сервер посылает файл назад к клиенту по тому же самому соединению. Многопотоковый HTTP-сервер сможет обеспечить более быструю передачу, так как множество запросов может обслуживаться по независимым потокам одной ассоциации. Такая возможность позволяет распараллелить ответы сервера. Это если и не повысит скорость отображения страницы, то позволит обеспечить ее лучшее восприятие благодаря одновременной загрузке кода HTML и графических изображений.

Многопотоковая передача − это важнейшая особенность протокола SCTP, особенно в том, что касается одновременной передачи данных и служебной информации в рамках протокола. В протоколе TCP данные и служебная информация передаются по одному соединению. Это может стать причиной проблем, так как служебные пакеты из-за передачи данных будут передаваться с задержкой. Если служебные пакеты и пакеты данных передаются по независимым потокам, то служебная информация будет обрабатываться своевременно, что, в свою очередь, приведет к лучшему использованию доступных ресурсов.

3.3.4. Безопасность устанавливаемого подключения

Создание нового подключения в протоколах TCP и SCTP происходит при помощи механизма подтверждения (квитирования) пакетов [6, 7]. В протоколе TCP данная процедура получила название *трехэтапное квитирование (three-way handshake*). Клиент посылает пакет `SYN` (сокр. *Synchronize*). Сервер отвечает пакетом `SYN-ACK` (*Synchronize-Acknowledge*). Клиент подтверждает прием пакета `SYN-ACK` пакетом `ACK`. На этом процедура установления соединения (показана на рис. 3.6) завершается.



Процедура установления соединения (треэтапное квитирование) в протоколе TCP

Процедура установления соединения (четерехэтапное квитирование) в протоколе SCTP

Рис. 3.6 Обмен пакетами при установлении соединения по протоколам TCP и SCTP

Протокол TCP имеет потенциальную уязвимость, обусловленную тем, что нарушитель, установив фальшивый IP-адрес отправителя, может послать серверу множество пакетов `SYN`. При получении пакета `SYN` сервер выделяет часть своих ресурсов для установления нового соединения. Обработка множества пакетов `SYN` рано или поздно займет все ресурсы с невозможностью обработки новых запросов сервером. Такая атака получила название «отказ в обслуживании» (*Denial of Service* (DoS)).

Протокол SCTP защищен от подобных атак с помощью механизма четырехэтапного квитирования (four-way handshake) и вводом маркера (cookie). По протоколу SCTP клиент начинает процедуру установления соединения посылкой пакета `INIT`. В ответ сервер посылает пакет `INIT-ACK`, который содержит маркер (уникальный ключ, идентифицирующий новое соединение). Затем клиент отвечает посылкой пакета `COOKIE-ECHO`, в котором содержится маркер, посланный сервером. Только после этого сервер выделяет свои ресурсы новому подключению и подтверждает это отправлением клиенту пакета `COOKIE-ACK`.

Для решения проблемы задержки пересылки данных при выполнении процедуры четырехэтапного квитирования в протоколе SCTP допускается включение данных в пакеты `COOKIE-ECHO` и `COOKIE-ACK`.

3.3.5. Формирование кадров сообщения

При формировании кадров сообщения обеспечивается сохранение границ сообщения в том виде, в котором оно передается сокету; это означает, что если клиент посылает серверу 100 байт, за которыми следуют 50 байт, то сервер воспринимает 100 байт и 50 байт за две операции чтения [6, 7]. Точно так же функционирует протокол UDP, это является особенностью протоколов, ориентированных на работу с сообщениями.

В противоположность им протокол TCP обрабатывает неструктурированный поток байт. Если не использовать процедуру формирования кадров сообщения, то узел сети может получать данные по размеру больше или меньше отправленных. Такой режим функционирования требует, чтобы для протоколов, ориентированных на работу с сообщениями и функционирующих поверх протокола TCP, на прикладном уровне был предоставлен специальный буфер данных и выполнялась процедура формирования кадров сообщений (что потенциально является сложной задачей).

Протокол SCTP обеспечивает формирование кадров при передаче данных. Когда узел выполняет запись в сокет, его корреспондент с гарантией получает блок данных того же размера (см. рис. 3.7).

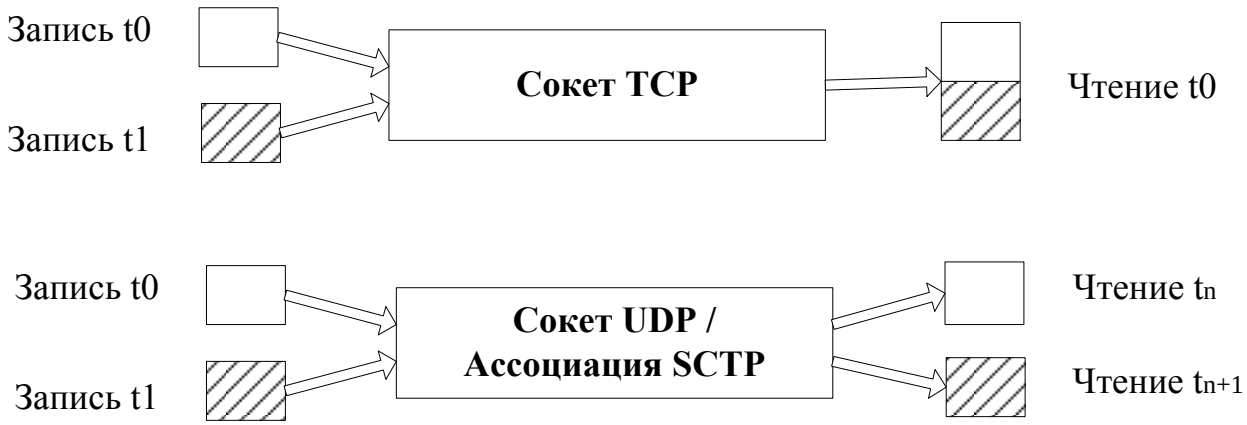


Рис. 3.7 Формирование кадров сообщений по протоколу UDP/SCTP и по протоколу, обрабатывающему неструктурированный поток байт

При передаче потоковых данных (аудио- и видеоданных) процедура формирования кадров необязательна.

3.3.6. Настраиваемая неупорядоченная передача данных

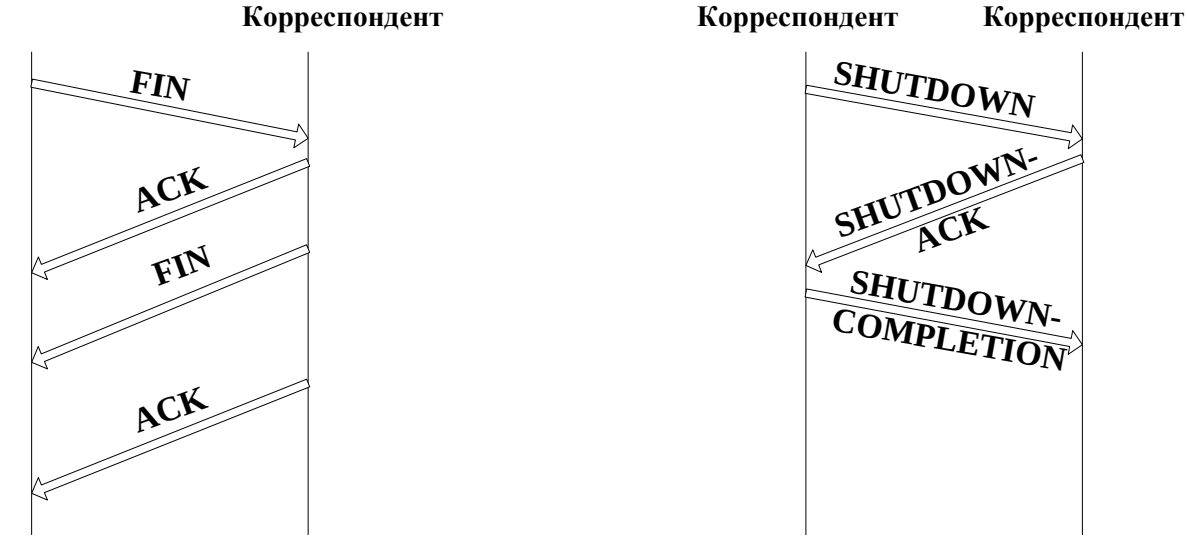
Сообщения по протоколу SCTP передаются с высокой степенью надежности, но необязательно в нужном порядке [6, 7]. Протокол TCP гарантирует, что данные будут получены именно в том порядке, в котором они

отправлялись (это хорошо, учитывая, что TCP – это потоковый протокол). Протокол UDP не гарантирует упорядоченную доставку. При необходимости вы можете настроить потоки в протоколе SCTP так, чтобы они принимали неупорядоченные сообщения.

Эта особенность может быть востребована в протоколах, ориентированных на работу с сообщениями, так как запросы в них независимы и последовательность поступления данных не очень важна. Кроме того, вы можете настроить использование неупорядоченной передачи по номеру потока в ассоциации, т.е. принимая сообщения по потокам.

3.3.7. Поэтапное завершение передачи данных

В отличие от протокола UDP, функционирование которого не предполагает установления соединения, протоколы TCP и SCTP являются протоколами с установлением соединения [6, 7]. Оба эти протокола требуют выполнения процедуры установления и разрыва соединения между корреспондентами. Рассмотрим отличия между процедурой закрытия сокетов протокола SCTP и процедурой частичного закрытия (*half-close*) протокола TCP. На рисунке 3.8 показана последовательность разрыва соединения в протоколах TCP и SCTP.



Последовательность разрыва соединения в протоколе TCP

Последовательность разрыва соединения в протоколе SCTP

Рис. 3.8 Последовательность разрыва соединения по протоколам TCP и SCTP

В протоколе TCP возможна ситуация, когда узел закрывает у себя сокет (выполняя посылку пакета `FIN`), но продолжает принимать данные. Пакет `FIN` указывает корреспонденту на отсутствие данных для передачи, однако до тех пор, пока корреспондент не закроет свой сокет, он может продолжать передавать данные. Состояние частичного закрытия используется приложениями крайне редко, поэтому разработчики протокола SCTP посчитали нужным заменить его последовательностью сообщений для разрыва

существующей ассоциации. Когда узел закрывает свой сокет (посылает сообщение `SHUTDOWN`), оба корреспондента должны прекратить передачу данных, при этом разрешается лишь обмен пакетами, подтверждающими прием ранее отправленных данных.

Контрольные вопросы

1. Для чего используются порты в протоколах транспортного уровня?
2. Какие Вы знаете особенности протокола UDP и TCP?
3. Какого размера заголовок протокола UDP?
4. Какой максимальный и минимальный размер заголовка протокола TCP?
5. Какие данные используются в расчете контрольной суммы заголовка протоколов UDP и TCP?
6. Что такое «окно» в протоколе TCP?
7. Как рассчитать оптимальный размер «окна» TCP?
8. В чем заключается проблема «бестолкового окна» и как ее решить?
9. В чем заключаются барьеры производительности протокола TCP?
10. В чем преимущества протокола SCTP в сравнении с UDP и TCP?
11. В чем заключается особенность DoS атак?

ГЛАВА 4. ПРОТОКОЛЫ ПРИКЛАДНОГО УРОВНЯ

4.1.Протокол POP3

Перед работой через протокол POP3 сервер прослушивает порт 110. Когда клиент хочет использовать этот протокол, он должен создать TCP соединение с сервером [8]. Когда соединение установлено, сервер отправляет приглашение. Затем клиент и POP3 сервер обмениваются информацией пока соединение не будет закрыто или прервано.

Команды POP3 состоят из ключевых слов, за некоторыми следует один или более аргументов. Все команды заканчиваются парой CRLF (в Visual Basic константа vbCrLf). Ключевые слова и аргументы состоят из печатаемых ASCII символов. Ключевое слово и аргументы разделены одиночным пробелом. Ключевое слово состоит от 3-х до 4-х символов, а аргумент может быть длиной до 40-ка символов.

Ответы в POP3 состоят из индикатора состояния и ключевого слова, за которым может следовать дополнительная информация. Ответ заканчивается парой CRLF. Существует только два индикатора состояния: "+OK" - положительный и "-ERR" - отрицательный.

Ответы на некоторые команды могут состоять из нескольких строк. В этих случаях каждая строка разделена парой CRLF, а конец ответа заканчивается ASCII символом 46 (".") и парой CRLF.

POP3 сессия состоит из нескольких режимов. Как только соединение с сервером было установлено и сервер отправил приглашение, то сессия переходит в режим AUTHORIZATION (Авторизация). В этом режиме клиент должен идентифицировать себя на сервере. После успешной идентификации сессия переходит в режим TRANSACTION (Передача). В этом режиме клиент запрашивает сервер выполнить определённые команды. Когда клиент отправляет команду QUIT, сессия переходит в режим UPDATE. В этом режиме POP3 сервер освобождает все занятые ресурсы и завершает работу. После этого TCP соединение закрывается.

У POP3 сервера может быть INACTIVITY AUTOLOGOUT таймер. Этот таймер должен быть, по крайней мере, с интервалом 10 минут. Это значит, что если клиент и сервер не взаимодействуют друг с другом, сервер автоматически прерывает соединение и при этом не переходит в режим UPDATE.

Авторизация

Как только будет установлено TCP соединение с POP3 сервером, он отправляет приглашение, заканчивающееся парой CRLF, например:

S: +OK POP3 server ready

Теперь POP3 сессия находится в режиме AUTHORIZATION. Клиент должен идентифицировать себя на сервере, используя команды USER и PASS. Сначала надо отправить команду USER, после которой в качестве аргумента следует имя пользователя. Если сервер отвечает положительно, то теперь необходимо отправить команду PASS, за которой следует пароль. Если после отправки команды USER или PASS сервер отвечает негативно, то можно

попробовать авторизироваться снова или выйти из сессии с помощью команды QUIT. После успешной авторизации сервер открывает и блокирует maildrop (почтовый ящик). В ответе на команду PASS сервер сообщает, сколько сообщений находится в почтовом ящике и передаёт их общий размер. Теперь сессия находится в режиме TRANSACTION. Подведём итоги с командами:
Команда: USER [имя]
Аргументы: [имя] - строка, указывающая имя почтового ящика
Описание: Передаёт серверу имя пользователя
Возможные ответы:

- +OK name is a valid mailbox
- -ERR never heard of mailbox name

Примеры:
C: USER MonstrVB
S: +OK MonstrVB is a real hoopy frood
...
C: USER MonstrVB
S: -ERR sorry, no mailbox for frated here

Команда: PASS [пароль]
Аргументы: [пароль] - пароль для почтового ящика
Описание: Передаёт серверу пароль почтового ящика
Возможные ответы:

- +OK maildrop locked and ready
- -ERR invalid password
- -ERR unable to lock maildrop

Примеры:
C: USER MonstrVB
S: +OK MonstrVB is a real hoopy frood
C: PASS mymail
S: +OK MonstrVB's maildrop has 2 messages (320 octets)
...
C: USER MonstrVB
S: +OK MonstrVB is a real hoopy frood
C: PASS mymail
S: -ERR maildrop already locked

Команда: QUIT
Аргументы: нет
Описание: Сервер завершает POP3 сессию и переходит в режим UPDATE
Возможные ответы:

- +OK

Примеры:
C: QUIT
S: +OK dewey POP3 server signing off

Основные команды (Transaction)

После успешной идентификации пользователя на сервере POP3 сессия переходит в режим TRANSACTION, где пользователь может передавать ниже следующие команды. После каждой из таких команд следует ответ сервера. Вот доступные команды в этом режиме:

Команда: STAT

Аргументы: нет

Описание: В ответ на вызов команды сервер выдаёт положительный ответ "+OK", за которым следует количество сообщений в почтовом ящике и их общий размер в символах. Сообщения, которые помечены для удаления, не учитываются в ответе сервера.

Возможные ответы:

- +OK n s

Примеры:

C: STAT
S: +OK 2 320

Команда: LIST [сообщение]

Аргументы: [сообщение] - номер сообщения (необязательный аргумент)

Описание: Если был передан аргумент, то сервер выдаёт информацию об указанном сообщении. Если аргумент не был передан, то сервер выдаёт информацию о всех сообщениях, находящихся в почтовом ящике. Сообщения, помеченные для удаления не перечисляются.

Возможные ответы:

- +OK scan listing follows
- -ERR no such message

Примеры:

C: LIST
S: +OK 2 messages (320 octets)
S: 1 120
S: 2 200
S: .
...
C: LIST 2
S: +OK 2 200
...
C: LIST 3
S: -ERR no such message, only 2 messages in maildrop

Команда:RETR [сообщение]

Аргументы: [сообщение] - номер сообщения

Описание: После положительного ответа сервер передаёт содержание сообщения

Возможные ответы:

- +OK message follows

- -ERR no such message

Примеры:
C: RETR 1
S: +OK 120 octets
S:
S: .

Команда: DELE [ообщение]
Аргументы: [ообщение] - номер сообщения
Описание: POP3 сервер помечает указанное сообщение как удалённое, но не удаляет его, пока сессия не перейдёт в режим UPDATE
Возможные ответы:
- +OK message deleted
- -ERR no such message

Примеры:
C: DELE 1
S: +OK message 1 deleted
...
C: DELE 2
S: -ERR message 2 already deleted

Команда: NOOP
Аргументы: нет
Описание: POP3 сервер ничего не делает и всегда отвечает положительно
Возможные ответы:
- +OK

Примеры:
C: NOOP
S: +OK

Команда: RSET
Аргументы: нет
Описание: Если какие - то сообщения были помечены для удаления, то с них снимается эта метка
Возможные ответы:
- +OK

Примеры:
C: RSET
S: +OK maildrop has 2 messages (320 octets)

Обновление

Когда клиент передаёт команду QUIT в режиме TRANSACTION, то сессия переходит в режим UPDATE. В этом режиме сервер удаляет все сообщения, помеченные для удаления. После этого TCP соединение закрывается.

Дополнительные POP3 команды

Следующие дополнительные команды дают вам большую свободу при работе с сообщениями: Команда: TOP [сообщение] [n] Аргументы: [сообщение] - номер сообщения [n] - положительное число (обязательный аргумент). Описание: Если ответ сервера положительный, то после него он передаёт заголовки сообщения и указанное количество строк из тела сообщения. Возможные ответы: +OK top of message follows -ERR no such message. Примеры: C: TOP 1 10 S: +OK S: <здесь POP3 сервер передаёт заголовки первого сообщения и первые 10-ть строк из тела сообщения.> S: . ... C: TOP 100 3 S: -ERR no such message Команда: UIDL [сообщение] Аргументы: [сообщение] - номер сообщения (необязательный аргумент). Описание: Если был указан номер сообщения, то сервер выдаёт уникальный идентификатор для этого сообщения. Если аргумент не был передан, то идентификаторы перечисляются для всех сообщений, кроме помеченных для удаления. Возможные ответы: +OK unique-id listing follows -ERR no such message Примеры: C: UIDL S: +OK S: 1 whqtswO00WBw418f9t5JxYwZ S: 2 QhdPYR:00WBw1Ph7x7 S: . ... C: UIDL 2 S: +OK 2 QhdPYR:00WBw1Ph7x7 ... C: UIDL 3 S: -ERR no such message, only 2 messages in maildrop

Пример простого сеанса с POP3 сервером

S: <создаём новое TCP соединение с POP3 сервером через порт 110>
S: +OK POP3 server ready
C: USER MonstrVB
S: +OK User MonstrVB is exists
C: PASS mymail
S: +OK MonsrVB's maildrop has 2 messages (320 octets)
C: STAT
S: +OK 2 320
C: LIST
S: +OK 2 messages (320 octets)
S: 1 120
S: 2 200
S: .
C: RETR 1
S: +OK 120 octets
S:
S: .
C: DELE 1
S: +OK message 1 deleted
C: RETR 2
S: +OK 200 octets
S:
S: .
C: DELE 2
S: +OK message 2 deleted

C: QUIT
S: +OK dewey POP3 server signing off (maildrop empty)
C: <закрываем соединение>

4.2.Протокол SMTP

Основная задача протокола SMTP (Simple Mail Transfer Protocol) заключается в том, чтобы обеспечивать передачу электронных сообщений (почту) [9]. Для работы через протокол SMTP клиент создаёт TCP соединение с сервером через порт 25. Затем клиент и SMTP сервер обмениваются информацией пока соединение не будет закрыто или прервано. Основной процедурой в SMTP является передача почты (Mail Procedure). Далее идут процедуры форвардинга почты (Mail Forwarding), проверка имён почтового ящика и вывод списков почтовых групп. Самой первой процедурой является открытие канала передачи, а последней - его закрытие.

Команды SMTP указывают серверу, какую операцию хочет произвести клиент. Команды состоят из ключевых слов, за которыми следует один или более параметров. Ключевое слово состоит из 4-х символов и разделено от аргумента одним или несколькими пробелами. Каждая командная строка заканчивается символами CRLF. Вот синтаксис всех команд протокола SMTP (SP - пробел):

HELO <SP> <domain> <CRLF>
MAIL <SP> FROM:<reverse-path> <CRLF>
RCPT <SP> TO:<forward-path> <CRLF>
DATA <CRLF>
RSET <CRLF>
SEND <SP> FROM:<reverse-path> <CRLF>
SOML <SP> FROM:<reverse-path> <CRLF>
SAML <SP> FROM:<reverse-path> <CRLF>
VRFY <SP> <string> <CRLF>
EXPN <SP> <string> <CRLF>
HELP <SP> <string> <CRLF>
NOOP <CRLF>
QUIT <CRLF>

Обычный ответ SMTP сервера состоит из номера ответа, за которым через пробел следует дополнительный текст. Номер ответа служит индикатором состояния сервера.

Отправка почты

Первым делом подключаемся к SMTP серверу через порт 25. Теперь надо передать серверу команду HELLO и наш IP адрес:

C: HELLO 195.161.101.33
S: 250 smtp.mail.ru is ready

При отправке почты передаём некоторые нужные данные (отправитель, получатель и само письмо):

C: MAIL FROM:<drozd> 'указываем отправителя

S: 250 OK
C: RCPT TO:<drol@mail.ru> 'указываем получателя
S: 250 OK
указываем серверу, что будем передавать содержание письма (заголовок и тело письма)
C: DATA
S: 354 Start mail input; end with <CRLF>.<CRLF>
передачу письма необходимо завершить символами CRLF.CRLF
S: 250 OK
C: From: Drozd <drozd@mail.ru>
C: To: Drol <drol@mail.ru>
C: Subject: Hello
между заголовком письма и его текстом не одна пара CRLF, а две.
C: Hello Drol!
C: You will be die on next week!
заканчиваем передачу символами CRLF.CRLF
S: 250 OK
Теперь завершаем работу, отправляем команду QUIT:
S: QUIT
C: 221 smtp.mail.ru is closing transmission channel

Другие команды

- SEND - используется вместо команды MAIL и указывает, что почта должна быть доставлена на терминал пользователя.
- SOML, SAML - комбинации команд SEND или MAIL, SEND и MAIL соответственно.
- RSET - указывает серверу прервать выполнение текущего процесса. Все сохранённые данные (отправитель, получатель и др.) удаляются. Сервер должен отправить положительный ответ.
- VRFY - просит сервер проверить, является ли переданный аргумент именем пользователя. В случае успеха сервер возвращает полное имя пользователя.
- EXPN - просит сервер подтвердить, что переданный аргумент - это список почтовой группы, и если так, то сервер выводит членов этой группы.
- HELP - запрашивает у сервера полезную помощь о переданной в качестве аргумента команде.
- NOOP - на вызов этой команды сервер должен положительно ответить. NOOP ничего не делает и никак не влияет на указанные до этого данные.

4.3. Протокол FTP

FTP (RFC-959) обеспечивает файловый обмен между удаленными пользователями [1, 10]. Протокол FTP формировался многие годы. Первые реализации в МТИ относятся к 1971. (RFC 114 и 141). RFC 172 рассматривает

протокол, ориентированный на пользователя, и предназначенный для передачи файлов между ЭВМ. Позднее в документах RFC 265 и RFC 281 протокол был усовершенствован. Заметной переделке протокол подвергся в 1973, и окончательный вид он обрел в 1985 году. Таким образом, данный протокол является одним из старейших.

Для реализации обмена между двумя персональными ЭВМ в пределах сети (программные пакеты PCTCP и т.д.) можно резидентно загрузить FTPSRV или другую эквивалентную программу. Также как и в случае TELNET необходима идентификация, но многие депозитарии допускают анонимный вход (имя пользователя ANONYMOUS, RFC-1635), который не требует слова пропуска (пароля) или допускает ввод вашего почтового адреса вместо него.

Работа FTP на пользовательском уровне содержит несколько этапов [10]:

1. Идентификация (ввод имени-идентификатора и пароля).
2. Выбор каталога.
3. Определение режима обмена (поблочный, поточный, ASCII или двоичный).
4. Выполнение команд обмена (get, mget, dir, mdel, mput или put).
5. Завершение процедуры (quit или close).

FTP довольно необычная процедура, так как поддерживает две логические связи между ЭВМ (рис. 4.1). Одна связь служит для удаленного доступа и использует протокол Telnet. Другая связь предназначена для обмена данными. Сервер производит операцию passive open для порта 21 и ждет соединения с клиентом. Клиент осуществляет операцию active open для порта 21. Канал остается активным до завершения процедуры FTP. TOS (тип IP-сервиса) соответствует минимуму задержки, так как этот канал используется для ручного ввода команд. Канал для передачи данных (TCP) формируется каждый раз для пересылки файлов. Канал открывается перед началом пересылки и закрывается по коду end_of_file (конец файла). IP-тип сервиса (TOS) в этом случае ориентирован на максимальную пропускную способность.

Конечный пользователь взаимодействует с протокольным интерпретатором, в задачи которого входит управление обменом информацией между пользователем и файловой системой, как местной, так и удаленной. Схема взаимодействия различных частей Internet при работе FTP изображена на рисунке 4.1.

Сначала по запросу клиента формируется канал управления, который в дальнейшем используется для передачи команд от клиента и откликов от сервера. Информационный канал формируется сервером по команде клиента, он не должен существовать постоянно на протяжении всей FTP-сессии и может формироваться и ликвидироваться по мере необходимости. Канал управления может быть закрыт только после завершения информационного обмена. Для канала управления используется протокол Telnet. После того как управляющий канал сформирован, клиент может посылать по нему команды. Сервер воспринимает, интерпретирует эти команды и передает отклики.

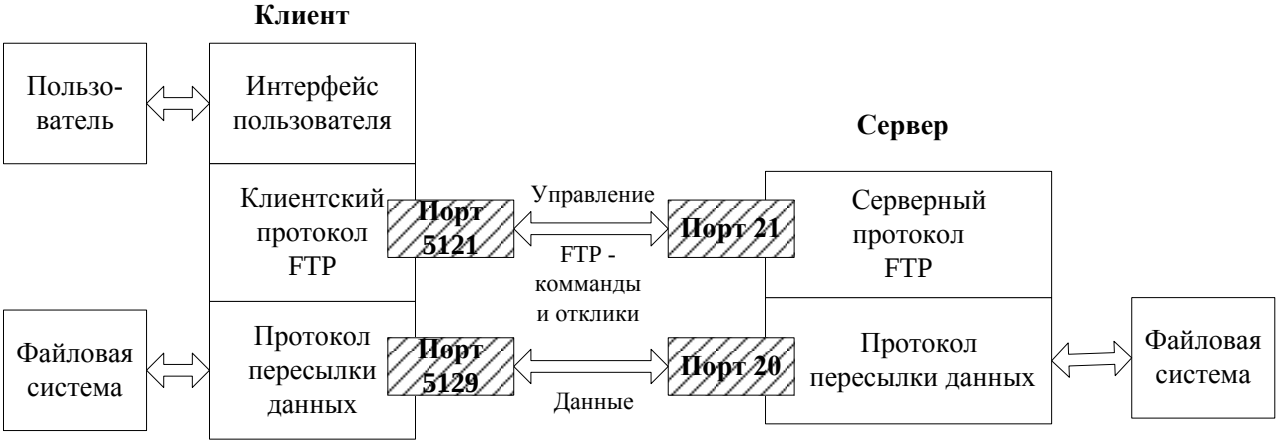


Рис. 4.1 Схема работы протокола FTP

Возможна и другая схема взаимодействия, когда по инициативе клиента осуществляется файловый обмен между двумя ЭВМ, ни одна из которых не является машиной клиента (см. рис. 4.2).

Рис. 4.2 Организация информационного обмена между двумя удаленными машинами

Какие команды можно передавать по управляющему соединению? Существуют команды аутентификации, дающие возможность пользователю указать идентификатор, пароль и регистрационную запись для работы с FTP.

Команды пересылки файлов позволяют:

- копировать одиночный файл между хостами;
- копировать несколько файлов между хостами;
- добавлять содержимое локального файла к удаленному файлу;
- копировать файл и добавлять к его имени номер для формирования уникального имени (например, файлы ежедневной регистрации получат имена log. 1, log.2 и т.д.).

Команды обслуживания файлов разрешают:

- просмотреть список файлов каталога;
- узнать текущий каталог и изменить его на другой;
- создавать и удалять каталоги;
- переименовывать или удалять файлы.

Управляющие команды служат для:

- идентификации пересылки файлов ASCII, EBCDIC или двоичных файлов;

- проверки структурирования файла (как последовательность байт или как последовательность записей);
- указания способа пересылки файла (например, как поток октетов).

Пересылаемые по управляющему соединению команды имеют стандартный формат. Например, команда *RETR* используется для копирования файла из сервера на сайт клиента.

Использование команд в текстовом диалоге

Многие пользователи предпочитают графический интерфейс, доступный на настольных системах, но текстовый интерфейс позволяет лучше понять внутренние процессы протокола FTP.

При выполнении команды *help* выводится справка. Существующие команды имеют синонимы, например *Is* и *dir* — для запроса сведений о каталоге, *put* и *send* — для копирования файла на удаленный хост, *get* и *recv* — для получения файла от удаленного хоста или *bye* и *quit* — для выхода из FTP.

Используя команды *mget* или *mput* и глобальные подстановочные символы можно одновременно копировать несколько файлов. Например, *mget* а* извлечет копии каждого файла с именем, начинающимся на букву а. Такой режим включается параметром ***glob,*** который разрешает или запрещает применение *глобальных* подстановочных символов.

В представленный ниже диалог включен вывод отладочной информации, чтобы дать некоторое представление о работе протокола:

- строки, начинающиеся на —>, показывают сообщения, посланные локальным' хостом по управляющему соединению;
- строки, начинающиеся с числа, соответствуют сообщениям, посланным удаленным сервером для отчета о результате выполнения команды.

csc.sibsutis> *ftp*

ftp> *help*

```
 Commands may be abbreviated. Commands are:
!       ?       acct    append  ascii   binary  bye     cd      debug
delete  dir     drive   exit    fcd     fdir    fpwd    get     help
iget    image   iput    lcd     ldir    lmkdir  local   login   lpwd
ls      mdelete mget    mkdir   mput    option  parent  passive put
pwd     quit    quote   rename  retrieve rmdir  send    server  show
stat    store   take    tenex   tget    tput    type    user    verbose
```

ftp> *debug*

```
 Debugging on (debug = 1).
```

ftp> *open sibsutis.ru*

```
 Connected to sibsutis.ru.
 220 sibsutis.ru FTP server ready.
```

Далее проводим авторизацию (команды *USER, PASS*).

Команда *status* (статус) показывает текущие параметры сеанса FTP. Пока отметим, что *тип данных* (Type) указан как ASCII. При пересылке текстовых файлов FTP часто предполагает это значение по умолчанию.

ftp> *status*

```
Connected to tigger.jvnc.net. No proxy connection.
Mode: stream; Type: ascii; Form: non-print;
Structure: file
Verbose: on; Bell: off; Prompting: on;
Globbing: on
Store unique: off; Receive unique: off
Case: off; CR stripping: on Ntrans: off
Nmap: off
Hash mark printing: off; Use of PORT cmds: on
```

Затем запрашивается список файлов каталога. Такой список может быть очень большим, поэтому FTP посылает его по соединению для данных:

ftp> *dir*

FTP необходим порт для пересылки данных. Клиент посылает команду ***PORT,*** которая идентифицирует его IP-адрес (4 байта) и новый порт (2 байта), чтобы использовать эти значения при пересылке данных. Байты преобразуются в десятичный формат и разделяются запятыми. IP-адрес 128.36.4.22 будет записан как 128,36,4,22, а порт 2613 — как 10,53.

--> *PORT 128,36,4,22,10,53*

```
200 PORT command successful.
```

Сервер откроет соединение по указанному адресу socket. Команда **LIST—** это формальное сообщение для запроса подробного списка файлов каталога:

—> *LIST*

Далее сервер открывает соединение с объявленным клиентом портом и предоставляет список файлов.

Сразу после пересылки списка файлов соединение данных будет закрыто. Затем мы можем получить необходимый файл (например, *article*).

ftp> *get article*

Клиент указывает новый адрес socket для переноса файла. Отметим, что на сей раз используется клиентский порт 2614 (10,54).

—> *PORT 128,36,4,22,10,54*

```
200 PORT command successful.
```

—> *RETR article*

```
150 Opening ASCII mode data connection for article
(3010 bytes).
226 Transfer complete.
```

По завершении пересылки файла соединение для данных закрывается.

Отметим, что сценарий для соединения данных был таким:

- локальный клиент получил новый порт и использовал управляющее соединение, чтобы сообщить серверу FTP номер своего порта;
- FTP-сервер связался с новым портом данных клиента;
- данные были переданы;
- соединение было закрыто.

Можно применять альтернативный сценарий. Если клиент посылает команду ***PASV,*** сервер возвращает номер порта и переходит к прослушиванию

установки соединения данных от клиента. Ранее преобладало использование команды *PORT*. Однако теперь клиент может послать команду *PASV* для пересылки файлов через простую систему защиты (firewall), которая не разрешает установку соединений из поступающих сообщений.

При работе с файлами большого размера иногда обнаруживается, что пересылается не тот файл. Хорошая реализация должна позволять отменить пересылку. Для текстового интерфейса это обычно делается через комбинацию клавиш CONTROL-C, а в графическом интерфейсе — специальной кнопкой *Abort* (остановить).

PASV/PORT?

Организации обеспечивают безопасность своих сетей через средства защиты (firewall), применяющие к датаграммам определенный критерий фильтрации и ограничивающие входящий трафик. Часто простейшие средства защиты разрешают пользователям локальной сети инициировать соединение, но блокируют все попытки создания соединения извне.

Исходная спецификация FTP определяет команду *PORT* как средство по умолчанию для установки соединения данных. В результате многие реализации основывают установку соединения только на этой команде. Однако команда *PORT* требует открытия соединения от внешнего файлового сервера к клиенту, что обычно блокируется средством защиты локальной сети.

Новые реализации поддерживают команду ***PASV,*** указывающую серверу на выделение нового порта для соединения данных с пересылкой IP-адреса и номера порта сервера в ответе клиенту. Далее клиент может самостоятельно открыть соединение с сервером.

Типы данных, структуры файлов и методы пересылки

На обоих концах соединения необходимо обеспечить единый формат для пересылаемых данных. Этот файл текстовый или двоичный? Он структурирован по записям или по блокам?

Для описания формата пересылки используются три атрибута: *тип данных* (data type), *структура файла* (file structure) и *режим пересылки* (transmission mode). Допустимые значения этих атрибутов рассмотрены ниже. В общем случае применяются:

- пересылка текста ASCII или двоичных данных;
- неструктурированный файл, который рассматривается как последовательность байт;
- режим пересылки рассматривает файл как поток байт.

Однако есть и несколько исключений. Некоторые хосты структурируют текстовые файлы как последовательность записей. Хосты IBM используют для текстовых файлов кодирование EBCDIC и проводят обмен файлами как набором структурированных блоков, а не как потоком байт.

Типы данных

Файл может содержать текст ASCII, EBCDIC или двоичный образ данных (существует еще тип, называемый *локальным* или *логическим байтом* и применяемый для компьютеров с размером байта в 11 бит). Текстовый файл

может содержать обычный текст или текст, форматированный для вывода на принтер. В последнем случае в нем будут находиться коды вертикального форматирования:

- символы вертикального форматирования *Telnet* для режима NVT (т.е. <CR>, <LF>, <NL>, <VT>, <FF>);
- символы вертикального форматирования ASA (ФОРТРАН).

Типом данных по умолчанию является нераспечатываемый текст ASCII (текст без управляющих символов форматирования)**.** Тип данных может быть изменен стандартной командой ***TYPE,*** пересылаемой по управляющему соединению.

Пересылка текста ASCII

Хотя текст ASCII является стандартным, компьютеры интерпретируют его по-разному из-за различия в кодах конца строки. Системы Unix используют для этого <LF>, компьютеры PC — <CR><LF>, а Macintosh — <CR>.

Для устранения этих различий FTP превращает локальный текстовый файл ASCII в формат NVT, а приемник преобразует NVT ASCII в собственный локальный формат. Например, если текстовый файл копируется с системы Unix на PC, все коды концов строк (в Unix — <LF>) при получении файла на PC нужно преобразовать в <CR><LF>.

Пересылка текста EBCDIC

Поддерживающие кодировку EBCDIC хосты обеспечивают весьма полезную команду пользовательского интерфейса, инициирующую пересылку по управляющему соединению команды *TYPE E***.** Текстовые символы EBCDIC пересылаются по соединению в своем обычном 8-разрядном формате. Строки завершаются символом новой строки EBCDIC (<NL>).

Пересылка двоичных данных

С пересылки текстов ASCII легко переключиться на двоичный образ данных. В текстовом пользовательском интерфейсе для этого служит команда *binary,* а в графическом — командная кнопка binary (двоичные данные). Клиент меняет тип пересылаемых данных командой *TYPE I,* передаваемой по управляющему соединению.

Что произойдет, если пользователь забудет переключить тип данных с ASCII на двоичный при копировании двоичного файла? Хорошие реализации FTP предупредят, что задана ошибочная операция, и позволят до начала пересылки файла изменить тип данных. К сожалению, многие реализации идут еще дальше и "помогают" изменять все двоичные байты, которые выглядят как символы конца строк (исправляя их на специальные заполнители или полностью удаляя их из текста). Некоторые действительно плохие реализации все же начинают пересылку файла и аварийно завершаются в середине выполнения такой операции.

Структуры файлов

В FTP поддерживаются две структуры:

- *файловая структура,* соответствующая неструктурированному файлу, который рассматривается как последовательность байт;

- *структура записей***,** которая применяется для файлов, состоящих из последовательности записей.

Более распространена *файловая структура***,** которая применяется по умолчанию. Перейти на *структуру записей* можно стандартной командой *STRU R,* пересылаемой по управляющему соединению.

Режимы пересылки

Режим пересылки и структура файла определяют, как будут форматированы данные для обмена по соединению. Существуют три режима пересылки: *stream* (поток), *block* (блочный режим) и *compressed* (сжатые данные).

- В режиме потока и файловой структуры файл передается как поток байт. FTP возлагает н TCP обеспечение целостности данных и не включает в данные никаких заголовков или разделителей. Единственным способом указания на конец файла будет нормальное завершение соединения для данных.
- Для режима потока и структуры записей каждая запись отделяется 2-байтовым управляющим кодом конца записи (End Of Record — EOR), а конец файла отмечается символами конца файла (End Of File — EOF). EOR кодируется как X'FF 01, а EOF — X'FF 02. Для последний записи файла EOR и EOF записываются как X'FF 03. Если файл содержит байт данных из одних единиц, то такой байт представляется при пересылке как X'FF FF.
- В блочном режиме файл пересылается как последовательность блоков данных. Каждый блок начинается 3-байтовым заголовком (см. рис.4.3).
- Режим сжатия данных используется крайне редко, поскольку обеспечивает очень неудачный метод архивирования, разрушающий последовательность повторяющихся байт. Обычно пользователю проще применить одну из более удачных программ сжатия, широкодоступных на современных компьютерах, и далее пересылать полученный архивный файл как двоичные данные.

8 бит	16 бит
Дескрипторные флаги	Счетчик байт
Конец блока — EOR Конец блока — EOF Restart Marker	Количество следующих далее байт

Рис. 4.3 Формат заголовка блочного режима пересылки FTP

Блок может содержать целую запись, или в записи объединяются несколько блоков. Дескриптор содержит:

- флаг End Of Record для идентификации границы записи;
- флаг End Of File, который указывает, является ли блок последним при пересылке файла;
- флаг Restart Marker (маркер перезапуска), указывающий, содержит ли данный блок текстовую строку, которую можно использовать для указания точки перезапуска после неудачной пересылки файла в более поздней точке.

Режим потока наиболее распространен и используется по умолчанию. Изменить его на блочный режим можно стандартной командой *MODE B*, пересылаемой по управляющему соединению.

Преимущество структуры записей или блочного режима, состоит в том, что будет явно отмечен конец файла и после завершения его пересылки можно сохранить соединение для данных, а, следовательно, использовать его для нескольких пересылок.

Например, в ответ на команду *status* может быть дан такой ответ:

```
Mode: stream; Type: ascii; Form: non-print;
Structure: file
```

Т.е. по умолчанию установлен поточный режим пересылки данных, тип данных ASCII без форматирования для печати и файловая структура (соответствующая неструктурированному файлу).

Базовые команды FTP

Ниже приведен список базовых команд FTP (см. табл. 4.1). Следует разделять внутренний набор команд FTP, которыми обмениваются клиент и сервер по командному каналу, и набор команд доступный пользователю. Служебные команды содержат три или четыре заглавные буквы. Эти наборы команд перекрываются лишь частично. Служебные команды унифицированы (они выделены в приведенном выше примере FTP-сессии жирным шрифтом, в помещенной ниже таблице эти команды представлены в ее верхней части), пользовательский же набор команд может варьироваться от реализации к реализации. Если выдать команду FTP без аргументов, система обычно откликается приглашением FTP> и вы можете выполнить некоторые из приведенных ниже команд (весь набор становится доступным только после идентификации).

Таблица 4.1

Субкоманды FTP	**Описание**
ABOR	Прерывание исполнения предыдущей FTP-команды и связанного с ней обмена
ACCT<SP> <account-information>	Ввод идентификатора пользователя (ID)
ALLO <SP> <десятичное целое> [<SP> R <SP> <десятичное целое>]	Зарезервировать достаточно места (в байтах) для пересылки файла. Для файлов с постраничной структурой после символа R указывается число записей
APPE <SP> <проход>	Присовокупить передаваемые данные к файлу, указанному в параметре проход
CDUP	Переход в каталог прародитель
CWD <SP> <проход>	Изменить рабочий каталог (CD)

DELE <SP> <проход>	Стереть файл (del)
HELP	Выдать справочную информацию о выполнимых командах
HELP [<SP> <строка>]	Выдать описание работы данной команды
LIST [<SP> <проход>]	Вывод списка файлов или каталогов (dir)
MKD <SP> <проход>	Создать каталог
MODE <SP> <код режима>	Режим обмена = поток, блоки или со сжатием
NLST [<SP> <проход>]	Переслать оглавление каталога от сервера к клиенту
NOOP	Пустая команда
PASS <SP> <пароль>	Слово-пропуск (пароль) пользователя, заполняется пользователем
PASV	Перевести сервер в режим прослушивания информационного порта на предмет установления соединения
PORT <SP> <порт ЭВМ>	IP-адрес и номер порта клиента
PWD	Выдать имя текущего каталога
QUIT	Уход из FTP
REIN	Завершение сессии и открытие новой
REST <SP> <маркер>	Возобновление обмена, начиная с места, указанного маркером
RETR <SP> <проход>	Переслать копию файла (get) другому адресату
RMD <SP> <проход>	Удалить каталог
RNFR <SP> <проход>	Начало процедуры переименования файла (Rename From)
RNTO <SP> <проход>	Указание нового имени файла при переименовании (Rename To)_
SITE <SP> <строка>	Используется сервером для реализации локально специфических команд

SMNT <SP> <проход>	Позволяет пользователю смонтировать нужную файловую систему
STAT	Выдать текущие значения параметров (STATUS)
STOR <SP> <проход>	Сервер должен запомнить полученные данные в виде файла
STOU	Аналог команды STOR, но записывает файл в текущий каталог и присваивает файлу уникальное имя
STRU <SP> <код структуры>	Структура файла = файл, запись или страница
SYST	Сервер сообщает тип системы
TYPE <SP> <код типа>	Специфицирует тип информации, часто для этой цели используются команды binary и ASCII
USER <SP> < [имя [пропуск]] >	Идентифицирует пользователя, запрашивается сервером
?	тоже что и HELP
lcd	Изменить локальный каталог (на вашей ЭВМ);
!	Выйти временно из FTP и уйти в Shell (UNIX)
! команда	Исполнить команду Shell (UNIX)
close	Прервать связь с удаленным сервером, оставаясь в FTP
open [имя_ЭВМ]	Установить связь с указанным удаленным сервером
dir	Выдать содержимое удаленного каталога

<SP> пробел; все команды завершаются последовательностью <CRLF> возврат каретки + перевод строки. В квадратных скобках записан опционный аргумент. Выполнение любой команды можно прервать с помощью Ctrl-C.

Возможная форма обращения к FTP (SunOS 4.1): **FTP [-опции] [имя_ЭВМ]**

Допустимы следующие опции (модификаторы) команды:

-d включение отладочного режима;

-g блокировка группового исполнения команд;

-i выключение интерактивного приглашения при множественной пересылке файлов;

-v отображает все отклики удаленного сервера и статистику обмена; этот режим работает обычно по умолчанию.

В рамках процедуры FTP доступны следующие команды (приведенный перечень команд является неполным):

! [команда]	Исполняется команда интерпретатора shell вашей ЭВМ (UNIX). Если имя команды явно не введено, система переходит в интерактивный режим shell.
$ имя-макро [аргументы]	Выполняется макро, имя которого введено, аргументы используются этим макро.
account [пароль]	Позволяет ввести пароль, необходимый для доступа в удаленный сервер.
append имя_местного_файла [имя_удаленного_файла]	Добавить местный файл к файлу на удаленном сервере.
Bye	Завершает FTP-сессию.
case	Переключает регистр символов, которыми записаны имена файлов на удаленной ЭВМ, в процессе выполнения команды MGET. Если case включен (по умолчанию выключен), все прописные буквы в именах файлов на удаленной ЭВМ, меняются при переносе в вашу ЭВМ на строчные.
close	Завершает FTP-сессию и возвращает систему в интерактивный командный режим. Все описанные ранее макро стираются.
debug [debug-value]	Включает/выключает режим отладки. Значение debug-value определяет отладочный уровень. Если отладка включена, FTP отображает на экране каждую команду, посылаемую удаленной ЭВМ. Эта информация помечается символом '-->'.
dir [удаленный каталог] [местный файл]	Выдает на экран содержимое удаленного каталога. Если в качестве параметра указано имя местного файла, результат заносится в

	него. Если имя удаленного каталога не указано, команда выполняется для текущего каталога.
disconnect	Синоним close.
hash	Включает/выключает знак (#). Во включенном состоянии отмечается пересылка каждого блока, что позволяет визуально контролировать процесс обмена.
macdef macro-name	Определяет макро. Последующие строки запоминаются в качестве текста макро с именем macro-name. Нулевая строка (двойное нажатие клавиши RETURN) завершает ввод текста макро. Можно ввести до 16 макро с суммарным объемом до 4096 символов.
mdelete [имена_файлов_на удаленной_ЭВМ]	Удаляет файлы на удаленной ЭВМ.
open имя-ЭВМ [port]	Устанавливает связь с указанным FTP-сервером (ЭВМ) через специфицированный порт.
prompt	Включает/выключает интерактивные запросы со стороны ЭВМ. Это бывает полезным при выполнении групповых команд MPUT, MGET или MDELETE и позволяет проводить соответствующие операции над файлами выборочно.
proxy	ftp-команда выполняет FTP-команду на вторичной удаленной ЭВМ. Эта команда позволяет связать два удаленных FTP-сервера и осуществить пересылку файлов между ними. Первой proxy-командой должна быть команда open, необходимая для связи с вторичным сервером. Введите команду proxy ?, чтобы проверить выполнимость этих команд на данном сервере.
quit	Синоним bye.
recv	Удаленный_файл [местный_файл] синоним команды get.
remotehelp [имя_команды]	Запрашивает справочную информацию у удаленного FTP-сервера. Если имя_команды

	задано, запрашивается информация о конкретной команде.
runique	Включает режим записи файлов в вашу ЭВМ только с уникальными именами. Если файл с таким именем уже существует, то новому файлу будет присвоено имя с расширением .1, если и такое имя уже есть, то с расширением .2. Это может продолжаться вплоть до расширения .99, после чего будет выдано сообщение об ошибке. Впрочем, такую ситуацию вообразить крайне трудно, если вы сами не наплодили файлов с цифровыми расширениями. Для команды mget это крайне полезная функция, которая застрахует вас от стирания ваших файлов из текущего каталога, имеющих имена, совпадающие с именами на удаленном сервере. По умолчанию runique не включено.
send local-file [remote-file]	Синоним команды put.
status	Отображает текущее состояние ftp.

В депозитариях можно встретить файлы следующих разновидностей (все виды нижеперечисленных файлов пересылаются в режиме binary, а не ASCII) (табл. 4.2):

Таблица 4.2

Тип файла	Пример записи имени файла	Программа обработки файла
Архивированный файл	файл.Z	compress, uncompress
tar-файл	файл.tar	tar
Архивированный tar-файл	файл.tar.Z	tar, compress, uncompress
	файл.tar.gz	Применен архиватор GZIP
uuencode-файл	файл.uue	uuencode, uudecode
Архивированный uuencode-файл	файл.uue.Z	uuencode, uudecode, compress, uncompress
zip-файл	файл.zip	pkzip, pkunzip
shar-файл	файл.shar	shar, sh, unshar
сжатый shar-файл	файл.shar.Z	shar, sh, unshar, compress, uncompress

При выполнении FTP система возвращает трехразрядные десятичные коды-отклики, которые позволяют судить о корректности обмена и диагностировать процедуру. Выдача кода сопровождается текстом-комментарием. Первая цифра может принимать значения от 1 до 5. Структура кодов показана в таблице 4.3:

Таблица 4.3 Коды диагностики

Значение кода-отклика	Описание
1yz	Позитивный предварительный отклик, который означает, что операция начата. До завершения процедуры следует ожидать как минимум еще один отклик.
2yz	Сигнал успешного завершения процедуры, говорящий о том, что можно ввести новую команду.
3yz	Положительный промежуточный отклик, указывающий на то, что команда воспринята, но для продолжения требуется дополнительная информация.
4yz	Негативный отклик, свидетельствующий о том, что команда не воспринята, но можно попробовать ее исполнить еще раз.
5yz	Отклик, говорящий о том, что команда не выполнена и не может быть выполнена вообще.

Значение кода "y" в вышеприведенной таблице может принимать значения от 0 до 5. Значения кодов "y" приведены ниже:

Значение кода-отклика	Описание
x0z	Указывает на синтаксическую ошибку; синтаксис верен, но команда не имеет смысла.
x1z	Указание на необходимость ввода дополнительной информации.
x2z	Отклик, связанный с управлением каналом связи.
x3z	Отклик для команд идентификации пользователя и проверки пароля.
x4z	Функция не определена.
x5z	Отклик, характеризующий состояние файловой системы.

Далее в тексте встречается выражение "анонимное FTP", это подразумевает следующую процедуру (см. также RFC-1635):

ftp> login: *anonymous*
ftp> password: *[ваш полный E-mail адрес]*

ftp> cd <имя_каталога>	(смена каталога)
ftp> *binary*	(если текст, например, архивирован, в противном случае команду выдавать не нужно)
ftp> *get <имя_файла>*	(копирование файла)
ftp> *quit*	(уход из процедуры)

Следует иметь в виду, что некоторые анонимные FTP-серверы (также как, например, GOPHER-серверы) требуют, чтобы ЭВМ, с которой осуществляется ввод, имела не только IP-адрес, но и зарегистрированное в локальном DNS-сервере имя. Эти FTP-серверы, получив запрос, пытаются выяснить имя ЭВМ, так как они ведут "журнал посещений", и в случае неуспеха прерывают сессию. Таким образом, анонимное FTP может считаться таковым лишь условно, в смысле ненужности быть авторизованным на сервере, чтобы иметь к нему доступ. Конкретные примеры кодов статуса обмена для FTP приведены в табл. 4.4.

Таблица 4.4 Коды откликов

Код-отклик	Описание
110	Комментарий
120	Функция будет реализована через nnn минут
125	Канал открыт, обмен данными начат
150	Статус файла правилен, подготавливается открытие канала
200	Команда корректна
211	Системный статус или отклик на справочный запрос
212	Состояние каталога
213	Состояние файла
214	Справочное поясняющее сообщение
220	Слишком много подключений к FTP-серверу (можете попробовать позднее). В некоторых версиях указывает на успешное завершение промежуточной процедуры
221	Благополучное завершение по команде quit
225	Канал сформирован, но информационный обмен отсутствует
226	Закрытие канала, обмен завершен успешно
230	Пользователь идентифицирован, продолжайте

250	Запрос прошел успешно
331	Имя пользователя корректно, нужен пароль
332	Для входа в систему необходима аутентификация
421	Процедура не возможна, канал закрывается
425	Открытие информационного канала не возможно
426	Канал закрыт, обмен прерван
450	Запрошенная функция не реализована, файл не доступен, например, занят
451	Локальная ошибка, операция прервана
452	Ошибка при записи файла (не достаточно места)
500	Синтаксическая ошибка, команда не может быть интерпретирована (возможно, она слишком длинна)
501	Синтаксическая ошибка (неверный параметр или аргумент)
502	Команда не используется (нелегальный тип MODE)
503	Неудачная последовательность команд
504	Команда не применима для такого параметра
530	Система не загружена (not logged in)
532	Необходима аутентификация для запоминания файла
550	Запрошенная функция не реализована, файл не доступен, например, не найден
552	Запрошенная операция прервана, недостаточно выделено памяти

В настоящее время разработаны версии FTP для работы с IPv6 (RFC-2428).

4.4. Протокол TFTP

Некоторым приложениям копирования файлов требуются очень простые реализации, например, для начальной загрузки программного обеспечения и конфигурационных файлов в маршрутизаторы, концентраторы или бездисковые рабочие станции [1, 10].

Простейший протокол пересылки файлов (Trivial File Transfer Protocol — TFTP) используется как очень полезное средство копирования файлов между компьютерами. TFTP передает данные в датаграммах UDP (при реализации в другом стеке протоколов TFTP должен запускаться поверх службы пакетной доставки данных). Для этого не потребуется слишком сложное программное

обеспечение — достаточно только IP и UDP. Особенно полезен TFTP для инициализации сетевых устройств (маршрутизаторов, мостов или концентраторов) [1]. Протокол FTP определен в RFC 959, а TFTP - в RFC 1350.

Характеристики TFTP

- Пересылка блоков данных размером в 512 октетов (за исключением последнего блока).
- Указание для каждого блока простого 4-октетного заголовка.
- Нумерация блоков от 1.
- Поддержка пересылки двоичных и ASCII октетов.
- Возможность чтения и записи удаленных файлов.
- Отсутствие ограничений по аутентификации пользователей.

Один из партнеров по TFTP пересылает нумерованные блоки данных одинакового размера, другой партнер подтверждает их прибытие сигналом АСК Отправитель ожидает АСК для посланного блока до того, как пошлет следующий блок. Если за время тайм-аута не поступит АСК, выполняется повторная отправка того же самого блока. Аналогично, если к получателю не поступят данные за время тайм-аута, он отправляет еще один АСК.

Сеанс TFTP начинается запросами *Read Request* (запрос чтения) или *Write Request* (запрос записи). Клиент TFTP начинает работу после получения порта, посылая Read Request или Write Request на порт 69 сервера. Сервер должен идентифицировать различные номера портов клиентов и использовать их для последующей пересылки файлов. Он направляет свои сообщения на порт клиента. Пересылка данных производится как обмен блоками данных и сообщениями АСК.

Каждый блок (за исключением последнего) должен иметь размер в 512 октетов данных и завершаться EOF (коней файла). Если длина файла кратна 512, то заключительный блок содержит только заголовок и не имеет никаких данных. Блоки данных нумеруются от единицы. Каждый АСК содержит номер блока данных, получение которого он подтверждает.

Улучшенный вариант TFTP разрешает согласование параметров через предварительные запросы чтения и записи. Его основная цель — позволить клиенту и серверу согласовывать между собой размер блока, когда он больше 512 байт (для увеличения эффективности пересылки данных).

Работу протокола TFTP можно описать простым сценарием. После отправки запрашиваемой стороной блока данных она переходит в режим ожидания АСК на посланный блок и, только получив этот АСК, посылает следующий блок данных.

Элементы данных протокола TPTP

В TFTP существуют пять типов элементов данных (см. рис. 4.4):

- Read Request (RRQ, запрос чтения)
- Write Request (WRQ, запрос записи)
- Data (DATA, данные)

- Acknowledgment (ACK, подтверждение)
- Error (ERROR, ошибка)

Сообщение об ошибке указывает на события, подобные таким: "файл не найден" или "для записи файла на диске нет места".

Каждый заголовок TFTP начинается операционным кодом, идентифицирующим тип элемента данных протокола (Protocol Data Unit — PDU). Форматы PDU показаны на рисунке.

Отметим, что длина Read Request и Write Request меняется в зависимости от длины имени файла и полей режима, каждое из которых представляет собой текстовую строку ASCII, завершенную нулевым байтом. В поле режима могут присутствовать netascii (сетевой ASCII) или octet (октет).

Read Request

2 байта	Строка	1 байт	Строка	1 байт
Операционный код = 1	Имя Файла	0	Режим	0

Write Request

2 байта	Строка	1 байт	Строка	1 байт
Операционный код = 2	Имя Файла	0	Режим	0

Data

2 байта	2 байта	
Операционный код = 3	Номер блока	Данные

Acknowledgment

2 байта	2 байта
Операционный код = 4	Номер блока

Error

2 байта	1 байт	Строка	1 байт
Операционный код = 5	Код ошибки	Сообщение об ошибке	0

Рис. 4.4 Форматы элементов данных протокола TFTP

Контрольные вопросы

1. Для чего предназначены протоколы POP3 и SMTP?
2. Какие по умолчанию порты зарезервированы под серверы POP3 и SMTP?
3. Есть ли авторизация в протоколах POP3 и SMTP?
4. Что делает команда *noop* протоколов POP3 и SMTP и для чего она может быть использована?
5. Сколько портов использует FTP-сервер? Назовите их номера (выделяемые по умолчанию) и назначение?
6. Чем отличаются режимы *active* и *passive* протокола FTP?
7. В чем отличия возможностей протоколов FTP и TFTP?
8. Сколько типов элементов данных протокола TFTP?
9. Какой по умолчанию порт зарезервирован под сервер TFTP?
10. Опишите механизм работы клиента-сервера по протоколу TFTP?
11. Какую информацию содержит блок подтверждения TFTP?
12. Как будет представлена информация, передаваемая по протоколу FTP командой *PORT* для клиента с портом 6987 и IP адресом 134.55.10.129?
13. Расшифруйте запись *PORT 131,132,133,134,135,136*?

ГЛАВА 5. РАЗРАБОТКА СЕТЕВЫХ ПРИЛОЖЕНИЙ

5.1. Гнезда и интерфейс транспортного уровня

В системе BSD UNIX 4.2 были впервые введены гнезда (sockets), которые предоставляют независимые от протокола услуги по организации сетевого интерфейса и способны обеспечить работоспособность методов IPC в масштабах локальной сети [11-13]. В частности, гнезда могут работать как с протоколом TCP, так и с протоколом UDP. Обращаться к гнезду можно по IP-адресу хост-машины и номеру порта. Заданный таким образом адрес уникален в масштабах всей Internet, так как для каждой машины комбинация адреса и номера порта уникальна. Следовательно, два процесса, выполняемых на отдельных машинах, могут взаимодействовать друг с другом через гнезда. Гнезда нашли широкое применение во многих сетевых приложениях.

Различают гнезда *с установлением соединения* (т.е. адреса гнезд отправителя и получателя выясняются заранее, до передачи сообщений между ними) и *без установления соединения* (адреса гнезд отправителя и получателя передаются с каждым сообщением, посылаемым из одного процесса в другой). В зависимости от того, к какому домену принадлежит гнездо, используются разные форматы адресов гнезд и базовые транспортные протоколы. При использовании гнезд обычно применяются следующие стандартные домены: AF_UNIX (формат адреса — путевое имя UNIX) и AF_NET (формат адреса — хост-имя и номер порта) [11].

Для каждого гнезда назначается тип, посредством которого определяется способ передачи данных между двумя гнездами. Если тип гнезда — *виртуальный канал* (virtual circuit), то данные передаются последовательно, с достаточной степенью надежности и не дублируются. Если тип гнезда — *датаграмма* (datagram), то условие последовательности пересылки данных не выполняется и надежность их передачи низкая. Тип гнезда с установлением соединения — как правило, виртуальный канал, а тип гнезда без установления соединения — датаграмма. Датаграммные гнезда обычно работают быстрее, чем виртуальные каналы (потоковые гнезда), и используются в приложениях, где быстродействие важнее, чем надежность.

Гнездо каждого типа поддерживает один или несколько транспортных протоколов, однако в любой UNIX-системе для любого гнезда всегда указывается протокол по умолчанию. Протокол по умолчанию для виртуального канала — TCP, а для датаграммы — UDP.

Гнезда, которые используются для связи компьютеров друг с другом, должны быть одного типа и относиться к одному домену. Кроме того, гнезда с установлением соединения взаимодействуют по схеме клиент/сервер. Серверному гнезду назначается общеизвестный адрес, и оно непрерывно ожидает прибытия клиентских сообщений. Клиентский процесс посылает сообщения на сервер по объявленному адресу серверного гнезда. Назначать адреса клиентским гнездам не нужно, потому что обычно ни один серверный процесс сообщения клиентам таким способом не передает.

Гнезда без установления соединения, с другой стороны, взаимодействуют по одноранговой схеме: каждому гнезду назначается адрес, и процесс может посылать сообщения другим процессам, используя адреса их гнезд.

Интерфейсы прикладного программирования (application programming interface — API) гнезд перечислены ниже (см. табл. 5.1.):

Таблица 5.1

API гнезд	Назначение
Socket	Создает гнездо заданного типа и с указанным протоколом для конкретною домена
bind	Присваивает гнезду имя
listen	Задает количество ожидающих клиентских сообщений, которые можно поставить в очередь к одному серверному гнезду
accept	Принимает запрос на соединение от клиентского гнезда
connect	Посылает запрос на соединение в серверное гнездо
Send, sendto	Передает сообщение в удаленное гнездо
recv,recvfrom	Принимает сообщение из удаленного гнезда
shutdown	Закрывает гнездо для чтения и/или записи

Последовательность вызовов API гнезд, которые устанавливают между клиентом и сервером соединение типа виртуальный канал, представлена на рисунке 5.1.

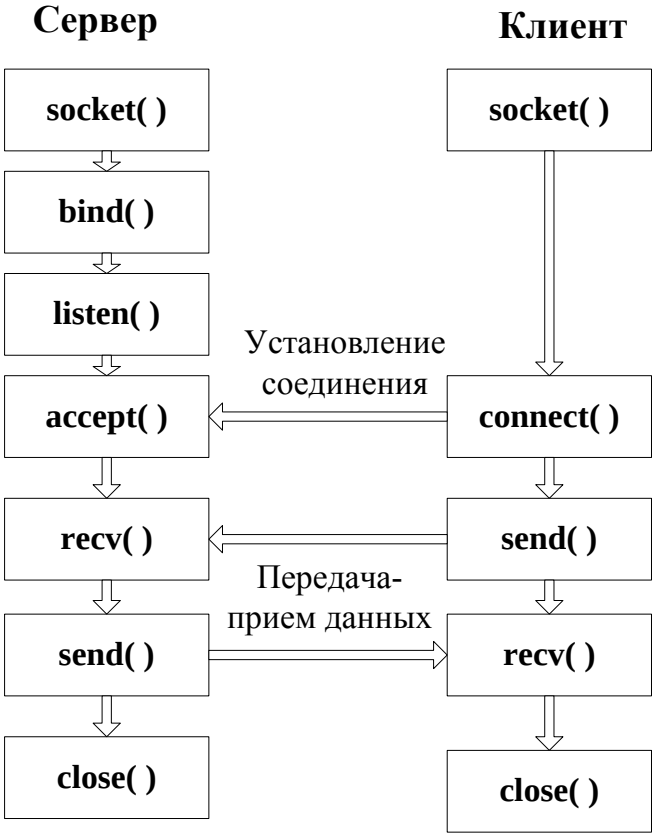


Рис. 5.1 Последовательность вызовов API для серверного и клиентского гнезд

Чтобы понять смысл использования этих API, представьте, что гнездо — это телефонный аппарат. API *socket* предназначен для покупки телефона в магазине [11]. API *bind* присваивает телефону номер. API *listen* просит вашу телефонную компанию подключить телефон к сети. API *connect* звонит кому-то с вашего телефона. API *answer* отвечает на телефонный звонок. API *send* разговаривает по телефону. Наконец, API *shutdown* кладет трубку после завершения разговора. Чтобы отказаться от услуг телефонной компании, используйте API *close* с дескриптором гнезда, возвращенным из вызова функции *socket.*

На стороне клиента процесс вызывает функцию *socket* для установки телефона. Затем он вызывает функцию *connect* и с ее помощью набирает номер сервера, после чего посредством вызова функций *send* и *recv* общается с сервером. По окончании разговора процесс вызывает функцию *shutdown,* которая дает сигнал отбоя, и функцию *close,* уничтожающую "телефон".

Последовательность вызовов API, создающих для обеспечения межпроцессного взаимодействия датаграммные гнезда, изображена на рисунке 5.2.

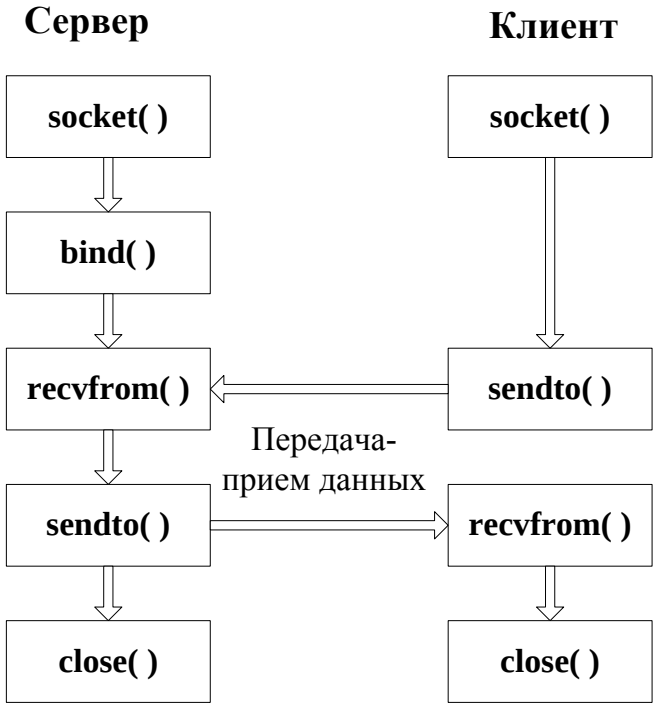


Рис. 5.2 Последовательность вызовов API, обеспечивающих взаимодействие процессов через датаграммные гнезда

Работать с датаграммными гнездами достаточно легко. Процесс вызывает функцию *socket,* которая создает гнездо, а затем с помощью функции *bind* присваивает гнезду имя. Затем процесс вызывает функцию *sendto* для передачи сообщений в другие процессы. Каждое сообщение снабжается адресом гнезда получателя. Процесс получает также сообщения из других процессов посредством вызова функции *recvfrom.* Каждое полученное сообщение содержит адрес гнезда отправителя, что позволяет процессу безошибочно отправить ответ.

По завершении межпроцессного взаимодействия процесс вызывает функцию *close,* которая удаляет гнездо. Вызывать функцию *shutdown* не нужно, потому что виртуальный канал, обеспечивающий взаимодействие с другими процессами, не организовывался.

В последующих разделах более подробно рассматривается синтаксис API гнезд и методика их использования.

5.1.1. Функция socket

Прототип функции *socket* выглядит следующим образом [11]:

```
#include <sys/types.h>
#include <sys/socket.h>

int socket ( int domain, int type, int protocol );
```

Эта функция создает для указанного пользователем домена гнездо заданного типа и с указанным протоколом.

Аргумент *domain* определяет правила именования гнезда и формат адреса, используемые в протоколе. Широко применяются такие домены, как *AF_UNIX* (домен UNIX) и *AF_NET* (Internet-домен).

Аргумент *type* задает тип гнезда. Возможные значения этого аргумента приведены в таблице 5.2.

Таблица 5.2

Тип гнезда	Смысл
SOCK_STREAM	Сообщения передаются в виде упорядоченного двунаправленного потока байтов с высокой степенью надежности и предварительным установлением соединения
SOCK_DGRAM	Межпроцессное взаимодействие обеспечивается с помощью датаграмм. Сообщения передаются быстро (как правило, без установления соединения), но с низкой степенью надежности
SOCK_SEQPACKET	Двунаправленная последовательная высоконадежная передача сообщений фиксированной максимальной длины с предварительным установлением соединения

Аргумент *protocol* указывает конкретный протокол, который следует использовать с данным гнездом. Фактическое значение этого аргумента зависит от значения аргумента *domain.* Как правило, оно устанавливается в 0, и ядро само выбирает для указанного домена соответствующий протокол.

В случае успешного выполнения рассматриваемая функция возвращает целочисленный дескриптор гнезда, а в случае неудачи возвращает -1. Отметим, что дескриптор гнезда — это то же самое, что и дескриптор файла; он занимает одну ячейку таблицы дескрипторов файлов в вызывающем процессе.

5.1.2. Функция bind

Прототип функции *bind* выглядит следующим образом [11]:

```
#include <sys/types.h>
#include <sys/socket.h>

int bind ( int sid, struct sockaddr* addr_p, int len );
```

Эта функция присваивает гнезду имя. Гнездо обозначается аргументом *sid*, значение которого, возвращенное функцией *socket*, представляет собой дескриптор гнезда. Аргумент *addr_p* указывает на структуру, содержащую имя, которое должно быть присвоено гнезду. Аргумент *len* задает размер структуры, на которую указывает аргумент *addr_p*.

В каждом домене используется своя структура объекта, на который указывает аргумент *addr_p*. В частности, в случае гнезда домена UNIX присваиваемое имя представляет собой путевое UNIX-имя, а структура объекта, на который указывает аргумент *addr_p*, имеет такой вид:

```
struct sockaddr {
  short       sun_family;
  char        sun_path[];
};
```

Здесь полю *sun_family* следует присвоить значение AF_UNIX, а поле *sun_path* должно содержать путевое UNIX-имя. При успешном выполнении вызова *bind* в файловой системе создается файл с именем, заданным в поле *sun_path*. Если гнездо больше не нужно, этот файл следует удалить с помощью API *unlink*.

В случае гнезда домена Internet присваиваемое имя состоит из хост-имени машины и номера порта, а структура объекта, на который указывает аргумент *addr_p*, имеет такой вид:

```
struct sockaddr_in {
  short       sin_family;
  u_short   sin_port;
  struct in_addr   sin_addr;
};
```

Здесь полю *sin_family* следует присвоить значение AFINET. Поле *sin_port* — это номер порта, а поле *sin_addr* — имя хост-машины, для которой создается гнездо. Структура *sockaddr_in* определяется в заголовке <neti-net/in.h>.

При успешном выполнении эта функция возвращает 0, а в случае неудачи возвращает -1.

5.1.3. Функция listen

Прототип функции *listen* выглядит следующим образом [11]:

```
#include <sys/types.h>
#include <sys/socket.h>

int listen ( int sid, int size);
```

Эта функция вызывается серверным процессом для создания гнезда, ориентированного на установление соединения (типа SOCK_STREAM или SOCK_SEQPACKET).

Аргумент *sid* представляет собой дескриптор гнезда, возвращенный функцией *socket.* Аргумент *size* задает максимальное число запросов на установление соединения, которые могут быть поставлены в очередь к данному гнезду. В большинстве UNIX-систем максимально допустимое значение аргумента *size* — 5.

При успешном выполнении эта функция возвращает 0, а в случае неудачи возвращает -1.

5.1.4. Функция connect

Прототип функции *connect* выглядит так [11]:

```
#include <sys/types.h>
#include <sys/socket.h>

int connect ( int sid, struct sockaddr* addr_p, int len );
```

Эта функция вызывается в клиентском процессе для установления соединения с серверным гнездом.

Аргумент *sid* представляет собой дескриптор гнезда, возвращенный функцией *socket.* В BSD 4.2 и 4.3 имя гнезда, указанное аргументом *sid,* совпадает с именем используемого транспортного протокола. В System V.4 имя гнезду присваивается транспортным протоколом.

Аргумент *addr_p* — это указатель на адрес объекта типа *struct sockaddr,* хранящего имя серверного гнезда, с которым должно быть установлено соединение. Фактически структура этого объекта зависит от домена, на основе которого создается гнездо. Возможный формат — *struct sockaddr* (для домена UNIX) или *struct sockaddr_in* (для домена Internet).

Аргумент *len* задает размер объекта (в байтах), на который указывает аргумент *addr_p.*

Если *sid* обозначает потоковое гнездо, то между клиентским и серверным гнездами устанавливается соединение с использованием виртуального канала.

Потоковое гнездо клиента может соединяться с гнездом сервера только один раз. Если *sid* обозначает датаграммное гнездо, то для всех последующих вызовов функции *send,* осуществляемых через это гнездо, устанавливается

адрес по умолчанию. Датаграммное гнездо может соединяться с гнездом сервера многократно, изменяя установленные по умолчанию адреса. Путем соединения с гнездом, имеющим NULL-адрес, датаграммные гнезда могут разорвать соединение.

При успешном выполнении эта функция возвращает 0, а в случае неудачи — 1.

5.1.5. Функция accept

Прототип функции *accept* выглядит следующим образом [11]:

```
#include <sys/types.h>
#include <sys/socket.h>

int accept ( int sid, struct sockaddr* addr_p, int len_p);
```

Эта функция вызывается в серверном процессе для установления соединения с клиентским гнездом (которое делает запрос на установление соединения посредством вызова функции *connect).*

Аргумент *sid* представляет собой дескриптор гнезда, возвращенный функцией *socket.* Аргумент *addr_p* — это указатель на адрес объекта типа *struct sock;* в нем хранится имя клиентского гнезда, с которым устанавливает соединение серверное гнездо.

Аргумент *len_p* изначально устанавливается равным максимальному размеру объекта, указанному аргументом *addr_p.* При возврате он содержит размер имени клиентского гнезда, на которое указывает аргумент *addr_p.*

Если аргумент *addr_p* или аргумент *len_p* имеет значение NULL, эта функция не передает имя клиентского гнезда обратно в вызывающий процесс.

В случае неудачи рассматриваемая функция возвращает -1. В противном случае она возвращает дескриптор нового гнезда, с помощью которого серверный процесс может взаимодействовать исключительно с данным клиентом.

5.1.6. Функция send

Прототип функции *send* выглядит следующим образом [11]:

```
#include <sys/types.h>
#include <sys/socket.h>

int send ( int sid, const char* buf, int len, int flag);
```

Эта функция передает содержащееся в аргументе *buf* сообщение длиной *len* байтов в гнездо, заданное аргументом *sid* и соединенное с данным гнездом.

Аргументу *flag* обычно присваивается значение 0, но он может иметь и значение MSG_OOB. В этом случае сообщение, содержащееся в *buf* должно быть передано как высокоприоритетное (out-of-band message).

Через гнезда можно передавать сообщения двух типов: обычные и высокоприоритетные. По умолчанию все сообщения, передаваемые гнездом, являются обычными, если явно не указано, что они высокоприоритетные. Если из гнезда передается более одного сообщения одного типа, другое гнездо принимает их по алгоритму FIFO. Гнездо-получатель может выбрать тип сообщений, которые оно хотело бы получать. Сообщения с высоким приоритетом следует использовать только в экстренных случаях.

Если процесс пользуется гнездом с установлением соединения или гнездом без установления соединения, для которого указан адрес получателя по умолчанию (посредством вызова функции *connect),* он может передавать обычные сообщения с помощью либо API *send,* либо API *write.* При этом функции *send* и *sendto* можно использовать для передачи сообщений нулевой длины, а *write* — нельзя. Кроме того, в BSD 4.2 и 4.3 функция *write* при обращении к гнезду, соединение с которым не установлено, дает сбой.

В случае неудачи эта функция возвращает -1; в случае успешного выполнения возвращается число переданных байтов данных.

5.1.7. Функция sendto

Прототип функции *sendto* выглядит следующим образом [11]:

```
#include <sys/types.h>
#include <sys/socket.h>

int sendto (int sid, const char* buf, int len, int flag,
struct sockaddr* addr_p, int* Ien_p );
```

Эта функция делает то же самое, что и API *send,* только вызывающий процесс указывает также адрес гнезда-получателя (в аргументах *addr_p* И *len_p).*

Аргументы *sid, buf, len* и *flag* — те же самые, что в API *send.* Аргумент *addr_p* — это указатель на объект, который содержит имя гнезда-получателя. Аргумент *len_p* содержит число байтов в объекте, на который указывает аргумент *addr_p.*

В случае неудачи данная функция возвращает -1; в случае успешного выполнения возвращается число переданных байтов данных.

5.1.8. Функция recv

Прототип функции *recv* выглядит следующим образом [11]:

```
#include <sys/types.h>
#include <sys/socket.h>

int recv ( int sid, char* buf, int len, int flag );
```

Эта функция принимает сообщение через гнездо, указанное в аргументе *sid.* Принятое сообщение копируется в буфер *buf,* а максимальный размер *buf* задается аргументом *len.*

Если в аргументе *flag* указан флаг MSG_OOB, то приему подлежит высокоприоритетное сообщение. В противном случае ожидается обычное сообщение. Кроме того, в аргументе *flag* может быть указан флаг MSG_PEEK, означающий, что процесс желает "взглянуть" на полученное сообщение, но не собирается удалять его из потокового гнезда. Такой процесс может повторно вызвать функцию *recv* для приема сообщения позже.

Если процесс пользуется гнездом (с установлением соединения или без установления соединения), для которого указан адрес получателя по умолчанию (посредством вызова API *bind),* он может принимать обычные сообщения через это гнездо с помощью либо API *recv,* либо API *read.* В BSD 4.2 и 4.3 функция *read* при использовании с гнездом, с которым не установлено соединение, дает сбой. В System V.4 в подобном случае функция *read* возвращает нулевое значение в блокирующем режиме и -1 в неблокирующем.

В случае неудачи функция *recv* возвращает -1; в случае успешного выполнения возвращается число байтов данных, записанных в буфер *buf.*

5.1.9.Функция recvfrom

Прототип функции *recvfrom* выглядит следующим образом [11]:

```
#include <sys/types.h>
#include <sys/socket.h>

int recvfrom ( int sid, const char* buf, int ien, int flag,
struct sockaddr* addr_p, int* len_p );
```

Эта функция делает то же самое, что и API *recv,* только при ее вызове задаются аргументы *addr_p* и *len_p,* позволяющие узнать имя гнезда-отправителя.

Аргументы *sid, buf len* и *flag* — те же самые, что в API *recv.* Аргумент *addr_p* — это указатель на объект, который содержит имя гнезда-отправителя.

Аргумент *len_***p** сообщает число байтов в объекте, на который указывая аргумент *addr_p.*

В случае неудачи функция *recvfrom* возвращает -1; в случае успешной: выполнения возвращается число принятых байтов данных.

5.1.10.Функция shutdown

Прототип функции *shutdown* выглядит следующим образом [11]:

```
#include <sys/types.h>
#include <sys/socket.h>

int shutdown ( int sid, int mode );
```

Данная функция закрывает соединение между серверным и клиентским гнездами.

Аргумент *sid* — это дескриптор гнезда, возвращенный функцией *socket* Аргумент *mode* задает режим закрытия. Возможные его значения приведены в таблице 5.3.

Таблица 5.3

Режим	Смысл
0	Закрывает гнездо для чтения. При попытке продолжить чтение будут возвращаться нулевые байты (EOF)
1	Закрывает гнездо для записи. Дальнейшие попытки передать данные в это гнездо приведут к выдаче кода неудачного завершения -1
2	Закрывает гнездо для чтения и записи. Дальнейшие попытки передать данные в это гнездо приведут к выдаче кода неудачного завершения -1, а при продолжении чтения будет возвращаться нулевое значение (EOF)

В случае неудачи данная функция возвращает -1, а в случае успешного выполнения — нуль.

5.2.Разработка клиент-серверных приложений

5.2.1. Однопоточная серверная программа TCP

Рассмотрим подробно пример серверной программы [1] (см. листинг 1). Сервер предназначен для непрерывной работы. Он будет выполнять следующие действия:

1. Запрашивать у *socket* создание главного TCB и возвращать значение дескриптора socket, который будет идентифицировать этот TCB в последующих вызовах.
2. Вводить локальный адрес сервера socket в структуру данных программы.
3. Запрашивать *связывание,* при котором в TCB копируется локальный адрес socket.
4. Создавать очередь, которая сможет хранить сведения о пяти клиентах.
   Оставшиеся шаги повторяются многократно.
5. Ожидать запросов от клиентов. Когда появляется клиент, создавать для него новый TCB на основе копии главного TCB и записи в него адреса socket клиента и других параметров.
6. Создавать дочерний процесс для обслуживания клиента. Дочерний процесс будет наследовать новый TCB и обрабатывать все дальнейшие операции по связи с клиентом
   (ожидать сообщений от клиента, записывать их и завершать работу).

Листинг 1. Однопоточная серверная программа TCP

```
 /* tcpserv.c * Для запуска программ ввести "tcpserv". •/
/* Сначала включить набор стандартных заголовочных файлов.

#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#Include <netinet/in.h>
#include <netdb.h>
#include <errno.h>

main()
int sockMain,  sockClient, length, child;
struct sockaddr_in servAddr;

/* 1. Создать главный блок управления пересылкой. */

if((sockMain=socket(AF_INET, SOCK_STREAM,0))<0){
    perror("Сервер не может открыть главный socket.");
    exit(1);
}

/* 2.   Создать структуру данных для хранения локальных
    IP-адресов и портов, которые будут использованы.
    Предполагается прием клиентских соединений от любых
    локальных IP-адресов (INADDR_ANY). Поскольку данный
    сервер не применяет общеизвестный порт, установить
    port = 0. Это позволит связать вызов с присвоением
    порта серверу и записать порт в TCB. */

bzero((char *) &servAddr, sizeof(servAddr));
servAddr.sin_family=AF_INET;
servAddr.sin_addr.s_addr=htonl(INADDR_ANY);
servAddr.sin_port=0;

/* 3.   Связать запрос,   выбор номера порта и запись его
    в TCB. */

if (bind(sockMain,&servAddr, sizeof(servAddr))){
    perror("Связывание сервера неудачно."); exit(1);
}
/* Чтобы увидеть номер порта,  следует использовать
функцию getsockname(), чтобы скопировать порт в servAddr.
    */
```

```
length=sizeof(servAddr);
if (getsockname(sockMain,&servAddr, &length)){
    perror("Вызов getsockname неудачен."); exit(1);
}
printf("СЕРВЕР: номер порта - % d\n",
    ntohs(servAddr.sin_port))

/* 4. Создать очередь для хранения пяти клиентов.

listen(sockMain, 5) ;

/* 5. Ожидать клиента. При разрешении возвратить новый
    дескриптор socket, который должен использоваться
    клиентом. */

for(;;){
if ((sockClient = accept(sockMain,0,0)) < 0){
    perror("Неверный socket для клиента."); exit(1);
}
BuffWork(sockClient);
close(sockClient); }

/* Дочерний процесс читает один поступивший буфер,
    распечатывает
сообщение и завершается. */

#define BUFLEN 81
int BuffWork(sockClient)
int sockClient;
{
char buf[BUFLEN];
int msgLength;

/* 6. Опустошить буфер.  Затем вывести recv для получения
    сообщения от клиента. */

bzero(buf, BUFLEN);
if((msgLength = recv(sockClient, buf, BUFLEN, 0))<0){
    perror("Плохое получение дочерним процессом.");
    exit(1);
}
printf ("SERVER: Socket для клиента - %d\n", sockClient);
printf("SERVER: Длина сообщения - %d\n", msgLength);
printf  ("SERVER: Сообщение:  %s\n\n", buf);
}
```

Вызовы в серверной программе TCP

1. *sockMain = socket (AF_INET, SOCK_STREAM, 0);*

Вызов socket имеет форму:

*дескриптор*_socket=socket(адрес_домена, тип_коммуникации, протокол)

AF_INET указывает на семейство адресов Интернета. *SOCK_STREAM* запрашивает socket TCP. Эта переменная должна иметь значение *SOCK_DGRAM*, чтобы создать socket UDP, а *SOCK_RAW* служит для непосредственного обращения к IP.

Не нужно явно определять никакую другую информацию протокола для TCP (или для UDP). Однако параметр *protocol* необходим для интерфейса с необработанными данными, а также для некоторых протоколов из других семейств, использующих socket.

2. *struct sockaddrin servAddr;*

```
bzero((char*)&servAddr, sizeof(servAddr));
servAddr.sinFamily = AF_INET;
servAddr.sin_addr.s_addr=htonl(INADDR_ANY);
servAddr.sin_port = 0;
```

Программная структура *servAddr* используется для хранения адресной информации сервера. Вызов *bzero* ( ) инициализирует *servAddr*, помещая нули во все параметры. Первая переменная в структуре *servAddr* указывает, что остальная часть значений содержит данные семейства адресов Интернета.

Следующая переменная хранит локальный IP-адрес сервера. Например, если сервер подключен к локальной сети Ethernet и к сети X.25, может потребоваться ограничить доступ клиентов через интерфейс Ethernet. В данной программе об этом можно не беспокоится. *INADDR_ANY* означает, что клиенты могут соединяться через любой интерфейс.

Функция *htonl ()* имеет полное название host-to-network-long. Она применяется для преобразования 32-разрядных целых чисел локального компьютера в формат Интернета для 32-разрядного адреса IP. Стандарты Интернета предполагают представление целых чисел с наиболее значимым байтом слева. Такой стиль именуется Big Endian (стиль "тупоконечников"). Некоторые компьютеры хранят данные, располагая слева менее значимые байты, т.е. в стиле Little Endian ("остроконечников"). Если локальный компьютер использует стиль Big Endian, *htonl* () не будет выполнять никакой работы.

Если сервер взаимодействует через общеизвестный порт, номер этого порта нужно записать в следующую переменную. Поскольку мы хотим, чтобы операционная система сама присвоила порт для нашей тестовой программы, мы вводим нулевое значение.

bind (sockMain, &servAddr, sizeof (servAddr));

getsockname (sockMain, &servAddr, &length);

Вызов *bind* имеет форму:

возвращаемый_код = bind(дескриптор_socket, адресная_структура, длина_адресной_структуры)

Если адресная структура идентифицирует нужный порт, bind попытается получить его на сервере. Если переменная порта имеет значение 0, *bind* получит один из неиспользованных портов. Функция *bind* позволяет ввести номер порта и IP-адрес в TCB. Вызов *getsockname* имеет форму:

```
возвращаемый_код = getsockname(дескриптор_socket,
адресная_структура, длина_адресной_структуры)
```

Мы запросили у *bind* выделение порта, но эта функция не сообщает нам, какой именно порт был предоставлен. Для выяснения этого нужно прочитать соответствующие данные из TCB (*Transmission Control Block* - блок управления пересылкой, созданный при вызове *socket)* [1, стр.335-336]. Функция *getsockname()* извлекает информацию из TCB и копирует ее в адресную структуру, где можно будет прочитать эти сведения. Номер порта извлекается и выводится следующим оператором:

```
printf ("SERVER: Номер порта %d\n",
ntohs(servAddr.sin_port));
```

Функция *ntohs ()* имеет полное название *network-to-host-short* и служит для преобразования номера порта из порядка следования байт в сети в локальный порядок следования байт на хосте.

3. *listen (sockMain,5);*

Вызов *listen* применяется для ориентированных на соединение серверов и имеет форму:

```
возвращаемый_код = listen(дескриптор_socket,
размер_очереди)
```

Вызов *listen* указывает, что это будет пассивный *socket*, и создает очередь требуемого размера для хранения поступающих запросов на соединения.

4. *sockClient = accept (sockMain, 0, 0);*

Вызов *accept* имеет форму:

```
Новый_дескриптор_socket = accept(дескриптор_socket,
клиентская_адресная_структура,
длина_клиентской_адресной_структуры)
```

По умолчанию вызов блокируется до соединения клиента с сервером. Если указана переменная *клиентская_адресная_структура,* после соединения клиента в эту структуру будут введены IP-адрес и порт клиента. В этом примере программы не проверяются IP-адрес и номер порта клиента, а просто два последних поля параметра заполняются нулями.

5. *close (sockClient);*

Этот вызов выполняется из родительской части программы. Когда родительский процесс закрывает *sockClient,* дочерний процесс все еще имеет доступ к *socket*.

6. *msgLength = recv(sockClient,buf, BUFLEN, 0));*
close (sockClient);

Вызов *recv* имеет форму:

```
длина_сообщения = recv(дескриптор_socket, буфер,
длина_буфера, флаги)
```

По умолчанию вызов *recv* блокированный. Функции *fcnt ()* или *iocntl()* позволяют изменить статус *socket* на неблокированный режим.

После получения данных дочерним процессом и вывода сообщения на печать, доступ к *sockClient* закрывается. Это заставит соединение перейти в фазу закрытия.

5.2.2. Клиентская программа TCP

Клиент соединяется с сервером, посылает одно сообщение, и далее работа программы завершается (см. листинг 2) [1]. Для запуска программы конечный пользователь должен ввести имя хоста сервера, номер порта и сообщение, которое будет послано на этот сервер.

Листинг 2. Клиентская программа TCP

```
/* tcpclient.c
Перед запуском клиента должен быть запущен сервер.
    Производится
поиск порта сервера. Для запуска клиента нужно ввести:
tcpclient имя_хоста порт сообщение */
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#include <errno.h>

    main(argc, argv)  /* Клиентская программа имеет
    входные аргументы. */
int argc; char* argv [ ]; {
int sock;
struct sockaddr_in servAddr;
struct hostent *hp, *gethostbyname();

/* Аргументами будут 0:имя_программы,   1:имя_хоста,
    2:порт,  3:сообщение */

if(argc < 4)
{printf("ВВЕСТИ tcpclient имя_хоста порт сообщение\n");
    exit(1) ;
}
/* 1. Создание TCB. */
if((sock=socket(AF_INET,SOCK_STREAM, 0))<0){perror("Не
    могу получить socket\n"); exit(1);
}
```

```
/* 2. Заполнить поля адреса и порта сервера в servAddr.
    Сначала заполнить нулями адресную структуру. Затем
    получить IP-адрес для данного имени хоста и ввести
    его в адресную структуру. Наконец ввести номер порта,
    взяв его из argv[2]. */

bzero((char *)&servAddr,sizeof(servAddr));
servAddr.sin_family = AF_I.NET;
hp=gethostbyname(argv [1]);
bcopy(hp->h_addr, &servAddr.sin_addr, hp->h_length);
servAddr.sinport=htons(atoi(argv[2]) );

/* 3.   Соединиться с сервером. Вызывать bind не нужно.
*    Система присвоит свободный порт во время выполнения
    соединения. */
if(connect(sock, &servAddr,sizeof(servAddr))<0){
    perror("Клиент не может соединиться.\n"); exit(1);
}

/* 4.   Клиент анонсирует свою готовность послать
    сообщение.
Сообщение отправляется, и распечатывается последняя
    строка. */

printf  ("CLIENT: Готов к пересылке\n") ;
if(send(sock,argv[3],strlen(argv[3]),0)<0){
    perror("Проблемы с пересылкой.\n"); exit(1);
}
printf("CLIENT: Пересылка завершена. Счастливо
    оставаться.\n"); close(sock); exit(0);
}
```

Вызовы в клиентской программе TCP

1. *sock — socket(AF_INET, SOCK_STREAM, 0);*

Клиент создает блок управления пересылкой ("socket") так же, как это делал сервер.

2. Сервер должен инициализировать адресную структуру для использования в *bind.*

Эта структура содержит локальный IP-адрес и номер порта сервера. Клиент также инициализирует адресную структуру, хранящую те же сведения. Эта структура будет использоваться в вызове *connect* для указания точки назначения.

Вызов *bzero( )* помещает нули в *servAddr* — адресную структуру сервера. Еще раз мы трактуем семейство адресов как Интернет.

Затем нужно преобразовать введенное пользователем имя хоста в IP-адрес. Это делает функция *gethostbyname*, которая возвращает указатель на структуру *hostent*, содержащую имя сервера и IP-адрес.

Функция *bcopy* применяется для копирования IP-адреса (который находится в *hp->h_addr)* в *servAddr*.

Второй введенный конечным пользователем аргумент определял порт сервера. Он читался как текстовая строка ASCII, поэтому ее сначала нужно преобразовать в целое число через *atoi()*, а затем изменить порядок следования байт через *htons()*. Наконец номер порта копируется в адресную переменную из *servAddr*.

```
bzero((char*)&servAddr, sizeof(servAddr));
servAddr.sin_famiiy = AF_INET;
hp = gethostbyname  (argv [1]);
bcopy(hp->h_addr, &servAddr.sin_addr, hp->h_length);
servAddr.sin_port = htons(atoi(argv[2]));
```

3. *connect (sock, AservAddr, sizeof (servAddr));*

Вызов *connect* имеет форму:

```
connect(дескриптор_socket, адресная_структура,
длина_адресной_структуры)
```

Клиент откроет соединение с сервером, IP-адрес и порт которого хранятся в адресной структуре.

4. *send (sock, argv\3], strlen(argvs [3]), 0);*

Вызов *send* имеет форму:

```
возвращаемый_код = send(дескриптор_socket, буфер,
длина_буфера, флаги)
```

Отметим, что введенный конечным пользователем третий аргумент (который появляется в программе как *argv [3]*) — это текст отправляемого сообщения. Обычно флаги используются для сообщения о срочных данных. В нашем случае параметры флагов установлены в 0.

5. *close (sock);*

Клиент выполняет *close* для закрытия соединения.

5.2.3. Пример реализации многопотоковой передачи по протоколу SCTP

Рассмотрим пример реализации серверного и клиентского приложений [7]. Исходный текст программы демонстрирует возможности протокола SCTP по многопотоковой передаче данных.

В этом примере сервер реализует одну из форм протокола точного времени, сообщая текущее время подключенному клиенту. Однако для демонстрации возможностей протокола SCTP, передается местное время по потоку 0 и время по Гринвичу (GMT) в потоке 1. Этот простой пример позволяет продемонстрировать интерфейс API потокового взаимодействия.

На рисунке 5.3 демонстрируется весь процесс взаимодействия: показан не только поток данных приложения по API сокетов, но и взаимосвязи между клиентом и сервером.

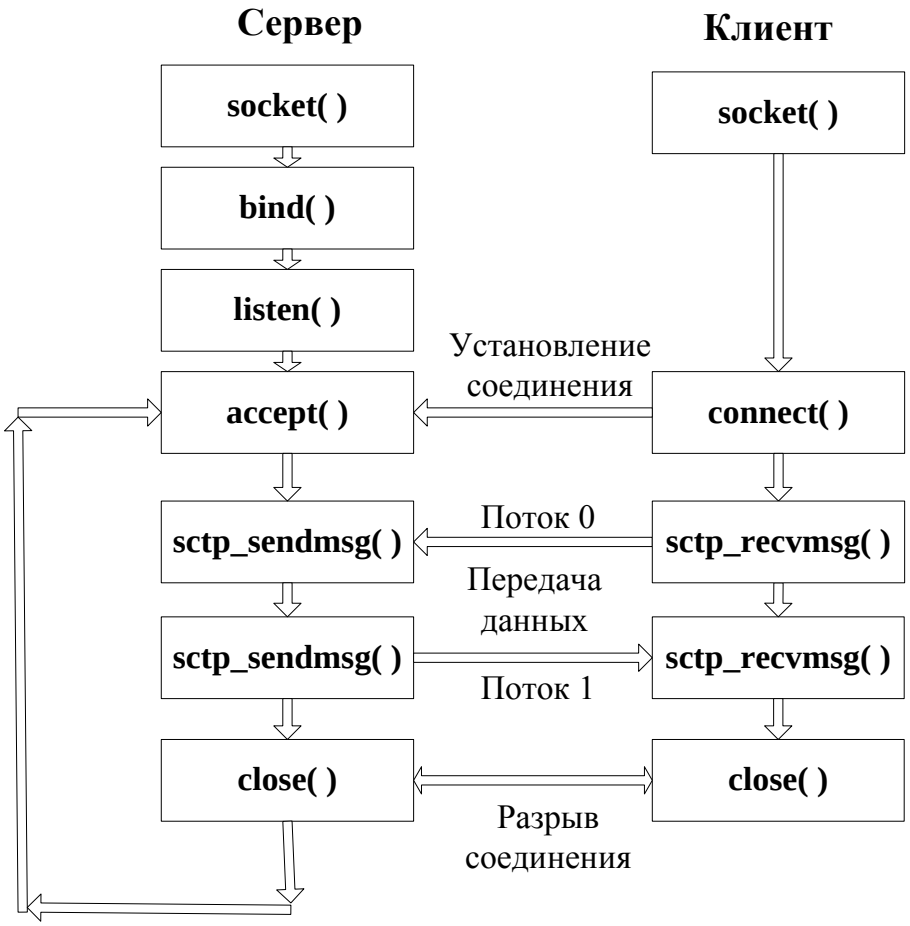


Рис. 5.3 Функции сокетов, используемые при многопотоковой реализации сервера и клиента точного времени

Сервер точного времени

Исходный код приложения многопоточного сервера точного времени показан в листинге 3.

Листинг 3. Сервер точного времени для протокола SCTP, использующий многопотоковую передачу

```
int main()
{
  int listenSock, connSock, ret;
  struct sockaddr_in servaddr;
  char buffer[MAX_BUFFER+1];
  time_t currentTime;

  /* Создание сокета SCTP в стиле TCP */
  listenSock = socket( AF_INET, SOCK_STREAM, IPPROTO_SCTP
    );

  /* Принимается соединение с любого интерфейса */
  bzero( (void *)&servaddr, sizeof(servaddr) );
```

```
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = htonl( INADDR_ANY );
servaddr.sin_port = htons(MY_PORT_NUM);

/* Адрес привязки – любой, порт - MY_PORT_NUM */
ret = bind( listenSock,
             (struct sockaddr *)&servaddr,
  sizeof(servaddr) );

/* Сокет сервера переводится в состояние ожидания
  соединения */
listen( listenSock, 5 );

/* Цикл работы сервера... */
while( 1 ) {

  /* Ожидание соединения клиента */
  connSock = accept( listenSock,
                     (struct sockaddr *)NULL, (int
  *)NULL );

  /* Соединение с новым клиентом */

  /* Выясняется текущее время */
  currentTime = time(NULL);

  /* Посылается текущее время по потоку 0 (поток для
  локального времени) */
  snprintf( buffer, MAX_BUFFER, "%s\n",
  ctime(&currentTime) );

  ret = sctp_sendmsg( connSock,
                      (void *)buffer,
   (size_t)strlen(buffer),
                     NULL, 0, 0, 0, LOCALTIME_STREAM,
  0, 0 );

  /* Посылается GMT по потоку 1 (поток для GMT) */
  snprintf( buffer, MAX_BUFFER, "%s\n",
            asctime( gmtime( &currentTime ) ) );

  ret = sctp_sendmsg( connSock,
                      (void *)buffer,
   (size_t)strlen(buffer),
                     NULL, 0, 0, 0, GMT_STREAM, 0, 0
```

```
    );

    /* Закрывается клиентское соединение */
    close( connSock );

  }

  return 0;
}
```

Исходный код в листинге 3 начинается с создания сокета сервера (`IPPROTO_SCTP` используется для создания сокета SCTP прямого соединения). Затем создается структура `sockaddr`, в которой указывается, что разрешаются соединения к любому локальному интерфейсу (используется подстановочный (wildcard) адрес `INADDR_ANY`). Структура `sockaddr` привязывается к сокету с помощью вызова `bind`, а потом сокет сервера переводится в состояние прослушивания (listening state). С этого момента возможны входящие соединения.

Обратите внимание на то, что протокол SCTP использует многие из тех же сокетов API, что и протоколы TCP и UDP.

Серверная программа ожидает в цикле нового подключения клиента. При возвращении из функции `accept` создается новое клиентское подключение, определяемое сокетом `connSock`. С помощью функции `time` выясняется текущее время и переводится в строку функцией `snprintf`. С помощью функции `sctp_sendmsg` (нестандартный вызов сокета) строка посылается клиенту по заданному потоку (`LOCALTIME_STREAM`). После того как строка с локальным временем послана, текущее время переводится в формат GMT и посылается по потоку `GMT_STREAM`.

Таким образом, сервер выполнил свои функции, поэтому сокет закрывается и переходит в режим ожидания нового клиентского подключения. Теперь рассмотрим, как клиент протокола точного времени обрабатывает многопотоковую передачу [7].

Клиент протокола точного времени

Реализация многопоточного клиента приводится в листинге 4.

Листинг 4. Реализация клиента точного времени для протокола SCTP, использующего многопотоковую передачу

```
int main()
{
  int connSock, in, i, flags;
  struct sockaddr_in servaddr;
```

```
struct sctp_sndrcvinfo sndrcvinfo;
struct sctp_event_subscribe events;
char buffer[MAX_BUFFER+1];

/* Создание сокета SCTP в стиле TCP */
connSock = socket( AF_INET, SOCK_STREAM, IPPROTO_SCTP );

/* Определяется адрес точки соединения */
bzero( (void *)&servaddr, sizeof(servaddr) );
servaddr.sin_family = AF_INET;
servaddr.sin_port = htons(MY_PORT_NUM);
servaddr.sin_addr.s_addr = inet_addr( "127.0.0.1" );

/* Соединение с сервером */
connect( connSock, (struct sockaddr *)&servaddr,
  sizeof(servaddr) );

/* Ожидается получение данных SCTP Snd/Rcv с помощью
  функции sctp_recvmsg */
memset( (void *)&events, 0, sizeof(events) );
events.sctp_data_io_event = 1;
setsockopt( connSock, SOL_SCTP, SCTP_EVENTS,
             (const void *)&events, sizeof(events) );

/* Ожидается получение двух сообщений */
for (i = 0 ; i < 2 ; i++) {

  in = sctp_recvmsg( connSock, (void *)buffer,
  sizeof(buffer),
                     (struct sockaddr *)NULL, 0,
                     &sndrcvinfo, &flags );

  /* Завершающий символ строки - 0 */
  buffer[in] = 0;

  if        (sndrcvinfo.sinfo_stream ==
  LOCALTIME_STREAM) {
    printf("(Local) %s\n", buffer);
  } else if (sndrcvinfo.sinfo_stream == GMT_STREAM) {
    printf("(GMT  ) %s\n", buffer);
  }

}

/* Закрытие сокета и выход */
```

```
  close(connSock);

  return 0;
}
```

В клиентском приложении создается сокет протокола SCTP, а затем структура `sockaddr`, содержащая информацию о конечной точке подключения. После этого с помощью функции `connect` устанавливается соединение с сервером. Для того чтобы получить номер потока сообщений по протоколу SCTP, необходимо указать параметр сокета `sctp_data_io_event`.

При получении сообщения с помощью функции API `sctp_recvmsg` дополнительно принимается структура `sctp_sndrcvinfo`, содержащая номер потока. Этот номер позволяет различать сообщения потока 0 (местное время) и потока 1 (GMT).

5.3.Применение процессов для обеспечения параллельной работы сервера

Рассмотри нижеследующий пример (см. листинг 5) [12].

Листинг 5. Пример многопроцессного сервера

```
signal(SIGCHLD,reaper);

while(1) {
ssock = accept(msock,(struct sockaddr *)&fsin,&len);
if(ssock <0){ /* Сообщение об ошибке */ }
switch(fork()) {
case(0):
close(msock);
/* Обработка поступившго запроса ведомым процессом */
close(ssock);
exit(0);
default:
close(ssock);
/* Ведущий процесс */
break;
case -1:
/* Сообщение об ошибке */
}
}
void reaper(int sig)
{
int status;
```

```
while(wait3(&status,WNOHANG,(struct rusage *)0)>= 0);
}
```

Ведущий процесс сервера начинает свое выполнение в главной процедуре. При каждом проходе по циклу ведущий сервер вызывает функцию *accept* для перехода в состояние ожидания запроса на установление соединения от клиента. После получения запроса создается сокет для новосоединения и вызов функции accept возвращает дескриптор этого сокета. Далее ведущий процесс вызывает функцию *fork(),* чтобы разделиться на два процесса. Затем он закрывает сокет, который был создан для обслуживания нового соединения, и продолжает выполнение бесконечного цикла.

Замечание: первоначальный и новые процессы имеют доступ к открытым сокетам после вызова функции *fork(),* и они оба должны закрыть эти сокеты, после чего система освобождает связанный с ним ресурсы.

Завершившийся процесс остается в виде так называемого процесса-зомби до тех пор, пока родительским процессом не будет выполнен системный вызов *wait3()*. Для полного завершения дочернего процесса (т.е. уничтожения процесса-зомби) необходимо перехватывать сигнал завершения дочернего процесса. Операционной системе дается указание, что для ведущего процесса сервера при получении каждого сигнала о завершении работы дочернего процесса (сигнал *SIGCHLD*) должна быть выполнена функция reaper, в виде следующего вызова: *signal(SIGCHLD, reaper);*

Функция reaper вызывает системную функцию wait3, которая остается заблокированной до тех пор, пока не произойдет завершение работы одного или нескольких дочерних процессов. Эта функция возвращает значение структуры status, которую можно проанализировать для получения дополнительной информации о завершившемся процессе. Поскольку данная программа вызывает функцию *wait3()* при получении сигнала *SIGCHLD*, вызов этой функции всегда происходит только после завершения работы дочернего процесса. Для предотвращения возникновения в сервере тупиковой ситуации в случае ошибочного вызова в программе используется параметр *WNOHANG*, который указывает, что функция *wait3* не должна блокироваться в ожидании завершения какого-либо процесса.

5.4. Применение потоков для обеспечения параллельной работы сервера

Поток выполнения представляет собой один из принципов организации отдельных вычислений, а один процесс может содержать от одного и более потоков [12, 13]. Операционная система ограничивает максимально допустимое количество параллельных потоков, также как и максимальное количество параллельных процессов. Все потоки процесса разделяют единый набор глобальных переменных и единый набор дескрипторов файлов.

Многопотоковые процессы обладают двумя основными преимуществами по сравнению с однопотоковыми процессами: более высокая эффективность и разделяемая память. Повышение эффективности связано с уменьшением издержек на переключение контекста. *Переключение контекста* – это действия,

выполняемые операционной системой при передаче ресурсов процессора от одного потока выполнения к другому. При переключении с одного потока на другой операционная система должна сохранить в памяти состояние предыдущего потока (например, значения регистров) и восстановить состояние следующего потока. Потоки в одном и том же процессе разделяют значительную часть информации о состоянии процесса, поэтому операционной системе приходится выполнять меньший объем работы по сохранению и восстановлению состояния. Вследствие этого переключение с одного потока на другой в одном и том же процессе происходит быстрее по сравнению с переключением между двумя потоками в разных процессах.

Второе преимущество потоков (разделяемая память), является более важным, чем повышение эффективности. Потоки упрощают разработку параллельных серверов, в которых все копии сервера должны взаимодействовать друг с другом или обращаться к разделяемым элементам данных. В частности, поскольку ведомые потоки в сервере совместно используют глобальную память.

Одним из недостатков потоков является то, что они имеют общее состояние процесса, поэтому действия, выполненные одним потоком, могут повлиять на другие потоки в том же процессе. Например, если два потока попытаются одновременно обратиться к одной и той же переменной, они могут помешать друг другу. API-интерфейс потоков предоставляет функции, которые могут использоваться потоками для координации работы.

Замечание: Многие библиотечные функции, возвращающие указатели на статические элементы данных, не являются безопасными с точки зрения потоков. Например, если два потока вызовут функцию *gethostbyname()* одновременно, то оба могут обратиться к одной и той же области памяти, что вызовет конфликт. Поэтому если несколько потоков вызывают определенную библиотечную функцию, они должны координировать свою работу для обеспечения того, чтобы в любое время ее вызов выполнял только один поток.

Еще один недостаток связан с отсутствием надежности. Если одна из параллельно работающих копий однопотокового сервера вызовет серьезную ошибку (например, в ней будет выполнена ссылка на недопустимую область памяти), то операционная система завершит только тот процесс, который вызвал ошибку. С другой стороны, если серьезная ошибка будет вызвана одним из потоков многопотокового сервера, то операционная система завершит весь процесс (т.е. все потоки этого процесса).

Координация и синхронизация работы потоков

В системе Linux предусмотрено три механизма синхронизации: мьютексы, семафоры и условные переменные [12].

1) В потоках мьютексы используются для обеспечения взаимно исключающего доступа к разделяемому элементу данных. Мьютекс инициализируется динамически путем вызова функции *pthread_mutex_init*. Следует предусматривать применение отдельного мьютекса для каждого элемента данных, который должен быть защищен. Перед использованием

элемента данных поток вызывает функцию *pthread_mutex_lock* и функцию *pthread_mutex_unlock* по окончании его использования. Эти два вызова позволяют одновременно обращаться к элементу данных только одному потоку. Первый поток, вызвавший функцию *pthread_mutex_lock* с конкретным мьютексом, продолжает свою работу без задержки. Однако система блокирует каждый следующий поток, вызывающий функцию *pthread_mutex_lock* с тем же мьютексом. Когда вызывается функция *pthread_mutex_unlock* операционная система разблокирует один из ожидающих освобождения этого мьютекса потоков.

2) *Семафор* представляет собой механизм синхронизации, обобщающий мьютекс и применяемый в том случае, если доступно *N* копий ресурса. Семафор инициализируется динамически. Для инициализации семафора применяется функция *sem_init*; в одном из ее параметров должен быть указан начальный счетчик *N*. Сразу после инициализации семафора поток вызывает функцию *sem_wait* до начала использования одной копии ресурса, а возвращая копию для использования другими потоками, вызывает функцию *sem_post*. Функция *sem_wait* может быть беспрепятственно вызвана потоками, число которых составляет *N*; после ее вызова каждый из них продолжает свое выполнение. Однако если к семафору попытаются обратиться дополнительные потоки, они будут заблокированы и будут заблокированными до тех пор, пока один из выполняющихся потоков не вызовет функцию *sem_post*.

3) Условные переменные требуются только в ситуации, при которой одновременно выполняются следующие два условия: ряд потоков использует мьютекс для получения взаимно исключающего доступа к некоторому ресурсу и сразу после приобретения ресурса поток должен перейти к ожиданию возникновения определенного события.

Без применения условных переменных в программах, которые сталкиваются описанной выше ситуацией, приходится использовать своего рода активное задание, в котором поток повторно приобретает мьютекс, проверяет условие, а затем освобождает мьютекс. Перед блокировкой на условной временной поток приобретает мьютекс. При вызове потоком функции *pthread_cond_wait* для перехода в состояние ожидания изменения условной переменной в потоке должна быть указана и условная переменная, изменения которой он ожидает, и захваченный им мьютекс. Операционная система освобождает мьютекс, захваченный потоком, и одновременно с этим блокирует поток в ожидании изменения условной переменной.

После выполнения функции *pthread_cond_wait* поток блокируется на указанной условной переменной до тех пор, пока какой-то другой поток не передаст сигнал об изменении этой переменной. Для передачи сигнала об изменении условной переменной могут применяться два способа; различие между ними определяется тем, что многочисленные ожидающие потоки могут обрабатываться по-разному. Функция *pthread_cond_signal* позволяет продолжить работу только одному потоку, даже если сигнала об изменении состояния переменной ожидают несколько потоков, а функция *pthread_cond_broadcast* позволяет продолжить работу всем потокам,

заблокированным на этой переменной. Разрешая одному потоку продолжить работу, операционная система одновременно разблокирует этот поток и позволяет ему снова приобрести мьютекс, которым он обладал, прежде чем заблокироваться на условной переменной, об изменении состояния которой получен сигнал. Ожидание изменения условной переменной равносильно отказу на время от мьютекса, а затем автоматическому повторному приобретению этого мьютекса после поступления сигнала об изменении условной переменной. В результате блокировка потока на условной переменной не исключает для других потоков возможности пройти через критический участок, поскольку мьютекс могут приобрести и другие потоки.

Описание основных функций библиотеки Pthread

Функция *pthread_create()* создает поток и имеет прототип

*int pthread_create(pthread_t *thread, pthread_attr_t *attr, void *(*handler *), void *arg);*

После создания нового потока в нем начинается выполняться функция (которая называется потоковой функцией), переданная параметром handler, причем ей самой в качестве первого параметра передается переменная arg. Параметр *attr* позволяет задать атрибуты потока (NULL для значений по умолчанию). *thread* — адрес переменной, в которую *pthread_create()* записывает идентификатор созданного потока. Созданный с помощью *pthread_create()* поток будет работать параллельно с существующими. Возвращается 0 в случае успеха и не ноль - в противоположном случае. Потоковая функция *handle*r имеет прототип

void my_thread_function(void *);*

Выполнение потока завершается в двух случаях: если завершено выполнение потоковой функции, или при выполнении функции *pthread_exit()*:

*void pthread_exit(void *retval);*

Эта функция завершает выполнение вызвавшего ее потока. Аргумент *retval* -- это код, с которым завершается выполнение потока. При завершении работы потока вы должны помнить, что *pthread_exit()* не закрывает файлы, и все открытые потоком файлы будут оставаться открытыми даже после его завершения, так что не забывайте подчищать за собой. Если вы завершите выполнение функции *main()* с помощью *pthread_exit()*, выполнение порожденных ранее потоков продолжится.

Потоки, как и порожденные процессы, по завершению работы сами по себе не освобождают ресурсы, занятые собой. Поэтому им необходимо помочь. Варианта, собственно, два: либо наряду с освобождением ресурсов какой-либо поток ждет его завершения, либо нет. Для первого варианта используем функцию *pthread_join()*:

*int pthread_join(pthread_t th, void **thread_return);*

Функция приостанавливает выполнение вызвавшего ее процесса до тех пор, пока поток, определенный параметром *th*, не завершит свое выполнение и если параметр thread_return не будет равен *NULL*, то запишет туда возвращенное потоком значение (которое будет равным либо *PTHREAD_CANCELED*, если поток был отменен, либо тем значением, которое было передано через аргумент функции *pthread_exit()*). Для второго варианта есть функция *pthread_detach()*:

int pthread_detach(pthread_t th);

Функция, которая делает поток *th* "открепленным" (detached). Это значит, что после того, как он завершится, он сам освободит все занятые им ресурсы. Обратите внимание на то, что нельзя ожидать завершения detached-потока (то есть функция *pthread_join* выполненная для detached потока завершится с ошибкой).

Следующая функция инициализирует блокировку, заданную параметром *mutex*

*int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr);*

где *mutexattr* -- ее атрибуты. Значение *NULL* соответствует установкам по умолчанию.

Функция *pthread_mutex_destroy* удаляет блокировку *mutex*. Вот ее прототип

*int pthread_mutex_destroy(pthread_mutex_t *mutex);*

Функция *pthread_mutex_lock* устанавливает блокировку *mutex*. Если *mute*x не была заблокирована, то она выполняет блокировку и немедленно завершается. Если же нет, то функция приостанавливает работы вызвавшего ее потока до разблокировки *mutex*, а после этого выполняет аналогичные действия. Вот ее прототип

*int pthread_mutex_lock(pthread_mutex_t *mutex);*

Функция *pthread_mutex_unlock* снимает блокировку *mutex* и имеет следующий прототип

*int pthread_mutex_unlock(pthread_mutex_t *mutex);*

Подразумевается, что эта функция будет вызвана тем же потоком, который ее заблокировал (через *pthread_mutex_lock*).

Ниже приведен пример (см. листинг 6) кода сервера, реализованного с применением потоков [12].

Листинг 6. Пример сервера, реализованного с применением потоков

```
struct {
pthread_mutex_t st_mutex;
/* Разделяемая переменная */
} GLOBAL;

int main() {
pthread_t  th;
pthread_attr_t  ta;

/* Создаем ведущий сокет, привязываем его к общепринятому
    порту и переводим в пассивный режим */

pthread_attr_init(&ta);
pthread_attr_setdetachstate(&ta, PTHREAD_CREATE_DETACHED);
pthread_mutex_init(&GLOBAL.st_mutex,0);

while (1)  {
ssock = accept(msock,  (struct sockaddr *)&fsin, &len);
if (ssock <0) {/* ошибка */}
if (pthread_create(&th, &ta, (void * )handler(void *),
    (void *)ssock) <0) {/* ошибка */}
}

int handler(int ssock)
{
pthread_mutex_lock(&GLOBAL.st_mutex);
/* выполнение операций с разделяемыми переменными */
pthread_mutex_unlock(&GLOBAL.st mutex);
return 0;
}
```

5.5.Однопотоковые псевдопараллельные сервера

В приложениях клиент/сервер, характеризующихся тем, что затраты на обеспечение ввода/вывода превышают затраты на подготовку ответа на запрос, в сервере может использоваться асинхронный ввод/вывод для организации псевдопараллельной работы клиентов [12]. Необходимо предусмотреть, чтобы единственный поток выполнения в сервере держал открытыми соединения с несколькими клиентами и обеспечивал обслуживание сервером того соединения, через которое в определенный момент поступают данные. Сам факт поступления данных используется для активизации обработки данных сервером.

Допустим, что один поток сервера обслуживает соединения TCP одновременно с несколькими клиентами. Поток блокируется, ожидая поступления данных. Сразу после поступления данных через любое соединение поток активизируется, обрабатывает запрос и передает ответ. Затем он снова блокируется, ожидая поступления данных из другого соединения. При условии, что процессор работает достаточно быстро для того, чтобы выдержать нагрузку, возложенную на сервер, однопотоковая версия сможет обслуживать соединения с таким же успехом, как и версия с несколькими потоками. Однопотоковая реализация не требует переключения между контекстами потоков или процессов, поэтому она может выдержать более высокую нагрузку по сравнению с реализацией, в которой используются несколько потоков или процессов.

В основе разработки программы однопотокового, параллельного сервера (см. листинг 7) лежит использование асинхронного ввода/вывода, организованного с помощь системного вызова *select*. Сервер создает сокет для каждого соединения, которое он должен поддерживать, а затем вызывает функцию *select*, которая ожидает поступления данных через каждое из них. Функция *select* может ожидать поступления запросов на выполнение операций ввода/вывода через все возможные сокеты, в том числе и одновременно ожидать поступления новых запросов на установление соединения.

Один единственный поток остается привязанным к общепринятому порту, через который он должен принимать запросы на установление соединения. Каждый из ведомых сокетов в наборе соответствует соединению, для которого ведомый поток должен обрабатывать запросы. Сервер передает набор дескрипторов сокетов функции *select* в качестве параметра и ожидает активизации любого из них. После возврата управления функция *select* передает в вызывающий оператор битовую маску, которая указывает, какой из дескрипторов в наборе готов к работе. В сервере для принятия решения о том, с каким из дескрипторов нужно продолжить работу, используется порядок их активизации.

Для определения того, какие операции (ведущего или ведомого потока) должны быть выполнены для данного дескриптора, в однопотоковом сервере используется сам дескриптор. Если к работе готов дескриптор, соответствующий ведущему сокету, сервер выполняет для него такие же операции, какие выполнил бы ведущий поток: он вызывает функцию *accept* с этим сокетом для получения нового соединения. Если же к работе готов дескриптор, соответствующий ведомому сокету, сервер выполняет операцию ведомого потока: вызывает функцию *read* для получения запроса, а затем отвечает на этот запрос.

Листинг 7. Пример однопотокового псевдопараллельного сервера

```
int msock;  /* Ведущий сокет сервера */
fd_set    rfds; /* Набор дескрипторов, готовых к чтению */
fd_set    afds; /* Набор активных дескрипторов */
int    fd, nfds,  ssock;
```

```
/* инициализация пассивного сокета msock*/
nfds = getdtablesize();
FD_ZERO(&afds);
FD_SET(msock,  &afds);
while (1) {
memcpy(&rfds, &afds, sizeof(rfds));
if (select(nfds, &rfds,  (fd_set *)0,  (fd_set *)0,
     (struct timeval *)0) <0) { /* ошибка */}
if (FD_ISSET(msock, &rfds)) {
alen = sizeof(fsin);
ssock = accept(msock,  (struct sockaddr *)&fsin, &alen);
     if (ssock <0) {/* ошибка */}
FD_SET(ssock, &afds);
}
for (fd=0; fd<nfds; fd++)
if (fd != msock && FD_ISSET(fd, &rfds))
if (handler(fd) == 0) { /* число полученных байт */
close(fd);
FD_CLR(fd,  &afds);
}
}
```

Выполнение потока этого сервера, как и выполнение ведущего потока сервера в параллельной реализации, начинается с открытия пассивного сокета в общепринятый порт. Используется системная функция *getdtablesize* для определения максимального числа дескрипторов, а затем применяются макрокоманды *FD_ZERO* и *FD_SET* для создания битового вектора, соответствующего дескрипторам сокетов, которые должны контролироваться функцией *select.* Затем сервер входит в бесконечный цикл, в котором он вызывает функцию *select* для ожидания готовности к работе одного или нескольких дескрипторов. Если готов дескриптор ведущего сокета, сервер вызывает функцию *accept* для получения нового соединения. Он добавляет дескриптор нового соединения к управляемому им набору и снова переходит в состояние ожидания активизации следующих дескрипторов. Если же готов дескриптор ведомого сокета, сервер вызывает процедуру *handler* для обработки клиентского запроса. Если один из дескрипторов ведомого сокета сообщает о получении признака конца файла (число полученных байт равно 0), сервер закрывает дескриптор и использует макрокоманду *FD_CLR* для удаления его из набора дескрипторов, используемых функцией *select*.

Контрольные вопросы

1. Какие Вы знаете основные вызовы API сокетов, которые используются при написании клиент-серверов?
2. Чем отличаются домены *AF_UNIX* и *AF_NET*?
3. Чем отличаются структуры *sock_addr* и *sock_addr_in*?

4. Что возвращают функции *send()* и *recv()*?
5. Чем отличается схема построения сервера с установлением соединения от сервера без установления соединения?
6. Как сделать, чтобы операционная система сама присвоила порт для Вашей серверной программы?
7. Чем отличаются функции *htonl()* и *htons()*?
8. Как организовать потоковую передачу данных по протоколу SCTP?
9. В чем достоинства и недостатки параллельных (мультипроцессных и мультипоточных) и псевдопараллельных серверов?
10. Как уничтожить зомби-процессы при построении многопроцессного сервера?
11. Что такое мьютекс? В чем его отличие от семафора?
12. В чем отличия между функциями *pthread_join* и *pthread_detach*?

Список литературы

1. Д-р Сидни Фейт.TCP/IP: Архитектура, протоколы, реализация (включая IP версии 6 и IP Security) – 2-е изд. 2000 г. 424 с.
2. Олифер В. Г., Олифер Н. А. Компьютерные сети. Принципы, технологии, протоколы. - 4-е изд. - СПб.: ПИТЕР, 2011. - 943с. - 451 с.
3. Олифер В. Г., Олифер Н. А. Основы компьютерных сетей. - СПб.: ПИТЕР, 2009. - 350с.
4. TCP/IP. Сертификационный экзамен – экстерном (экзамен 70-059) – СПб: Изд. «Питер», 1999. – 416 с.
5. Стивенс У.Р., Феннер Б., Рудофф Э. М. UNIX: разработка сетевых приложений. - 3-е изд. - СПб. : ПИТЕР, 2007. - 1038с. - (Мастер-класс). - Библиогр.: с.991-997 . - Алф. указ.: с. 998--1038. - 758 с.
6. RFC 3286 — Введение в SCTP // RFC 2.0 — Русские Переводы RFC. URL: http://rfc2.ru/3286.rfc (дата обращения: 28.01.2013).
7. М. Тим Джонс. Надежная передача данных по протоколу SCTP. Протокол передачи с управлением потоком объединяет преимущества протоколов TCP и UDP // IBM. developerWorks Россия, 2008. URL: http://www.ibm.com/developerworks/ru/library/l-sctp/ (дата обращения: 28.01.2013).
8. Описание протокола POP3 //CODENET. URL: http://www.codenet.ru/webmast/pop3.php (дата обращения: 28.01.2013).
9. Описание протокола SMTP // URL: http://www.codenet.ru/webmast/smtp.php (дата обращения: 28.01.2013).
10. Семенов Ю.А. Протокол пересылки файлов FTP // Telecommunication technologies - телекоммуникационные технологии (v4.0, 21 января 2013 года). URL: http://book.itep.ru/4/45/ftp_454.htm (дата обращения: 27.01.2013).
11. Теренс Чан. Системное программирование на C++ для Unix. Издательская группа BHV, 1999. – 592 с.
12. Камер Дуглас Э., Стивенс Дэвид Л. Сети TCP/IP, т.3. Разработка приложений типа клиент/сервер для Linux/Posix. – М.: Издательский дом «Вильямс», 2002. – 595 с.
13. Уолтон Ш. Создание сетевых приложений в среде Linux: перевод с английского – М.: Издательский дом «Вильямс», 2001.

Кирилл Валерьевич Павский

Протоколы TCP/IP и разработка сетевых приложений

Учебное пособие

Редактор: Ю.С. Майданов
Корректор: В.В. Сиделина

Подписано в печать 24.04.2013г.
Формат бумаги 60 х 84/16, отпечатано на ризографе, шрифт № 10,
изд. л. 8,1 заказ № 36, тираж – 100 экз., СибГУТИ.
630102, г. Новосибирск, ул. Кирова, д. 86