Федеральное агентство по образованию
Государственное образовательное учреждение
высшего профессионального образования
«Тверской государственный университет»

С. В. Сорокин

Разработка сетевых приложений с использованием стека протоколов TCP/IP

Учебное пособие

ТВЕРЬ 2009

УДК 681.3(075)
ББК 32.973.73
К

Рецензенты:

Сорокин С.В.
Разработка сетевых приложений с использованием стека протоколов TCP/IP: Учеб. пособие.-Тверь: Твер. гос. ун-т, 2013 – 80 с.

Излагаются базовые принципы организации сетевого взаимодействия компьютерных систем. Подробно рассматривается архитектура стека протоколов TCP/IP и интерфейс программирования Berkley Sockets. Пособие предназначено для студентов, изучающих предмет «Компьютерные сети» в рамках обучения по направлениям 010300, 010400, 080500 и 230700.

Оглавление

Введение

Стек протоколов TCP/IP стал стандартом де-факто для компьютерных сетей. Прежде всего, его популярность связана с бурным ростом сети Интернет. С другой стороны, такое развитие Сети стало возможным благодаря возможностям протоколов, положенных в её основу.

Пособие предназначено для студентов, которые хотят получить базовые знания о принципах построения компьютерных сетей и научиться разрабатывать приложения, использующие стек протоколов TCP/IP. Для овладения практическим материалом, изложенным в пособии, студент должен иметь навыки программирования на языках C или C++.

Пособие состоит из трех частей.

- Первая часть – «Сетевые протоколы», является введением в изучение компьютерных сетей. В ней изложены основы организации сетей. Описана эталонная модель взаимодействия открытых систем. Рассмотрена организация стека сетевых протоколов TCP/IP версии 4.
- Во второй части – «Программирование сетевых приложений для стека TCP/IP», рассказано о разработке сетевых приложений с использованием библиотеки Berkeley Sockets. Описано использование базовых функций библиотеки в рамках типичных сценариев: открытие активного соединения, приём входящих соединений. Описаны блокирующий и неблокирующий режимы работы сокетов. Основная часть материала излагается с позиции программирования в операционных системах семейства Unix. В конце главы описываются основные особенности использования библиотеки winsock в Microsoft Windows.
- Третья часть представляет собой обзор «классических» сетевых приложений, используемых для настройки и тестирования компьютерных сетей. В ней описаны такие программы, как ifconfig, ping, traceroute, arp, route, netstat и nslookup и netcat. Для каждой программы даётся краткое описание, перечисляются основные параметры и приводятся примеры использования.

Пособие содержит ряд заданий, позволяющих отработать навыки программирования сетевых приложений.

1 Сетевые протоколы

1.1 Основы организации сетевого взаимодействия

Создание компьютерных сетей представляет собой сложную организационную задачу. Во многих случаях бывает необходимо организовать взаимодействие между компьютерами различных архитектур, операционным системами и прикладным обеспечением разных версий и производителей. Среда передачи данных тоже может быть крайне разнородной – оптические каналы, сети Ethernet и другие специализированные кабельные сети, модемы для связи через телефонные сети и различные беспроводные технологии, действующие от масштаба «рабочего стола» (Bluetooth) до всей планеты (спутниковая связь). Успешная организация связи в таких разнородных условиях возможна только за счет использования общепринятых стандартов и структурного подхода к решению этой задачи.

Для работы в разнородной аппаратной среде обычно создают программное обеспечение с многоуровневой структурой. Каждый уровень решает свою конкретную задачу и взаимодействует только с соседними уровнями. Самым нижним уровнем в случае сетевого взаимодействия будет являться уровень, на котором будет осуществляться физическая связь между непосредственно соединенными узлами сети (компьютерами или специальными сетевыми устройствами). На самом верхнем уровне будут находиться пользовательские процессы. Промежуточные уровни должны «спустить» данные от пользовательского процесса до физического уровня, а затем (на другом компьютере) «поднять» их опять до уровня пользовательского процесса.

Вертикальное взаимодействие между уровнями обеспечивается стандартизацией способов передачи данных между уровнями, называемыми **интерфейсами.**

Спуская данные до самого нижнего уровня, и поднимая их обратно на другом компьютере, нижние слои обеспечивают своеобразный виртуальный канал данных между слоями, находящимися на одном уровне на разных компьютерах. Как правило, каждый уровень добавляет к передаваемым данным свою служебную информацию, которая анализируется аналогичным уровнем другой системы.

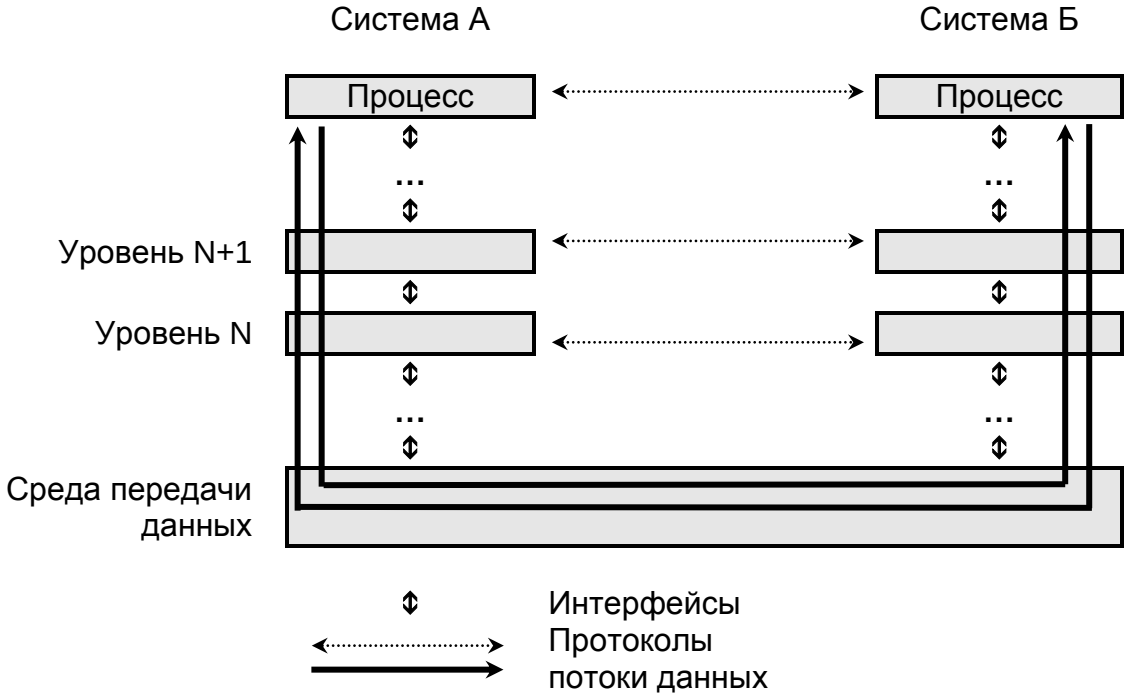
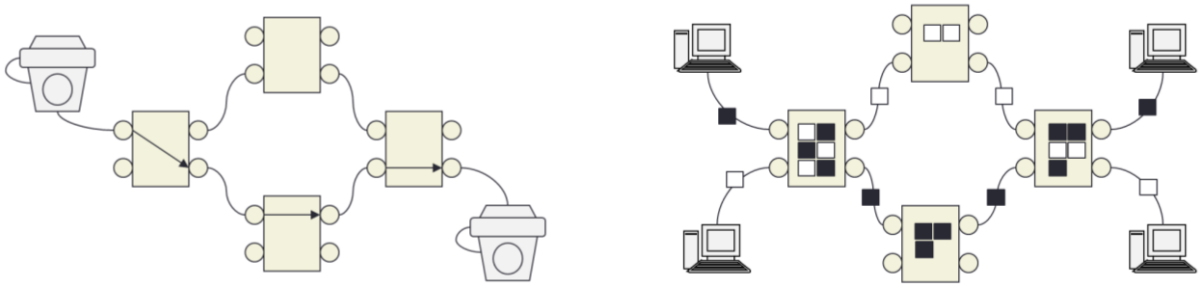


Рис. 1. Организация передачи данных с использованием стека сетевых протоколов

Формальный набор правил, определяющих последовательность и формат сообщений, которыми обмениваются сетевые компоненты различных вычислительных систем, находящихся на одном уровне, называют **сетевым протоколом**. Также этот термин используют для обозначения программного обеспечения, реализующего это набор правил.

Совокупность интерфейсов и протоколов, достаточную для организации взаимодействия удаленных процессов, называют **семейством протоколов** или **стеком протоколов**.

Среди существующих методов построения сетей, наиболее распространёнными являются два метода, известные как коммутация каналов и коммутация пакетов [1]. В случае коммутации каналов перед обменом данных между клиентами сети создаётся составной канал, используемый для передачи данных только этими клиентами (рис. 2а). Преимуществом такого способа является предоставление клиентам гарантированного качества связи: пропускная способность и задержки в канале известны заранее и не изменяются в процессе работы. Такой способ коммутации хорош для передачи непрерывного потока данных, например видео или звука. Не удивительно, что таким образом строились телефонные сети.



а б

Рис. 2. Способы коммутации

Однако выяснилось, что такой способ не очень хорошо подходит для компьютерных сетей передачи данных. Дело в том, что компьютерный трафик, как правило, пульсирующий: периоды интенсивной передачи данных сменяются периодами простоя, во время которых ресурсы, выделенные на поддержание канала, расходуются впустую. Для передачи таких данных был предложен другой метод – коммутация пакетов (рис. 2б). В этом случае передаваемый поток данных разбивается на небольшие фрагменты – пакеты, которые передаются по сети независимо друг от друга. Образующие инфраструктуру сети́ с коммутацией пакетов устройства должны обладать памятью, в которую они помещают принимаемые пакеты, а затем, по мере освобождения соединяющих их каналов, извлекают пакеты и отправляют их дальше. Таким образом, и каналы, и коммутаторы одновременно участвуют в передаче данных от многих клиентов.

Способ коммутации пакетов тоже неидеален – платой за одновременное использование ресурсов сети клиентами является непредсказуемость, непостоянность характеристик связи. Пропускная способность и задержки при передаче пакетов зависят от степени активности всех клиентов сети и могут меняться в процессе работы. Тем не менее, сейчас этот способ получил самое широкое использование не только в компьютерных сетях, но и в телефонии. Поэтому далее мы будем рассматривать передачу данных именно в режиме коммутации пакетов.

Для реализации обмена пакетами в рамках многоуровневой модели протоколов, программное обеспечение одинаковых уровней должно иметь возможность передавать информацию аналогичному уровню другого компьютера. Для этого в большинстве случаев, каждый уровень добавляет к переданной ему информации заголовок и передаёт получившийся пакет на более низкий уровень. Для нижних уровней заголовки, добавленные на более высоких уровнях, становятся данными. Таким образом, данные,

спускаясь вниз по стеку протоколов, постепенно обрастают всё новыми и новыми заголовками. Некоторые уровни накладывают ограничения на максимально допустимый размер пакета. В этом случае данные слишком большого размера могут быть разделены на несколько фрагментов. После получения на другом компьютере начинается обратный процесс – заголовки анализируются на соответствующем уровне, затем отбрасываются и оставшиеся данные передаются на уровень выше (рис. 3).

Рис. 3. Заголовки пакетов

1.2 Эталонная модель взаимодействия открытых систем

Стандартной схемой построения стека сетевых протоколов является **эталонная модель взаимодействия открытых систем** (Open System Interconnection – OSI), предложенная международной организацией стандартов (International Standard Organization – ISO). Эта схема показана на рисунке 4.

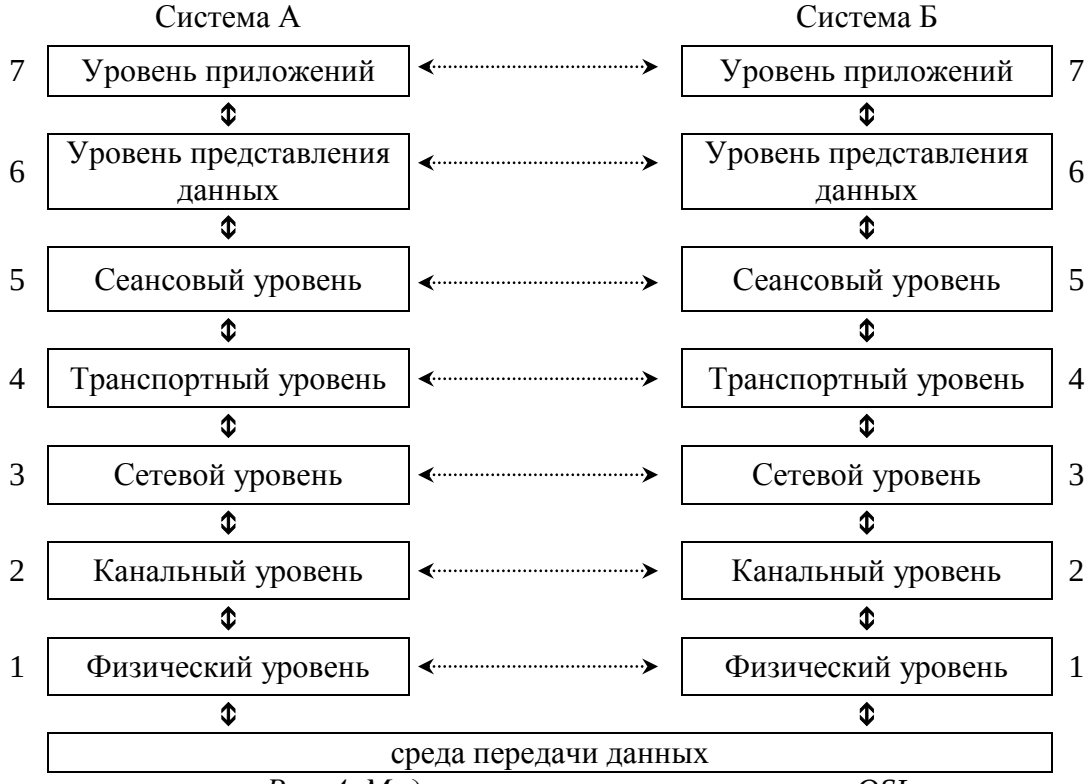


Рис. 4. Модель стека сетевых протоколов OSI

В модели OSI определены 7 уровней. Рассмотрим их, начиная с самого нижнего:

Уровень 1 – **физический**. Он описывает работу линии передачи данных и обеспечивает передачу отдельных битов между непосредственно соединёнными сетевыми устройствами. На нем определяются такие физические аспекты передачи информации по линиям связи, как: природа передающей среды, напряжения, крутизна фронтов импульсов, полоса пропускания, используемые частоты, помехозащищенность, способ передачи двоичной информации по физическому носителю и так далее, вплоть до размеров и формы используемых разъемов.

Часто линия передачи данных допускает подключение к ней одновременно нескольких устройств, которые могут с её помощью передавать данные друг другу. Такие линии называют **разделяемой средой передачи данных**.

Функции физического уровня реализуются во всех сетевых устройствах на аппаратном уровне. В компьютерах поддержку физического уровня обычно обеспечивает сетевой адаптер.

Уровень 2 – **канальный**. Этот уровень отвечает за передачу данных по физическому уровню между непосредственно связанными узлами сети. На нем формируются блоки данных для доставки по физическому уровню, называемые **кадрами** (**frame**). Протоколы канального уровня реализуются совместно сетевыми адаптерами и их драйверами.

На этом уровне определяются **канальные** или **MAC-адреса** (по названию подуровня **Media Access Control**). Эти адреса позволяют указывать получателя информации в пределах одного сегмента сети, образованного разделяемой средой передачи данных.

В задачи канального уровня входит также определение порядка доступа сетевых устройств к разделяемой среде передачи данных.

Уровень 3 – **сетевой.** Сетевой уровень несет ответственность за доставку информации от узла-отправителя к узлу-получателю в глобальной сети. На этом уровне решаются вопросы адресации, осуществляется выбор маршрутов следования пакетов данных, решаются вопросы стыковки сетей, построенных с использованием разных технологий канального уровня, а также осуществляется управление скоростью передачи информации для предотвращения перегрузок в сети. Определённые на этом уровне **сетевые адреса** позволяют уникальным образом адресовать устройство в рамках глобальных сетей. Передаваемые на сетевом уровне блоки данных называют **пакетами** (**packet**).

Уровень 4 – **транспортный**. Регламентирует передачу данных между удаленными процессами. Обеспечивает доставку информации вышележащим уровнем с необходимой степенью надежности, компенсируя, если это требуется, ненадежность нижележащих уровней. Этот уровень может исправлять такие ошибки, как искажение или потерю данных, доставка пакетов в неправильном порядке. Наряду с сетевым уровнем может управлять скоростью передачи данных и решает задачу адресации прикладных процессов в рамках хоста.

Уровень 5 – **сеансовый**. Координирует взаимодействие связывающихся процессов. Основная задача – предоставление средств синхронизации взаимодействующих процессов. Такие средства синхронизации позволяют создавать контрольные точки при передаче больших объемов информации. В случае сбоя в работе сети передачу данных можно возобновить с последней контрольной точки, а не начинать заново.

Уровень 6 – **уровень представления данных**. Отвечает за форму представления данных, перекодирует текстовую и графическую информацию из одного формата в другой, обеспечивает ее сжатие и распаковку, шифрование и декодирование.

Уровень 7 – **прикладной**. Это набор разнообразных протоколов, с помощью которых пользователи сети получают доступ к разделяемым ресурсам, таким как файлы, принтеры или гипертекстовые Web-страницы, а также организуют совместную работу, например, с помощью протокола электронной почты. На прикладном уровне программы обмениваются **сообщениями** (**message**) – логически законченными блоками данных.

Надо отметить, что модель OSI создавалась как обобщение опыта, полученного при создании стеков сетевых протоколов, таких как TCP/IP, IPX/SPX[1] и других. Поэтому в большинстве распространенных сетевых протоколов мы не увидим строгого соответствия с этой моделью. Модель OSI не следует рассматривать как жесткую схему, по которой построены все сетевые протоколы. Она скорее задает общее направление, в котором должны двигаться разработчики сетей, показывает основные проблемы, которые должны быть разрешены в процессе проектирования стека и устанавливает общую терминологию.

[1] Стек протоклов, который использовался в сетевой операционной ситеме Novel Netware.

На практике во многих случаях мы увидим, что часть задач, решающихся согласно модели OSI на разных уровнях, может быть решена на одном уровне, часть – наоборот перенесена в соседний, а что-то может вообще остаться на разрешение для пользовательских приложений или даже самому пользователю.

Модель OSI описывается во многих книгах, посвященных сетевым технологиям, например [1], [2], [3], [4].

1.2.1 Адреса в модели OSI

В модели OSI определяются три вида адресов, используемых в компьютерных сетях.

В случае, если среда передачи данных позволяет соединять более двух устройств, для идентификации получателя и отправителя информации на канальном уровне используются адреса, называемые **канальными**, **физическими** или **MAC-адресами**. Формат MAC-адресов зависит от используемой технологии канального уровня. Адресное пространство чаще всего является плоским, то есть адрес представляет собой число, без выделения какой-либо структуры.

Для нормальной работы канального уровня каждый контроллер сетевого интерфейса, подключенный к общей среде передачи данных, должен иметь уникальный MAC-адрес. Для обеспечения уникальности соответствующие организации по стандартизации обеспечивают распределение фрагментов адресного пространства между производителями сетевых контроллеров, а производители распределяют адреса из этих блоков между контроллерами. В итоге гарантируется, что все выпускаемые сетевые контроллеры имеют различные адреса.

Большинство современных сетевых контроллеров позволяют изменять MAC-адрес с помощью программного обеспечения. Неосторожное использование этой возможности может привести к нарушению работы сети.

Наряду с индивидуальными адресами, существуют **широковещательные** (**broadcast**) MAC-адреса. Кадры, адресованные на такой адрес, принимаются и обрабатываются всеми устройствами, подключенными к разделяемой среде.

Однако создать большую сеть с использованием только MAC-адресов невозможно. Во-первых, как уже отмечалось, формат MAC-адреса зависит от конкретной технологии канального уровня, и мы бы не смогли использовать разные технологии канального уровня в рамках одной сети. Во-вторых, процедура распределения MAC-адресов гарантирует их уникальность, однако при этом устройство с любым адресом может оказаться в произвольном месте сети. Поэтому, для того чтобы искать маршруты между устройствами по MAC-адресам, потребовалось бы хранить и обрабатывать полную карту сети, что не возможно для сети масштаба Интернет.

Для разрешения этих проблем на сетевом уровне вводится второй вид адресов – сетевые или логические адреса. Этот адрес предназначен для того, чтобы идентифицировать устройство в рамках глобальной сети, объединяющей множество, возможно разных, сетей канального уровня.

Практика построения сети Интернет показала, что для построения больших сетей сетевые адреса должны иметь иерархическую структуру. Это позволяет агрегировать описания сетей. Например, маршрутизаторам в США не обязательно знать структуру компьютерных сетей в России, им достаточно понимать, что все пакеты, адресованные в Европу, надо передавать, например, через трансатлантический кабель.

В идеальном случае распространение сетевых адресов тоже должно происходить в соотвествии с иерархической схемой – регистратор верхнего уровня выдает блоки адресов региональным регистраторам, те – крупным интернет-провайдерам, которые распределят их между подключенными к ним провайдерами поменьше, а те уже выдадют их клиентам. Такая схема обеспечила бы хорошие возможности по агрегации адресов. К сожалению, в таком случае сеть имела бы топологию деревьев, связанных произвольным образом лишь

на уровне крнупных провайдеров. Такая арихитектура была бы недостаточно надёжна из-за отсутствия дублирующих каналов. На практике многие провайдеры разных уровней и крупные организации-клиенты предпочитают иметь несколько независимых соединений с интернет, что увеличивает надёжность, но уменьшает возможности агрегации адресов.

Сетевые адреса позволяют идентифицировать компьютер или другое устройство в рамках глобальной сети. Однако на одном компьютере может работать несколько приложений, работающих с сетью. Например, на сервере могут одновременно работать почтовый сервер, сервер баз данных и служба удалённого доступа к файлам. Для того чтобы клиент мог указать, с какой из служб он желает соединиться, на транспортном уровне вводятся адреса прикладных процессов. Они позволяют указать конкретную точку сетевого взаимодействия в рамках одного сетевого устроства. Такая точка сетевого взаимодействия называется **сокет** (**soket**). Сокетом также называют и соответствующий программный объект, используемый для получения и отправки данных через сеть.

1.2.2 Устройства передачи данных

При передаче данных между компьютерами, которые не являются непосредственно соединенными на физичсском уровне или для передачи данных между сегментами сети, использующими разные физические носители, приходится поднимать данные на несколько уровней, чтобы обеспечить их трансляцию. Для этого используются специальные устройства, название которых зависит от того уровня, на котором они работают:

- **Хаб** (**hub**) или **ретранслятор** (**repeater**) работают на физическом уровне. Они обеспечивают ретрансляцию данных между несколькими сетями с одинаковыми или разными средами передачи (например, витая пара и коаксиальный кабель). При этом данные, поступившие на один порт, ретранслируются на все другие порты. Как правило, ретранслятором называют устройство, имеющее два порта, а хаб – больше.
- **Коммутатор** (**switch**) анализирует данные на канальном уровне. Обычно он используется для объединения нескольких сегментов сети. Коммутатор анализирует MAC-адреса получателей и отправителей пакетов и пересылает пакет только в тот сегмент, где находится получатель данных. Использование коммутаторов вместо хабов позволяет уменьшить объем передачи ненужной информации и увеличивает безопасность. В связи с удешевлением коммутаторов в последнее время они практически полностью вытеснили хабы.
- **Маршрутизатор** (**router**) работает на сетевом уровне. Он анализирует сетевые адреса отправителя и получателя и используется для организации передачи данных в сложных сетях, включая глобальные.
- **Шлюз** (**gateway**) используется для передачи данных между сетями, использующими существенно различающиеся архитектуры. Шлюзы могут работать на самых верхних, вплоть до 7-го, уровнях модели OSI. Как правило, они реализуются программным способом.

Поскольку хабы и коммутаторы работают, используя данные не выше, чем второго уровня модели OSI, то они не могут пересылать пакеты между сетями, в которых используются разные форматы кадров канального уровня и MAC-адресов. Соответственно, они могут объединять только сети, построенные на основе одной и той же технологии. Такие сети работают так, какбудто они представляют собой один сегмент, в котором все устройства соединены единой разделяемой средой передачи данных. Для адресации внутри такой сети достаточно использовать MAC-адреса; широковещательный пакет, отправленный одним из компьютеров, будет принят всеми подключенными к такой сети устройствами.

Использование коммутаторов вместо хабов позволяет исключить передачу данных в те участки сети, в которых они не нужны. Это снижает нагрузку на сеть и не позволяет потенциальным хакерам перехватывать информацию из соседних сегментов. Как правило, коммутаторы автоматически определяют, к каким портам подключены устройства с какими MAC-адресами, наблюдая за проходящим через них сетевым трафиком. Первое время, пока коммутатор ещё не накопил данные о конфигурации сети, он работает как хаб, пересылая пакеты во все интерфейсы. Однако он быстро получает необходимую информацию и начинает перенаправлять входящие пакеты только на тот интерфейс, к которому подключен получатель. Такой режим работы называется прозрачным и позволяет использовать коммутаторы без каких-либо настроек. Коммутаторы высокого класса имеют возможность ручной конфигурации и могут поддерживать различные дополнительные функции. Подробное описание вопросов, связанных с работой коммутаторов, может быть найдено в [1].

Для соединения разнородных сетей необходимо применять маршрутизаторы, которые используют адресацию сетевого уровня для пересылки данных через сегменты с различными канальными протоколами. Маршрутизаторы не пропускают через себя широковещательные пакеты. Для определения маршрутов передачи данных используется ручная настройка маршрутизаторов или специальные протоколы взаимодействия, которые позволяют маршрутизаторам самостоятельно определять кратчайшие маршруты до разных подсетей и автоматически корректировать их при изменении топологии сети. Маршрутизаторы также решают задачи по фильтрации трафика, распределению его между альтернативными каналами, мониторингу сети и т.д. Применение маршрутизаторов требует грамотного администрирования. Более детально работа маршрутизаторов, используемые ими алгоритмы и протоколы описаны в [3].

В настоящее время выпускаются и гибридные устройства, соединяющее в себе возможности управляемых коммутаторов и маршрутизаторов. Такие устройства иногда называют коммутаторами 3-го уровня.

Примеры передачи данных с использованием устройств разного уровня

На рисунке 5 показан фрагмент сети, в котором осуществляется передача данных от отправителя X.17 к получателю A.3. В этом примере сетевой адрес состоит из буквы (адрес сети) и цифры (адрес компьютера в сети), а MAC-адреса будут состоять из четырёх цифр.

В рассмотриваемом фрагменте сети находятся:

- маршуртизатор M1, соединяющий сети A и B с глобальной сетью;
- коммутатор S1;
- хабы H1 и H2;
- четыре компьютера A.1-A.4, входящих в сеть A.

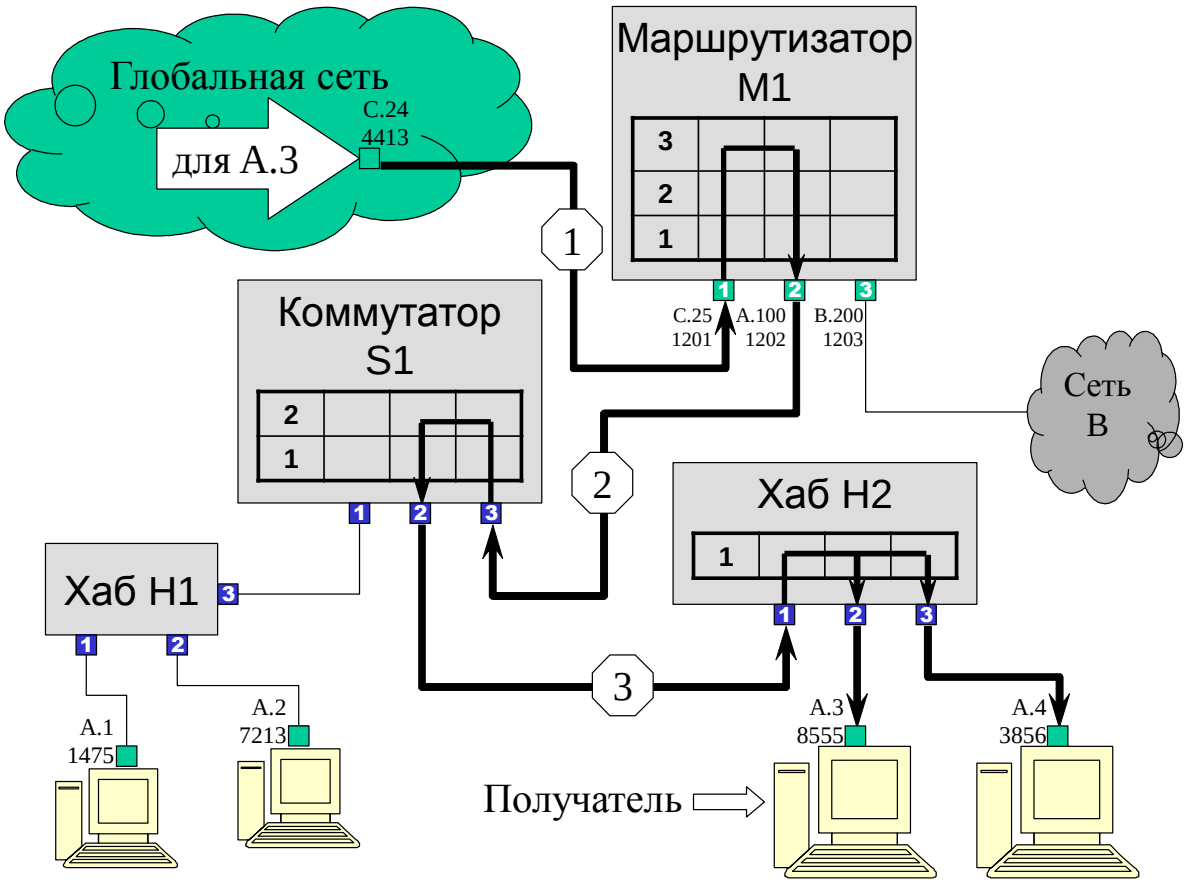


Рис. 5. Фрагмент сложной сети

Для работы маршрутизатора необходима **таблица маршрутизации (routing table)**. В ней прописаны адреса сетей и соответствующие этим сетям адреса маршрутизаторов (которые должны находиться в одной сети с отправителем). Таблица маршрутизации для M1 представлена в таблице 1.

Таблица 1. Таблица маршрутизации для M1

Сеть	Маршрутизатор	Интерфейс
A	---	2
B	---	3
Любая другая	C.24	1

Также маршрутизатору потребуется ARP-таблица для преобразования сетевых адресов в MAC-адреса (обычно она пополняется по мере необходимости – смотри далее протокол ARP). ARP-таблица для маршрутизатора M1 представлена в таблице 2.

Таблица 2. ARP-таблица маршрутизатора M1

Сетевой адрес	MAC-адрес
A.1	1475
A.3	8555
B.6	9965
C.24	4413
…	…

ARP-таблицы необходимы и всем компьютерам, подключенным к сети. При этом в такой таблице содержатся данные только про устройства тех сетей, к которым рассматриваемый компьютер (или маршрутизатор) подключен непосредственно.

Коммутатор в своей работе опирается на таблицу коммутации, описывающую, к какому из его портов подключено устройство с тем или иным MAC-адресом. Полная таблица коммутации для S1 представлена в таблице 3.

Таблица 3. Таблица коммутации коммутатора S1

MAC-адрес	Порт
1475	1
7213	1
8555	2
3856	2
1202	3

Таблица коммутации хранит MAC-адреса устройств только той сети, в которой находится коммутатор.

Обратите внимание, что порты коммутатора и хабов не имеют адресов. Эти устройства никогда не выступают в роли конечного получателя кадров, и, соответственно, адреса для них не нужны[2].

Пример 1: поступление пакета из внешней сети

Допустим, что один из клиентов глобальной сети (адрес X.17) посылает пакет, адресованный компьютеру с сетевым адресом A.3. Пакет поступает маршрутизатору M1 через порт C.25, соединенный с глобальной сетью (с одним из магистральных маршрутизаторов, имеющим адрес C.24). Наш маршрутизатор должен решить, на какой из его трех портов он должен переправить этот пакет. Для этого он анализирует адрес сети получателя (A) и по таблице маршрутизации определяет, что пакеты для этой сети следует отправлять на второй порт. Поскольку второй порт маршрутизатора непосредственно подключен к сети A (его второй порт имеет адрес из сети A, между M1 и получателем больше нет устройств 3-го уровня), маршрутизатор устанавливает, что сетевому адресу A.3 соответствует MAC-адрес 8555. После этого маршрутизатор упаковывает пакет в кадр с MAC-адресом получателя 8555 и отправляет его через интерфейс 2.

Отправленный маршрутизатором кадр попадает на коммутатор S1. Коммутатор, ориентируясь уже только на MAC-адрес, согласно таблице коммутации (табл. 3), перенаправляет данные на один из своих портов (в данном случае – на второй). Оттуда они попадают на хаб H2, который ретранслирует полученные данные на все свои порты.

В результате, кадр получает компьютер, которому адресован пакет, а также все компьютеры, подключенные к одному с ним хабу (компьютер A.4). Последние должны проигнорировать чужие кадры, что, как правило, и происходит, за исключением случаев, когда эти компьютеры используются хакерами для перехвата чужих данных или системными администраторами для контроля состояния сети.

В процессе передачи между сетями пакет сетевого уровня будет помещаться в различные кадры канального уровня. На рис. 6 показано, какие адреса будут указаны в заоголовках канального и сетевого уровня при передаче данных в точке 1.

Заголовок кадра			**Заголовок пакета**			**Данные**
Отправитель	**Получатель**	**…**	**Отправитель**	**Получатель**	**…**	
4413	1201	…	X.17	A.3	…	…

Рис. 6. Заголовки канального и сетевого уровней при передаче данных от X.17 к A.3 в сегменте 1

После обработки маршрутизатором, в точках 2, 3 и далее кадр будет выглядеть так, как показано на рис. 7.

Заголовок кадра			**Заголовок пакета**			**Данные**
Отправитель	**Получатель**	**…**	**Отправитель**	**Получатель**	**…**	
1202	8555	…	X.17	A.3	…	…

[2] Для контроля управляемых коммутаторов используется адрес (это может быть как MAC-адрес, так и сетевой адрес). Однако это адрес блока управления, который может рассматриваться как дополнительный компьютер, подключенный к коммутатору.

Рис. 7. Заголовки канального и сетевого уровней при передаче данных от X.17 к A.3 в сегментах 2 и 3

Пример 2: передача пакета внутри одного сегмента сети

Если компьютер A.3 пошлёт пакет компьютеру A.4, то этот пакет будет передан сразу в кадре с указанием MAC-адреса 3856 (поскольку отправитель и получатель находятся в одной сети). Этот кадр попадёт на хаб H2 и будет передан получателю (A.4) и коммутатору S.1. Коммутатор проанализирует этот кадр, но, установив, что получатель (3856) подключен к тому же порту, на который этот кадр был принят, пересылать в другие порты его не будет.

В этом примере данные передаются в кадре, показанном на рис. 8.

Заголовок кадра			**Заголовок пакета**			**Данные**
Отправитель	**Получатель**	**…**	**Отправитель**	**Получатель**	**…**	
8555	3856	…	A.3	A.4	…	…

Рис.8. Заголовки канального и сетевого уровней при передаче данных от A.3 к A.4

Пример 3: передача пакета через коммутатор

При отправке пакета с компьютера A.4 на компьютер A.2 он также будет сразу отправляться на MAC-адрес 7213. Начав путешествие через хаб H2, этот кадр попадёт компьютеру A.3 и коммутатору S1. Коммутатор определит, что адрес 7213 подключен к первому порту и ретранслирует кадр на этот порт. Там он будет принят хабом H1 и передан компьютерам A.1 и A.2 (что и требовалось).

Соответствующий этому примеру кадр представлен на рис. 9.

Заголовок кадра			**Заголовок пакета**			**Данные**
Отправитель	**Получатель**	**…**	**Отправитель**	**Получатель**	**…**	
3856	7213	…	A.4	A.2	…	…

Рис. 9. Заголовки канального и сетевого уровней при передаче данных от A.4 к A.2

Пример 4: отправка пакета в другую сеть

В случае отправки с компьютера A.1 пакета, направленного на адрес B.6, во-первых, установлено, что отправитель и получатель находятся в разных сетях. Соответственно компьютер A.1 будет определять, куда ему следует передать пакет с помощью своей собственной таблицы маршрутизации, приведённой в таблице 4.

Таблица 4. Таблица маршрутизации компьютера A.1

Сеть	**Маршрутизатор**	**Интерфейс**
A	---	1
Любая другая	A.100	1

Таким образом, пакет должен быть сначала отправлен маршрутизатору M1. A.1 определяет MAC-адрес соответствующий сетевому адресу A.100 (1202) и формирует кадр, в заголовке канального уровня которого указан получатель 1202, а в заголовке сетевого – B.6 (рис. 10).

Заголовок кадра			**Заголовок пакета**			**Данные**
Отправитель	**Получатель**	**…**	**Отправитель**	**Получатель**	**…**	
1475	1202	…	A.1	B.6	…	…

Рис. 10. Заголовки канального и сетевого уровней при передаче данных от A.1 к B.6 в пределах сети A

Этот кадр пройдет через хаб H1 и будет ретранслирован коммутатором S1 через третий порт. В результате пакет достигает своего получателя канального уровня – маршрутизатора M1. Он, в свою очередь, по таблицам маршрутизации и преобразования адресов определяет, что передача пакета должна осуществляться через порт 3 на MAC-адрес 9965. Соответственно, в сеть B будет отправлен кадр, представленный на рис. 11.

Заголовок кадра			**Заголовок пакета**			**Данные**
Отправитель	**Получатель**	**…**	**Отправитель**	**Получатель**	**…**	

1203	9965	…	A.1	B.6	…	…

Рис. 11. Заголовки канального и сетевого уровней при передаче данных от A.1 к B.6 в пределах сети B

Далее кадр будет передаваться через коммутаторы и хабы в сети B, пока не достигнет получателя.

Пакеты, адресованные в любые другие сети, также попадут на маршрутизатор M1. Он отправит их через первый интерфейс и упакует их в кадры, адресованные на MAC-адрес магистрального маршрутизатора (4413). Магистральный маршрутизатор будет осуществлять их дальнейшую передачу согласно своей таблице маршрутизации.

Наблюдения

В рамках локальной сети пакеты передаются упакованными в кадр канального уровня и обрабатываются, ориентируясь на MAC-адрес получателя (или шлюза, если пакет должен попасть за пределы сети). Эти кадры передаются хабами и коммутаторами без изменений.

Второй и третий пример показывают нам, что применение коммутаторов позволяет уменьшить нагрузку на сегменты сети, локализуя передачу каждого кадра только в тех сегментах, где его присутствие действительно необходимо.

Для правильной подготовки кадра отправляющей стороне необходимо преобразовать сетевой адрес получателя (шлюза) в MAC-адрес. Для этого все хосты поддерживают специальные таблицы соответствия.

При передаче данных между сетями пакеты сетевого уровня извлекаются маршрутизаторами из кадров, в которых они к ним прибыли, и переупаковываются в кадр, соответствующий следующей сети, через которую пакет должен пройти. При выборе сети для отправки пакета маршрутизатор ориентируется на сетевой адрес получателя и таблицу маршрутизации.

1.3 Стек протоколов TCP/IP

Стек протоколов TCP/IP начал разрабатываться в США в 60-х годах XX века. Эта работа выполнялась по заказу Министерства Обороны США по созданию военной вычислительной сети ARPANET. Первая реализация сетевых протоколов позволяла обмениваться сообщениями внутри небольшой сети и предполагала отсутствие потерь данных. С ростом сети появилась необходимость усложнения протоколов, и в результате их дальнейшего развития были разработаны протоколы TCP/IP.

Сеть ARPANET росла, появлялись новые части и закрывались старые, появлялись и гражданские сегменты сети, организованные на тех же протоколах. В результате развития и объединения этих сетей и появилась глобальная сеть Internet.

Протоколы стека TCP/IP определены в документах **Request For Comments** (RFC), издаваемых организацией Internet Engineering Task Force (IETF). Ознакомиться с оригинальной версией можно на сайте http://www.ietf.org/. Доступны также и русские переводы этих документов.

1.3.1 Сетевой порядок байтов

Компьютеры разных архитектур используют разные соглашения относительно хранения многобайтных чисел в памяти компьютера. Например, процессоры Intel записывают младший байт по наименьшему адресу. Такой способ на английском языке называется **little-endian**. Другие системы, напротив, сохраняют младший байт по большему адресу (например, PowerPC, Sparc). Этот порядок называют **big-endian**.

Чтобы успешно передавать числа, содержащие более одного байта, между различными системами, необходимо установить соглашение о порядке передачи байтов[3]. В стеке протоколов TCP/IP используется соглашение, согласно которому сначала передается старший байт, затем – младшие. Такой порядок называют **сетевым порядком байтов** (**network byte order**). Все многобайтные числа, используемые протоколами семейства TCP/IP, передаются в этом порядке. При обработке пакета необходимо преобразовать их в формат, используемый на локальном компьютере (англ., **host byte order**). При передаче данных следует наоборот преобразовать данные из локального формата в сетевой.

Соглашение о порядке передачи данных установлено только для информации, которая находится в заголовках протоколов. Формат данных, передаваемых приложениями, определяется самими приложениями.

Задание для самостоятельного выполнения

Напишите программу, определяющую порядок байтов для компьютера, на котором она запущена.

1.3.2 Стек протоколов TCP/IP

Стек протоколов TCP/IP организован по многоуровневому принципу. Согласно RFC761, в нем выделяется 4 уровня протоколов[4] [5]:

1. **сетевой уровень** (**network level**);
2. **уровень интернет** (**gateway level**);
3. **уровень узлов** (**host level**);
4. **уровень пользовательских приложений/процессов** (**application level**).

На каждом уровне находится несколько протоколов. Связь между наиболее распространенными протоколами показана на рисунке 12.

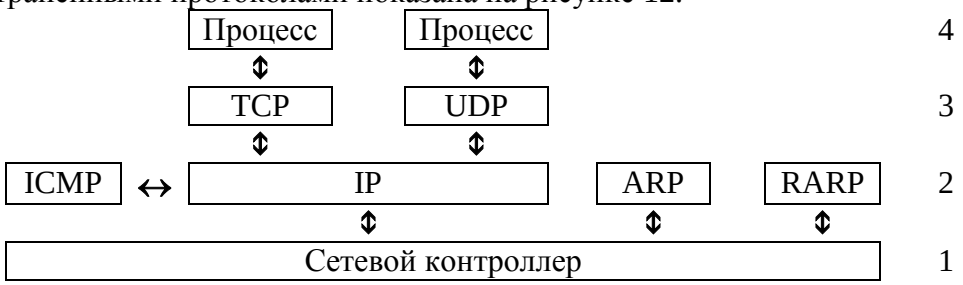


Рис. 12. Стек протоколов TCP/IP

1.3.3 Сетевой уровень

Этот уровень обеспечивает передачу данных между устройствами, непосредственно соединенными между собой. К этому уровню относятся протоколы Ethernet, SLIP, PPP и другие, а также физические средства передачи сигналов, такие как оптоволоконный кабель, витая пара. Эти протоколы не являются частью семейства TCP/IP и определяются

[3] Порядок передачи битов одного байта определяется протоколами нижних уровней модели OSI, он существенен только для сетевого контроллера. Однако для лучшего понимания формата заголовков, как они изображаются в RFC, полезно знать, что в стандартах TCP/IP используется соглашение, согласно которому нулевым битом является старший (как и получается, если писать число в двоичном виде слева направо).

[4] Обратите внимание, что разделение на уровни и их названия не соответствуют модели OSI ISO. На практике названия уровней стека TCP/IP, приведённые на рис. 6, обычно не используются. Вместо них используют названия уровней модели OSI ISO. Тем не менее, в этом разделе будут использоваться названия уровней стека TCP/IP.

отдельными стандартами. Однако стандарты TCP/IP определяют, каким образом должна осуществляться передача данных протоколов TCP/IP с использованием этих протоколов.

В дальнейшем мы будем использовать термин «локальная сеть» для обозначения группы сетевых устройств, соединенных одной физической средой передачи данных.

1.3.4 Уровень интернет

Основная задача протоколов этого уровня – обеспечить передачу данных от одного устройства к другому в глобальной сети. На этом уровне появляются адреса, идентифицирующие устройства, подключенные к сети, и обеспечивается передача данных между различными локальными сетями.

Реализованные на этом уровне протоколы позволяют сетевым устройствам обмениваться отдельными сообщениями, называемыми пакетами. При передаче контроллируется только отсутствие искажений данных в пакете. Уровень не гарантирует, что отправленные пакеты будут доставлены получателям. Порядок получения пакетов может не соответствовать порядку отправления, а в некоторых случаях возможно дублирование пакетов при передаче.

IP-адреса и маршрутизация

Адреса, используемые для идентификации компьютеров в сетях TCP/IP, называют IP-адресами. Эти адреса являются сетевыми адресами в терминах модели OSI ISO. IP-адрес представляет собой число длиной 32 бита. В текстовом виде его обычно записывают побайтно в десятичной системе, разделяя числа точками. Реже используют запись в виде одного 32-битного числа в шестнадцатеричной системе счисления.

IP-адрес назначается для каждого сетевого контроллера, который должен работать с сетью TCP/IP. Если в компьютере установлено несколько сетевых адаптеров, то каждому из них назначается свой адрес.

Вместе с адресом каждому сетевому интерфейсу назначается еще одно четырехбайтовое число, называемое маской. Маска имеет специальную структуру – она состоит из двух частей: младшие биты равны 0, а старшие 1.

Маска определяет, какие IP адреса принадлежат компьютерам одной локальной сети. Биты адреса, соответствующие нулевым битам маски, считаются номером устройства, а соответствующие 1 – номером сети. На рисунке 13 представлен пример структуры IP-адреса.

IP Адрес	192 1 1 0 0 0 0 0 0	0 1 0 1 0 1 0 0 0	2 0 0 0 0 0 0 1 0	75 0 1 0 0 1 0 1 1
Маска	255 1 1 1 1 1 1 1 1	255 1 1 1 1 1 1 1 1	255 1 1 1 1 1 1 1 1	192 1 1 0 0 0 0 0 0
Номер сети	192 1 1 0 0 0 0 0 0	168 1 0 1 0 1 0 0 0	2 0 0 0 0 0 0 1 0	64 0 1 0 0 0 0 0 0
Номер устройства	0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0	11 0 0 0 0 1 0 1 1

Рис. 13. Структура IP-адреса

В этом примере сетевой интерфейс имеет адрес 192.0.2.75 и маску 255.255.255.192. Как видно на рисунке, в таком случае адрес сети равен 192.0.2.64, а номер устройства в этой сети – 11 (0.0.0.11). Часть адреса, соответствующая номеру устройства, выделена цветом.

При отправке пакета драйвер протокола IP, прежде всего, определяет номер сети получателя сообщения. Если этот номер совпадает с номером сети одного из сетевых

адаптеров компьютера, то считается, что отправитель и получатель должны находиться в одной локальной сети и пакет передается напрямую с использованием протоколов сетевого уровня.

В случае, если номер сети получателя не совпадает с номерами локальных сетей, к которым отправитель имеет прямой доступ, стек TCP/IP определяет, куда направить пакет, используя таблицу маршрутизации. Найдя в таблице адрес подходящего маршрутизатора, стек отправляет ему пакет. Маршрутизатор анализирует IP-адреса назначения в полученных им пакетах и пересылает их получателям или другим маршрутизаторам, используя те же правила.

В простейшем случае таблица маршрутизации может включать всего одну строку, которая содержит адрес единственного маршрутизатора. Ему пересылаются пакеты для всех компьютеров, находящихся за пределами локальной сети. Такой маршурт называют **маршрутом по умолчанию** (**default route**).

Уменьшая число битов выделенных на номер устройства можно разбивать большую сеть на фрагменты меньшего размера. Крупные провайдеры Интернет получают большие блоки адресов, маршруты к которым прописываются в маршрутизаторах других крупных провайдеров. Затем провайдер разделяет этот блок на части, уменьшая число нулевых битов в маске, и распределяет их между своими клиентами. Провайдер прописывает информацию о маршрутизации до этих блоков в своих маршрутизаторах. «Снаружи» этот участок сети выглядит как одна большая сеть, а внутри она распадается на более мелкие фрагменты.

Часть адресов имеют специальное назначение. В качестве номеров устройств никогда не используются номера, у которых все биты равны нулю или единице. Адрес, у которого в номере устройства записаны только нули, считается адресом самой сети. Адрес, у которого все биты номера устройства равны 1, является **широковещательным** (**broadcast**). Пакеты, направленные на этот адрес, должны приниматься и обрабатываться всеми компьютерами, находящимися в данной сети[5]. Таким образом, самая маленькая сеть имеет маску 255.255.255.252 и включает четыре адреса: адрес сети, два адреса устройств и широковещательный адрес.

Некоторые блоки IP адресов зарезервированы для специальных целей. Согласно RFC1918 [6] следующие сети зарезервированы для использования вне Интернет:

- 10/8 (10.0.0.0 – 10.255.255.255);
- 172.16/12 (172.16.0.0 – 172.31.255.255);
- 192.168/12 (192.168.0.0 – 192.168.255.255).

Используя адреса из этих блоков, можно стороить частные сети, не имеющие непосредственной маршрутизации с Интернет. Отсутствие адресов из этих блоков в Интернет гарантирует, что если мы подключим компьютер одновременно к частной сети и Интернет, то компьютер сможет правильно определить, какие соединения надо устанавливать через Интернет, а какие – через частную сеть. Адреса из этих блоков часто используются при построении сетей микрорайонов и внутренних сетей провайдеров.

Ещё одни блок адресов, который имеет специальную функцию, это блок 127/8 (127.0.0.0 – 127.255.255.255). Этот блок является своеобразным синонимом для термина «моя локальная сеть». Чаще всего, из этого блока используется всего один адрес – 127.0.0.1, который является адресом «моего компьютера». Операционные системы, поддерживающие стек протоколов TCP/IP, создают специальный виртуальный сетевой интерфейс **внутренней петли** (**loopback**), на который назначается этот адрес. Его можно использовать для тестирования сетевых приложений, работая на одном компьютере. Для этого адреса также зарезервиоровано доменное имя `localhost`.

Другие специальные блоки IP адресов описаны в RFC5735 [7].

[5] Из соображений безопасности многие системы не обрабатывают пакеты, отправленные на широковещательные IP-адреса.

Протокол IP (Internet Protocol)

Основным протоколом уровня интернет является протокол IP. Именно он используется для передачи данных между компьютерами в сети. Этот протокол определен в документе RFC791 [8].

Протокол IP обеспечивает передачу пакетов данных от одного компьютера к другому. Этот протокол обеспечивает маршрутизацию пакета от отправителя к получателю; фрагментацию пакета, в случае если какие-то сегменты сети ограничивают максимальный размер передаваемого пакета, и его сборку обратно. Доставка пакета не гарантирована, он может быть потерян в процессе передачи или наоборот дойти в нескольких экземплярах. Отправленные по очереди пакеты могут быть получены в другом порядке. Решение задач контроля целостности передаваемых данных возлагается на протоколы более высокого уровня.

При передаче данных с помощью протокола IP перед передаваемым пакетом добавляется заголовок. Формат заголовка показан на рис. 14.

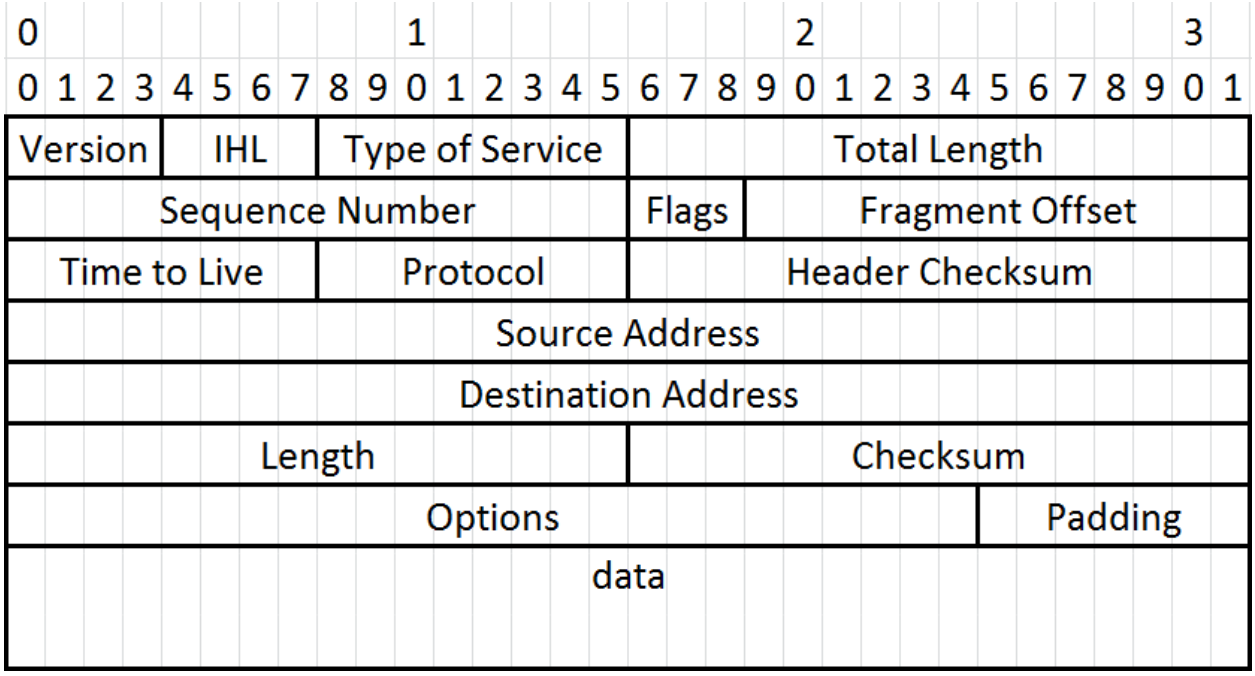


Рис. 14. Заголовок протокола IP

Version: 4 бита.

Поле версии содержит номер версии IP-протокола. Наиболее распространенной на данный момент является версия 4, определенная в RFC791. О новой версии IP v6 будет рассказано далее.

IHL: 4 бита.

Размер заголовка в 32-битных словах. Минимальное допустимое значение – 5.

Type of Service: 8 бит.

Параметр, который позволяет указать желаемый способ обработки пакета. Стандартом предусмотрен выбор между низкими задержками, высокой надежностью или высокой пропускной способностью. В Интернет этот параметр, как правило, игнорируется в связи с тем, что его использование может позволить «несознательным» пользователям перераспределить ресурсы глобальной сети в свою пользу за счет других пользователей.

Total Length: 16 бит.

Общая длинна пакета в байтах. Максимальный размер пакета, таким образом, не может быть более 65535 байт. Однако использование таких больших пакетов в сети не желательно. Стандарт требует, чтобы любой узел сети мог работать с

пакетами максимальной длины не менее 576 байт (из расчета 512 байт данных и 64 байта на заголовки IP и протоколов более высокого уровня).

Identification: 16 бит.

Идентификационный номер, назначаемый отправителем и используемый для сборки фрагментированных пакетов.

Flags: 3 бита.

Флаги управления:

Бит 0: зарезервирован, должен быть равен 0.

Бит 1: (DF) 0 = пакет может быть фрагментирован, 1 = пакет нельзя фрагментировать.

Бит 2: (MF) 0 = последний фрагмент, 1 = есть фрагменты далее.

Fragment Offset: 13 бит.

Поле показывает, к какому месту пакета относится этот фрагмент. Смещение измеряется в 8-байтных (64-битных) частях. Первый пакет имеет смещение 0.

Time to Live: 8 бит.

Это поле определяет максимальное время существования пакета в сети. Если в поле находится 0, то пакет уничтожается. Значение этого поля изменяется в процессе обработки пакета в сети. При передаче пакета каждый маршрутизатор должен уменьшать этот параметр на 1.

Введение этого параметра позволяет удалять из сети пакеты, при доставке которых из-за неправильных настроек маршрутизаторов возникают циклы.

Protocol: 8 бит.

Это поле определяет протокол более высокого уровня, который использует IP для передачи данных. Соответствие численных значений протоколам исходно определялось в RFC1700 [9], а в настоящее время фиксируется в онлайн базе данных, поддерживаемой IANA [10].

Header Checksum: 16 бит.

Контрольная сумма заголовка IP. Алгоритм вычисления подробно описан в RFC1071 [11].

Source Address: 32 бита.

IP-адрес отправителя.

Destination Address: 32 бита.

IP-адрес получателя.

Options: переменного размера.

Это поле позволяет задать дополнительные опции по обработке пакета. Опции могут отсутствовать либо может быть задана одна или несколько опций. Опции позволяют:

- задать маршрут доставки пакета;
- записать маршрут доставки пакета;
- записать время прохождения пакета на маршрутизаторах;
- задать параметры безопасности.

Padding: переменного размера.

Используется для выравнивания размера пакета по границе 32-битного слова.

Протокол ICMP

Протокол ICMP используется для передачи служебных сообщений протокола IP. Он определен в RFC792 [12]. Несмотря на то, что ICMP находится на одном уровне с IP, он использует пакеты протокола IP для передачи данных. При этом в поле Protocol заголовка IP-пакета указывается значение 1.

Многие из ICMP пакетов служат для уведомления отправителя о проблемах с доставкой его сообщения. В случае возникновения проблемы с доставкой самих сообщений ICMP, новое уведомление не отправляется, чтобы предотвратить

возникновение бесконечных потоков уведомлений об ошибках. Другие типы сообщений ICMP используются для проверки работоспособности сети.

В протоколе ICMP определены следующие виды сообщений:

Destination Unreachable Message

Адрес получателя недоступен (например, из-за сбоя на одном из каналов, задействованных в процессе маршрутизации).

Source Quench

Посылается отправителю, если у маршрутизатора не хватает ресурсов на обработку пакета, например, закончилась память, и пакет был удалён.

Time Exceeded Message

Превышено время доставки пакета. Значение Time To Live достигло 0.

Parameter Problem Message

Ошибка в одном из параметров IP-заголовка. Например, при анализе дополнительных опций.

Redirect Message

Перенаправление. Это сообщение может отправляться маршрутизатором (А) отправителю исходного сообщения, если маршрутизатор А обнаруживает, что он должен был отправлять пакет другому маршрутизатору (Б), находящемуся в той же сети, что и отправитель пакета. В этом случае маршрутизатор А сообщает отправителю, что следует отправлять пакеты напрямую маршрутизатору Б. В данный момент такие пакеты, как правило, игнорируются, в связи с тем, что эта опция может быть легко использована хакерами для различных атак.

Echo / Echo Reply Message

Это сообщение используется программой `ping` для проверки работоспособности IP-сети на нижнем уровне. Любое устройство, получившее пакет `echo`, должно выслать его отправителю пакет `echo reply`. Программа `ping` позволяет отправлять такие пакеты и выводит статистику по времени получения ответов.

Timestamp / Timestamp Reply Message

Позволяет запросить системное время с удаленной системы.

Information Request / Information Reply Message

Позволяет запросить адрес сети, к которой подключен отправитель сообщения.

Таким образом, протокол ICMP позволяет информировать отправителя сообщения о проблемах с доставкой его пакетов. Надо заметить, что доставка ICMP сообщений не гарантирована, поэтому для гарантированного контроля доставки сообщения получателю необходимо использовать дополнительные методы. Однако, если сообщение ICMP достигает отправителя исходного пакета, то оно позволяет узнать о возникновении проблем и их возможных причинах, не ожидая окончания таймаутов.

Практика показала, что некоторые из сообщений ICMP могут быть легко использованы для проведения различных атак на компьютеры, подключенные к сети. Поэтому сейчас часть ICMP сообщений часто не обрабатывается компьютерами и блокируется сетевыми фильтрами.

Протоколы ARP и RARP

Как уже отмечалось, для отправки пакетов на сетевом (канальном в терминах модели OSI) уровне используются MAC-адреса. При отправке IP пакета возникает задача определения MAC-адреса получателя или маршрутизатора по его IP-адресу.

Этот процесс обеспечивается протоколом Address Resolution Protocol (ARP) – протокол определения адреса. При первой отправке сообщения сетевое устройство с помощью широковещательного пакета запрашивает, какому MAC-адресу соответствует данный IP-адрес. Если в сети есть устройство с таким адресом, то оно отвечает отправителю и сообщает свой MAC-адрес. Определив соответствие IP и MAC-адреса,

устройство запоминает эту информацию в специальной ARP таблице. Старая информация из этой таблицы удаляется по таймеру. Однако при внезапной смене MAC адреса по какой-либо причине, возможны временные проблемы с работой сети, пока информация не обновится во всех ARP таблицах.

Протокол RARP позволяет производить обратное преобразование – из MAC-адреса в адрес IP.

Описание работы протоколов ARP и RARP может быть найдено, например, в [2].

1.3.5 Уровень узлов

Протоколы уровня интернет, такие как IP и ICMP, как правило, не используются для передачи пользовательских данных непосредственно. Программы пользователей работают с протоколами более высокого уровня, наиболее распространенными из которых являются UDP и TCP.

На уровне узлов в стеке протоколов TCP/IP решается задача адресации прикладных процессов. В роли такого адреса в протоколах TCP и UDP выступает **порт**. Порт представляет собой 16-битное слово, которое идентифицирует конкретного получателя данных в компьютере. Пара `{IP адрес, порт}` идентифицирует конечную точку сетевого соединения - **сокет**.

Один порт может участвовать в нескольких соединениях с другими сокетами. Уникальный канал передачи данных идентифицируется четверкой – `{IP адрес получателя, порт получателя, IP адрес отправителя, порт отправителя}`, или, что то же самое, парой сокетов.

В онлайн базе данных ”Well Known Port Numbers” [13] приведены номера портов, используемых распространенными программами. Однако конкретное назначение портов на конкретном устройстве определяется пользователем произвольным образом. Обычно операционные системы позволяют использовать порты с номерами меньше 1024 только привилегированным пользователям. Как правило, порты в этом диапазоне используются для приема входящих соединений различными серверами. Но и программы-клиенты для установления исходящих соединений также должны использовать какой-то порт. В основном для этого используется случайно выбранный порт с большим номером.

Протокол UDP (User Datagram Protocol)

Протокол UDP используется для отправки отдельных сообщений-пакетов между пользовательскими процессами. Этот протокол определен в RFC768 [14].

Также как IP, протокол UDP не дает гарантий от потери или дублирования пакетов при передаче. В нём отсутствуют какие-либо средства контроля получения пакетов адресатом. В случае необходимости, пользовательское приложение должно самостоятельно контролировать доставку, используя собственный протокол, или отказаться от применения UDP и выбрать другие протоколы.

Протокол UDP максимально прост. Перед отправляемыми данными добавляется UDP-заголовок (и далее заголовок IP) и данные отправляются получателю. Рассмотрим формат заголовка протокола UDP, представленного на рис. 15.

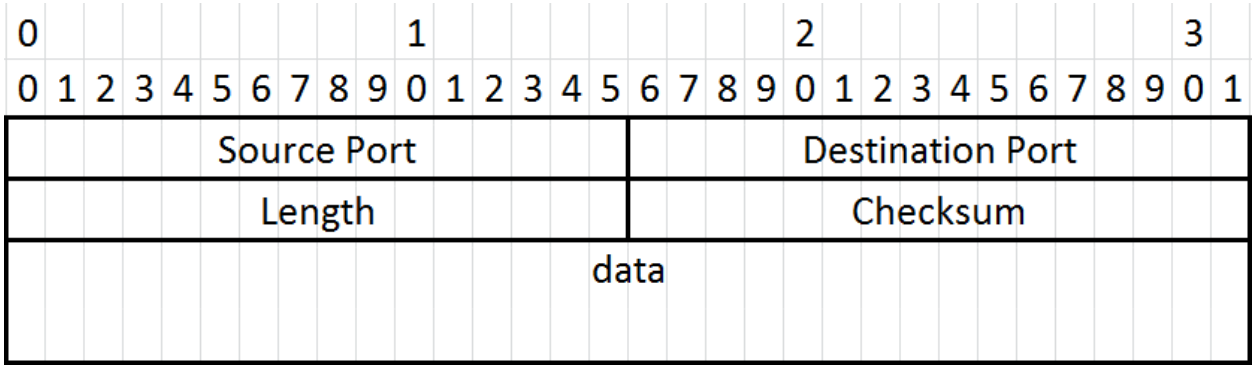


Рис. 15. Заголовок протокола UDP

Source Port: 16 бит.

Номер порта отправителя. Это значение может использоваться как порт, на который нужно отправлять ответ, если отсутствует другая информация. Во многих случаях этот параметр может не использоваться.

Destination Port: 16 бит.

Номер порта получателя.

Length: 16 бит.

Размер пакета, включая заголовок и пользовательские данные. Минимальное значение для этого поля – 8.

Checksum: 16 бит.

Контрольная сумма всех данных в пакете. Контрольная сумма также покрывает псевдозаголовок, который состоит из IP адреса отправителя и получателя, номера протокола и длинны UDP пакета.

Протокол UDP разработан для обеспечения минимальных накладок (как временных, так и по объему избыточных данных) при передаче данных. Его можно рекомендовать использовать в случаях, когда необходимо организовать простейший вариант взаимодействия двух программ, или когда скорость доставки свежих данных более важна, чем гарантированная доставка всех сообщений. Последний вариант часто используется в компьютерных играх или в системах IP-телефонии.

Протокол TCP (Transmission Control Protocol)

Рассмотренные ранее протоколы позволяют обмениваться пакетами ограниченного размера и не гарантируют их доставку. Однако в большинстве случаев пользовательским процессам требуется организовать между собой надежный канал данных, который бы позволял гарантированно передавать данные произвольного объема в двух направлениях. Такой канал позволяет организовать протокол TCP, определенный в [5].

Протокол TCP передает данные, используя пакеты IP. Дополнительные механизмы, введенные в протоколе TCP, позволяют разбить непрерывный поток пользовательских данных на отдельные пакеты, проконтролировать получение данных в правильном порядке и передать их пользовательскому процессу получателя. В случае потери или искажения данных при передаче, они посылаются заново.

Протокол TCP позволяет передавать информацию независимо в обоих направлениях. Причем имеется возможность независимо открывать или закрывать соединения в каждую из сторон.

Формат заголовка TCP пакета приведен на рис. 16:

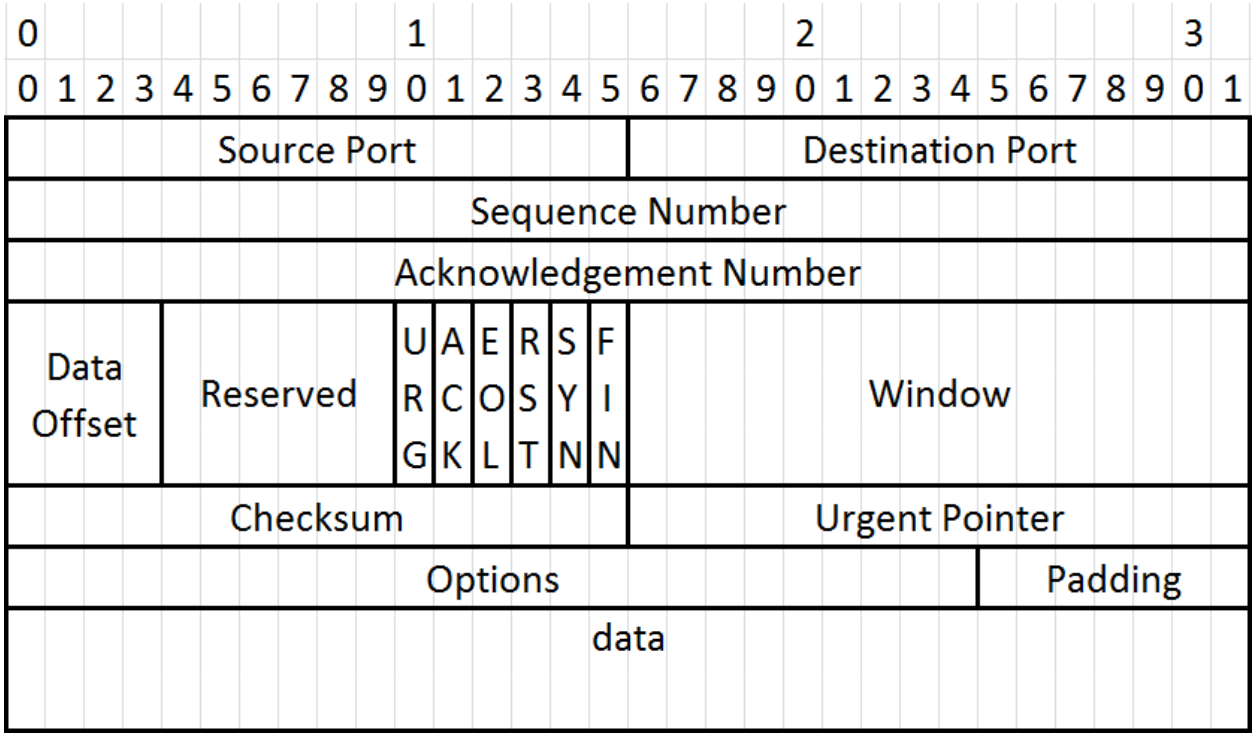


Рис. 16. Заголовок протокола TCP

Source Port: 16 бит.

Номер порта отправителя.

Destination Port: 16 бит.

Номер порта получателя.

Sequence Number: 32 бита.

Порядковый номер первого байта в пакете (за исключением случая, когда установлен флаг SYN).

Acknowledgment Number: 32 бита.

Если установлен флаг `ACK`, то это поле содержит порядковый номер следующего байта, который ожидает получить отправитель данного сообщения. После установки соединения это поле всегда устанавливается и используется для контроля передачи данных.

Data Offset: 4 бита.

Число 32-битных слов в TCP заголовке.

Reserved: 6 бит.

Не используется и должно равняться 0.

Control Bits: 8 бит:

Флаги. Биты имеют следующее значение:

`URG:` используется поле Urgent Pointer;

`ACK:` используется поле Acknowledgment;

`EOL:` конец письма;

`RST:` сброс соединения;

`SYN:` синхронизовать порядковые номера (Sequence number);

`FIN:` данных от отправителя больше не будет.

Window: 16 бит.

Число байт, которое готов принять получатель, начиная с указаного в поле acknowledgment.

Checksum: 16 бит.

Контрольная сумма данных и заголовка. Контрольная сумма также покрывает псевдозаголовок, в котором находятся IP адреса отправителя и получателя, номер протокола и длинна TCP пакета.

Urgent Pointer: 16 бит.

Смещение начала области срочных данных внутри блока данных. При получении эти данные немедленно передаются процессу-получателю в обход потока обычных данных, которые могут быть еще не прочитаны им. Значение этого поля анализируется, только если установлен флаг `URG`.

Options: переменный размер.

Дополнительные опции. Определена только одна дополнительная опция, которая позволяет при установке соединения передать размер буфера, используемого для обработки входящих данных.

Padding: переменный размер.

Используется для выравнивания размера TCP заголовка по границе 32-битных слов.

Установка соединения

Перед началом передачи данных должна быть выполнена процедура установки соединения, которая запускается обращением через интерфейс (функцию) `CONNECT`. Пользовательская программа может запросить активное соединение, в этом случае она указывает локальный и удаленный сокеты, между которыми устанавливается соединение.

Пассивное соединение означает, что программа будет ожидать входящего соединения от другого компьютера. При этом не обязательно указывать адрес и порт второй стороны, в этом случае будут приниматься соединения с любого входящего адреса. Протокол TCP позволяет открыть несколько пассивных соединений с одного порта, если будут указаны разные сокеты для удаленной стороны.

Одна из основных задач при установке соединения это синхронизация начального значения порядкового номера (**initial sequence number**, **ISN**) для передаваемых данных. Для этой цели нельзя использовать какое-то фиксированное значение (скажем 0), поскольку в этом случае могут возникнуть проблемы с разделением разных соединений. Если соединение между двумя сокетами будет внезапно прервано, а затем вновь открыто, то возможна ситуация, когда после установки нового соединения будет получен пакет данных, соответствующих старому соединению. Для того чтобы предотвратить ситуацию, когда этот пакет будет воспринят как корректные данные, протокол должен назначать различные начальные значения порядкового номера байтов. При этом отправляющая сторона генерирует новое значение ISN, а принимающая узнает его в процессе установки соединения.

Рассмотрим самый простой вариант установки соединения между двумя TCP системами А и Б (рис. 17). Система Б ожидает входящих соединений (пассивное соединение), как обычно делает сервер.

Система А открывает активное соединение с системой Б. Для этого система А генерирует и отправлет системе Б своё начальное значение порядокового номера (`ISN_A`). Флаг `SYN`, установленный в поле `CTL` сообщает, что в данном пакете сообщается новое значение порядкового номера.

Получив пакет с флагом `SYN`, система Б понимает, что с ней хотят установить соединение. Она генерирует своё начальное значение (`ISN_B`) и отправляет системе А пакет, в котором указывает:

- в поле `SEQ` – своё начальное значение `ISN_B`;
- в поле `ACK` – значение `ISN_A+1`. Это означает, что система Б готова принять от А байт данных с номером `ISN_A+1` и подтверждает получение начального значения от А.

Получив от системы Б её начальное значение ISN_B и подтверждение значения `ISN_A`, система А переходит в состояние «соединение установлено». Она посылает

системе Б подтверждение получения ISN_B. Получив подтверждение, система Б также переходит в состояние «соединение установлено».

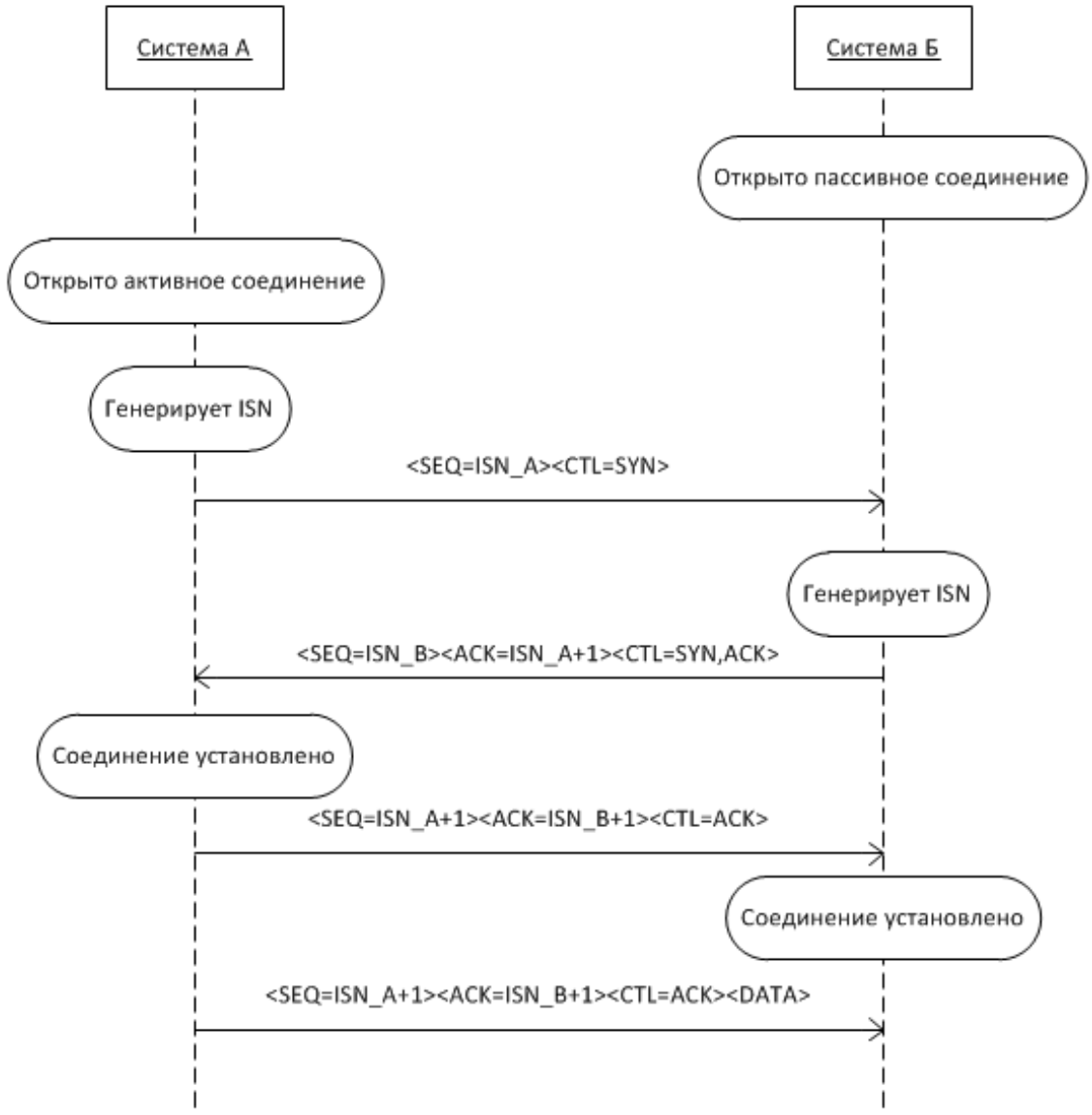


Рис. 17. Процедура установки TCP соединения

Обратите внимание, что при отправке пакетов с установленным флагом SYN значение счетчика SEQ увеличивается на 1. При отправке пакета, у которого установлен только флаг ACK и отсутствуют данные, значение SEQ не увеличивается, чтобы не привести к необходимости отправлять уведомление о получении уведомления и так далее, до бесконечности.

Протокол TCP допускает и более сложные последовательности действий при установке соединения. Допустимо одноверменное открытие активного соединения друг с другом. Возможна повторная отправка пакетов в случае их потери. В процесс могут вмешиваться пакеты от предыдушего соединения этих же систем, задержавшиеся в сети.

Передача и прием данных

При передаче данных пользовательский процесс передает данные протоколу TCP с помощью интерфейса SEND. Эта функция позволяет передать протоколу для обработки блок данных произвольной длинны, называемый **письмом** (**letter**). При передаче письмо может быть разбито протоколом TCP на более мелкие части. Флаг конца письма (EOL) говорит протоколу, что имеющиеся в буфере данные необходимо передать дальше (на

отправку или на получение пользовательской программой) не ожидая полного заполнения буфера.

Для приема входящих данных выделяется буфер, называемый **окно** (**Window**) и запоминается значение порядкового номера (sequence number) соответствующее первому байту буфера. Текущие размер окна и первый порядковый номер передаются в каждом пакете, отправляемом другой стороне. Если принимается пакет с данными, которые попадают внутрь окна, то эти данные сохраняются в буфере. Пакеты, содержащие данные, выходящие за пределы окна, игнорируются. При получении данных, расположенных в самом начале окна, эти данные передаются для обработки пользовательскому процессу, а номер ожидаемого байта сдвигается на первый отсутствующий байт.

Протокол TCP хранит отправленные данные до тех пор, пока по порядковому номеру первого ожидаемого байта не станет понятно, что они успешно приняты. Если подтверждение не приходит в течение определенного времени, передача данных повторяется.

Получение данных пользовательским процессом происходит через интерфейс `RECEIVE`.

Завершение соединения

Для завершения соединения предусмотрен интерфейс `CLOSE`. Закрытие соединения понимается как уведомление второй стороны о том, что никакие данные больше передаваться не будут. При этом система может продолжать прием данных до тех пор, пока от второй стороны также не поступит сообщения о завершении передачи.

При завершении соединения протокол гарантирует, что все отправленные пользовательским процессом данные были успешно получены протоколом TCP второй стороны[6].

Предусмотрен также интерфейс `ABORT`, который позволяет аварийно завершить соединение. При этом все данные, находящиеся в буферах приема и отправки, уничтожаются, и удаленной системе отправляется пакет с установленным флагом `RST`. Получение такого пакета приводит к аварийному завершению соединения со второй стороны.

Машина состояний TCP

Стандарт описывает работу реализации протокола TCP в виде конечного автомата. Автомат может находиться в одном из реальных состояний: `LISTEN`, `SYN-SENT`, `SYN-RECEIVED`, `ESTABLISHED`, `FIN-WAIT-1`, `FIN-WAIT-2`, `TIME-WAIT`, `CLOSE-WAIT`, `CLOSING`, или в фиктивном состоянии `CLOSED`. `CLOSED` является фиктивным, поскольку в этом состоянии отсутствует **Transmission Control Block** (**TCB**) – блок данных, используемых протоколом для хранения информации о соединении, включая состояние.

На рисунке 18 представлена машина состояний протокола TCP согласно RFC761.

Приведем краткое описание состояний:

`LISTEN` – ожидание входящего соединения с любого адреса и порта;

`SYN-SENT` – ожидание соответствующего пакета открытия соединения, после отправки запроса на соединение;

`SYN-RECEIVED` – ожидание подтверждения открытия соединения после получения и отправки запроса на соединение;

`ESTABLISHED` – соединение открыто, протокол готов к приему и передаче данных;

[6] Обратите внимание, что на момент завершения вызова CLOSE не гарантируется получение данных пользовательским процессом со второй стороны. Данные могут находится в буфере приёма TCP.

FIN-WAIT-1 – ожидание запроса на завершение соединения от удаленной стороны, или подтверждения о получении отправленного запроса на завершение соединения;

FIN-WAIT-2 – ожидание запроса на завершение соединения от удаленной стороны;

TIME-WAIT – ожидание в течение достаточно длительного периода, чтобы быть уверенным, что удаленная сторона успела получить подтверждение о завершении соединения;

CLOSE-WAIT – ожидание запроса о завершении соединения от местного пользователя;

CLOSING – ожидание подтверждения о закрытии соединения от удаленной стороны;

CLOSED – состояние соединения отсутствует.

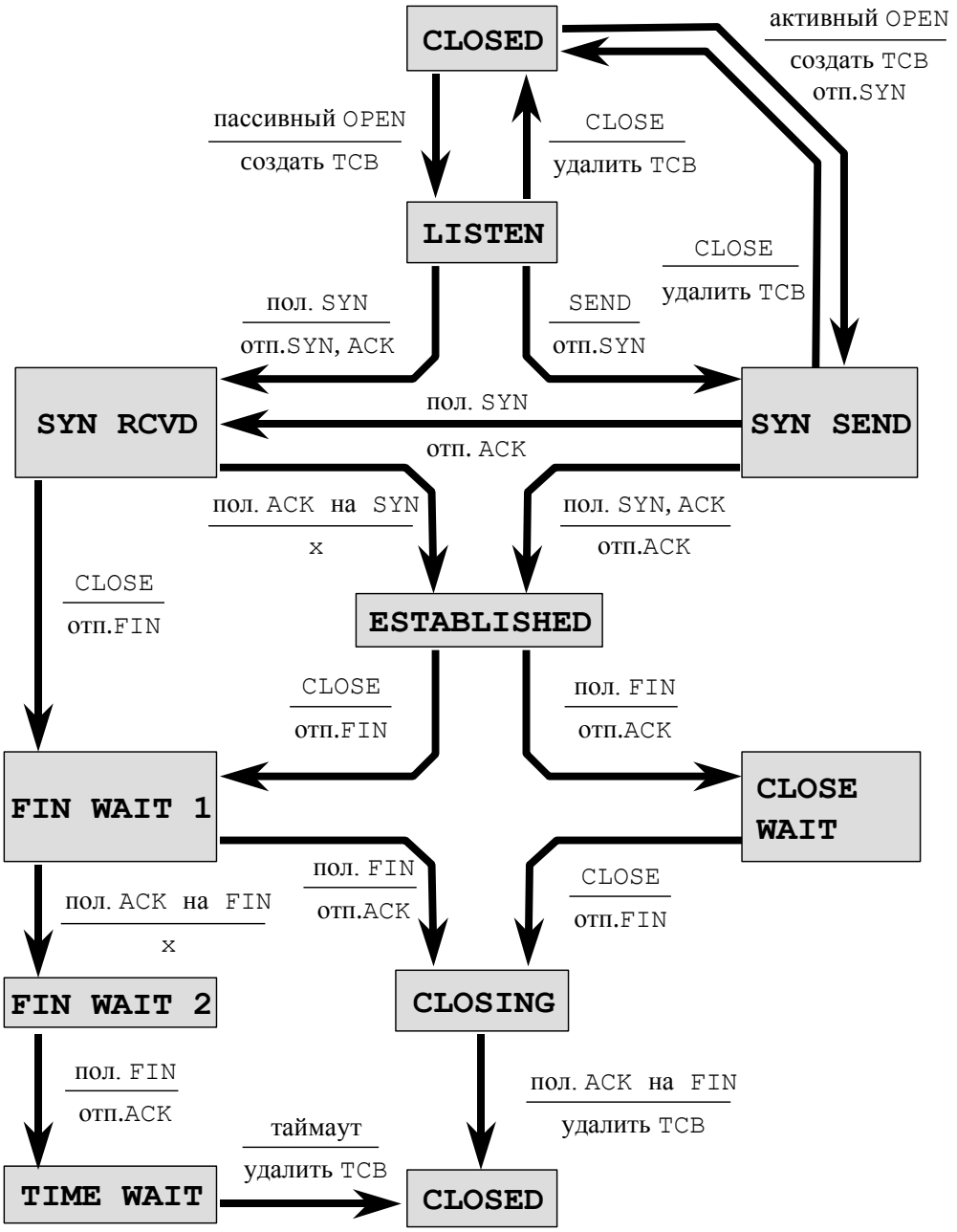


Рис. 18. Машина состояний TCP

1.3.6 Уровень приложений

Используя возможности уровня узлов, можно создавать протоколы, которые обеспечивают обмен информацией между пользовательскими приложениям. Такие протоколы определяют правила передачи по сети файлов, электронной почты или гипертекстовых документов. В настоящее время разработано большое число протоколов уровня приложений, предназначенных для решения различных задач. Многие из распространенных протоколов, которые были разработаны в первые годы развития Интернет, определены в стандартах RFC. Среди таких протоколов можно назвать:

- протокол передачи файлов FTP [15],
- протоколы для получения почты клиентом с почтового сервера POP3 [16] и более совершенный IMAP [17],
- протокол для отправки почты от клиента на почтовый сервер SMTP [18],
- протокол передачи конференций USENET NNTP [19].

Но не все протоколы уровня приложений определяются в документах IETF. Один из самых распространенных в данный момент протоколов – протокол передачи гипертекстовых документов, лежащий в основе World Wide Web – HTTP – разрабатывается организацией World Wide Web consortium (W3C).

1.3.7 DNS

Непосредственное использование IP-адресов возможно, разве что, в небольших сетях, в которых можно при настройке прописать адреса всех необходимых межкомпьютерных связей в конфигурации программного обеспечения. Использование цифровых адресов простыми пользователями в сети достаточно большого размера было бы крайне затруднительным и неудобным. Представьте, что вам пришлось бы запоминать адрес электронной почты друга как ivanov@127.156.87.92.

Чтобы человеку было проще запонить адрес, его лучше задать словом. Для обеспечения такой возможности необходимо реализовать механизм, позволяющий сопоставить символьные имена компьютеров и их IP-адреса. Одним из первых вариантов решения этой проблемы было использование механизма преобразования адресов с помощью текстового файла. Такой файл называется `hosts` и находится в директории `/etc` в UNIX системах (в Windows аналогичный файл можно найти в поддиректории `SYSTEM32\drivers\etc` внутри системной директории Windows). В каждой строке этого файла указывается IP-адрес и соответствующие ему символьные имена. В результате у пользователей появилась возможность вводить в программах адреса в символьном виде. Программа, обращаясь к специальной системной функции, преобразует введенную пользователем текстовую строку в цифровой IP-адрес.

Однако такое решение проблемы также подходит только для небольших сетей, в которых можно прописать в локальный файл на каждом компьютере адреса всех других компьютеров. Для распространения данных об именах компьютеров по сети первоначально использовали протокол FTP. Предполагалось, что пользователи будут скачивать файл, содержащий имена всех компьютеров в сети, с центрального сервера. Но с ростом сети это приводило к возрастанию объемов передаваемой информации и сложности поддержания единого файла. Для поддержания роста сети была разработана распределённая база данных **Domain Name System** (**DNS**).

DNS определяет иерархию символьных имен. Имя компьютера в системе DNS выглядит как текстовая строка, которая разделяется на части точками. Самая левая часть является именем компьютера, части, расположенные правее, определяют принадлежность компьютера к элементам иерархии более высокого уровня. Такой элемент иерархии называют **доменом** (**domain**). **Полный** (или **абсолютный**) адрес в системе DNS должен

кончаться точкой, расположенной справа после адреса. Эта точка соответствует корню дерева DNS имен, который имеет пустое имя. Однако на практике последнюю точку часто опускают. Если точка справа не указана, то считается, что имя является **относительным.**

При разрешении относительного адреса система будет пытаться добавить справа имена доменов из известного ей списка для поиска имен. В этот список входят, как минимум, имя домена системы и корневой домен ".". Как правило, операционная система позволяет пользователю дополнить этот список произвольными именами доменов.

База данных DNS имеет распределенную структуру. Хранение и передачу информации о компьютерах и доменах более низкого уровня, входящих в какой-то домен, обеспечивает DNS-сервер этого домена. Он отвечает на запросы пользователей и других серверов и передает им информацию из своей базы данных, используя специальный протокол уровня приложений. Хранение информации о доменах верхнего уровня обеспечивают несколько серверов. Информация об IP-адресах этих серверов указывается в конфигурации всех других серверов.

Для обеспечения работы сетевого устройства с адресами DNS необходимо указать в конфигурации IP-адрес одного (или нескольких) ближайших DNS-серверов. При разрешении введенного пользователем адреса система обращается к этому серверу. Если сервер располагает информацией о запрашиваемом имени, то он сразу же возвращает ответ клиенту. Если запрошенная информация отсутствует, то он рекурсивно обращается к другому серверу, который мог бы располагать такой информацией. Как правило, запрос сначала продвигается вверх по иерархии, а затем спускается вниз по ветке, соответствующей разрешаемому адресу.

Полученная информация запоминается сервером и используется для ускорения последующих запросов об этом же адресе. Допустимое время хранения «чужой» информации указывается первичным сервером. Система кэширования доменных имен кроме ускорения повторных запросов обладает не очень приятным побочным эффектом – если мы изменим информацию на первичном сервере, то на многих серверах останется закэшированная копия старой информации, которая будет сообщаться всем их клиентам. Клиенты смогут получить корректную обновленную информацию только после того, как закончится допустимое время хэширования информации.

Существует и нерекурсивный способ разрешения адресов. В этом случае сервер, не обладающий необходимой информацией, возвращает клиенту адрес того сервера, который мог бы ею обладать. Клиент, получив такой ответ, самостоятельно обращается к следующему серверу, который может перенаправить его к третьему, и так далее, пока имя не будет разрешено.

Наряду с IP-адресами система DNS позволяет передавать и другую информацию. Например, с каждым адресом DNS можно сопоставить один или несколько адресов почтовых серверов, которым необходимо посылать почту, предназначенную для этого домена. Поэтому мы можем получать почту на имя домена (например, tversu.ru), даже если этому имени непосредственно не соответствует ни один IP-адрес.

Часто может возникнуть и обратная задача – определить адрес DNS, соответствующий какому-то IP-адресу. Для этого существует специальная иерархия IN-ADDR.ARPA. Для получения информации об IP-адресе, например 1.2.3.4, необходимо сделать запрос об имени 4.3.2.1.IN-ADDR.ARPA. Обратите внимание, что цифры адреса записываются в обратном порядке – это позволяет передавать соответствующие поддомены этой иерархии совместно с разделением IP-сетей на подсети.

Описание концепции и реализации системы DNS можно найти в [4], [2] и в документах RFC, в первую очередь, в RFC1034 [20] и RFC1035 [21].

1.3.8 Развитие стека протоколов TCP/IP

Недостатки IP версии 4

Стек протоколов IPv4 оказался очень успешной сетевой архитектурой. Начав развитие как нишевая технология, в основном используемая в университетских и научных кругах, он, на данный момент, практически полностью вытеснил все другие сетевые технологии. При этом была продемонстрирована беспрецедентная масштабируемость – от первой сети из нескольких компьютеров, до глобальной сети, включающей сотни миллионов узлов, размеры которой вряд ли предвидели его первые разработчики.

Несмотря на всю свою успешность, архитектуру IPv4 нельзя называть идеальной, и о необходимости дальнейшего развития или разработки новой технологии стали говорить ещё в конце 80-ых годов.

Можно выделить три основные проблемы, присутствующие в IPv4, которые не могут быть разрешены без существенного изменения архитектуры протокола [22].

1. Размер адресного пространства.

   Уже в начале 90-х годов стало очевидно, что адресное пространство IPv4 недостаточно для поддержания существующих темпов развития Интернет. Для решения этой проблемы несколько раз изменялись принципы распределения адресов и было введено множество обходных решений. Надо отметить, что эти меры до сих пор позволяют избежать окончательного исчерпания адресного пространства[7].

2. Маршрутизация.

   Вторая проблема вызвана плоской структурой адресного пространства IPv4 и практикой его произвольного распределения между клиентами. Это привело к стремительному росту таблиц маршрутизации у роутеров магистральных сетей: каждый из них должен содержать информацию обо всех подключенных к Интернет сетях.

3. Одним из основных способов решения проблемы нехватки адресного пространства стало широкое использование частного адресного пространства и технологий трансляции адресов (Network Address Translation, **NAT**). Однако эти технологии сами по себе стали проблемой, нарушив один из исходных постулатов построения IP-сетей, утверждающий, что взаимодействие между конечными пользователями осуществляется напрямую, без вмешательства посредников. Несмотря на то, что для большинства приложений были найдены приемлемые решения, использование NAT не позволяет обеспечить максимальной прозрачности при взаимодействии конечных узлов. Наибольшие проблемы технологии трансляции адресов возникают при использовании криптографических протоколов для обеспечения безопасности сетевых соединений и при установлении соединений с системами, расположенными за NAT.

Наряду с наличием перечисленных выше фундаментальных проблем, эксплуатация сетей IPv4 выявила наличие и других недостатков, которые более или менее успешно решались в рамках возможностей протокола. Можно отметить, например, следующие проблемы.

- Безопасность.

  Несмотря на то, что стек протоколов IP создавался с учётом возможности применения в армейских системах, он не содержит механизмов, обеспечивающих конфиденциальность передаваемых данных, поддерживающих криптографически стойкую аутентификацию и авторизацию. Такие механизмы появились в IP Security Protocol (IPSec), который может работать поверх IPv4.

- Автоконфигурация.

  Большинство технологий динамической автоконфигурации для IPv4 опираются на привязку к каким-либо заранее известным особенностям узлов (как правило, это

[7] 3 февраля 2011 года ICANN выдала последние 5 блоков адресов IPv4. В настоящее время IPv4 адреса остались только у региональных регистраторов.

MAC-адреса). Эти механизмы не всегда оказываются эффективными в использовании ограниченного адресного пространства и требуют предварительной ручной настройки администратором.

Разработка новых версий протоколов

Начиная с конца 80-х годов велась работа по разработке различных расширений протоколов стека TCP/IP или новых протоколов, которые должны были прийти им на замену. Предлагались следующие усовершенствования по сравнению с IPv4:

- расширение адресного пространства, в том числе с использованием иерархической организации для облегчения маршрутизации;
- упрощение заголовков для ускорения обработки пакетов;
- улучшение кодирования и использования опциональных частей заголовков;
- приемлемая для практической реализации поддержка QoS;
- поддержка аутентификации и конфиденциальности;
- поддержка различных способов автоконфигурации.

Результаты этой работы были обобщены в 1995 году в документе RFC1752 [23], “Recommendation for the IP Next Generation Protocol” («Рекомендации для протокола IP следующего поколения»). Первые RFC, описывающие следующую версию IP (IPv6), начали появляться уже с 1996 года и к настоящему моменту прошли несколько этапов переработки.

Особенности IP версии 6

По сравнению с IPv4 можно выделить следующие особенности IPv6.

Расширенное адресное пространство.

Наиболее очевидным отличием адресного пространства IPv6 является увеличение размера адреса с 32 до 128 бит.

Так же, как и в случае IPv4, адресное пространство IPv6 разделяется на несколько категорий, опираясь на значения старших битов адреса. На данный момент большая часть адресного пространства ещё не распределена. Распределённая часть адресов разделяется на два вида – адреса, распределяемые по подключению к провайдеру и адреса, распределяемые по географическому положению пользователя. Часть адресов выделена для мультикастовой передачи.

В IPv6 предусмотрено 4 типа обычных (unicast) адресов.

- **Global unicast** - глобальный уникальный IP адрес, который может использоваться во всей сети Internet без ограничений.
- **Link local unicast** - адрес, уникальный в пределах локальной сети. Может использоваться для автоконфигурации, обнаружения соседних компьютеров и работы без информации о роутинге. Эти адреса ни при каких условиях не должны транслироваться за пределы локальной сети.
- **Site local unicast** – адрес, уникальный в пределах сети некоторой организации. Не должны передаваться за пределы сети организации.
- **Unicast address with embedded IPv4 or encoded NSAP address** – адреса, предназначеные для поддержки совместимости с IPv4 и протоколом OSI Connectionless Network Protocol (CNLP).

В отличие от 4-ой версии, в IPv6 каждый сетевой интерфейс должен работать сразу с несколькими уникаст адресами: он должен, как минимум, отвечать на глобальные и link local адреса, а если используются site local, то и на них.

Упрощенный формат заголовков.

В отличие от заголовка IPv4, который включал 12 полей и мог иметь размер от 20 до 60 байт, заголовок IPv6 включает 8 полей и имеет фиксированный размер 40 байт. Сокращение размера заголовка произошло за счёт изменения некоторых особенностей работы прокола IP:

1. за счёт постоянного размера заголовка стало возможно отказаться от поля «размер заголовка»;
2. были удалены поля, связанные с фрагментацией пакетов;
3. было удалено как избыточное поле «контрольная сумма заголовка»;
4. опции протокола IP вынесены из заголовка.

В IPv6 возможность фрагментировать пакеты осталась только у отправителя, промежуточные узлы фрагментацию не осуществляют. Отказ от фрагментации потребовал, чтобы все используемые каналы позволяли передавать пакеты размером не менее 1280 байт. В случае, если это невозможно, должна быть предусмотрена процедура фрагментации и восстановления пакетов на уровне, более низком, чем IP. Кроме того, протокол предусматривает механизм определения максимально допустимого размера (MTU) пакета на маршруте. Если требуется отправка пакета, превышающего MTU, он фрагментируется отправителем, с использованием новых опций IP.

Упрощение заголовка IP повышает эффективность обработки пакетов маршрутизаторами.

Улучшенная поддержка расширений и опций.

Специальные возможности, реализованные в виде опций заголовка IPv4, в IPv6 вынесены за пределы основного заголовка IP пакета. Для них предусмотрена возможность добавлять в пакет дополнительные заголовки, следующие за основным. Такой подход позволяет промежуточным маршрутизатором обрабатывать пакеты с опциями так же, как и обычные пакеты. Дополнительная обработка производится только там, где это необходимо. В результате, специальные функции могут быть реализованы без существенного снижения производительности при обработке как обычных пакетов, так и пакетов с опциями.

Разработанный на данный момент перечень опций включает:

- задание маршрута доставки сообщения;
- фрагментацию;
- аутентификацию и другие криптографические возможности;
- туннелирование[8].

Безопасность.

В стандарте IPv6 введено обязательное требование поддержки протокола IPSec [24]. Этот протокол обеспечивает безопасную передачу данных на сетевом уровне. В настоящее время он широко используется для шифрования корпоративного трафика при его передаче через Интернет с использованием различных протоколов виртуальных частных сетей и туннелирования.

IPSec использует криптографически устойчивые методы для обеспечения следующих возможностей:

- шифрование;
- аутентификацию данных и отправителя;
- контроль доступа к чувствительным данным и частным сетям;

[8] Туннелирование – создание защищенного логического соединения между двумя конечными точками посредством инкапсуляции различных поротоколов для передачи через промежуточную сеть. При этом инкапсулируемый протокол относится к тому же или более низкому уровню, чем используемый в качестве тоннеля.

- проверка целостности данных, передаваемых протоколами без установки соединения (IP);
- защита от вмешательства в передачу данных на промежуточных узлах (когда данные могут быть приняты, расшифрованы, модифицированы и отправлены далее);
- ограничение возможностей взломщиков, получающих доступ к открытой части информации при перехвате пакетов на промежуточных узлах;
- гарантирует безопасность передачи IP пакетов между конечными узлами;
- безопасное туннелирование через небезопасные сети, включая Интернет и другие публичные сети;
- интеграции алгоритмов, протоколов и инфраструктуры безопасности.

Несмотря на то, что IPSec используется и в сетях IPv4, его обязательную реализацию в IPv6 можно рассматривать как нововведение, повышающее безопасность.

Другие возможности

Имеющиеся в IPv4 средства автоконфигурации в той или иной степени опираются на необходимость предварительной настройки. Система адресации IPv6 упрощает создание протоколов автоконфигурации, которые позволили бы подключить любой компьютер к сети, не требуя никаких настроек. Такой протокол автоконфигурации предусмотрен в RFC 4862 “IPv6 Stateless Address Autoconfiguration” [25].

Наряду с обычными (уникастными) и мультикастными адресами в IPv6 появилась поддержка так называемых аникастных (anycast) адресов. Такой адрес могут иметь сразу несколько систем, подключенных в различных участках сети. При этом пакеты, адресованные на такой адрес, будут доставляться ближайшему хосту. Реализация этого механизма наталкивается на ряд трудностей, которые задерживают его применение на практике.

Поддержка мобильных клиентов предусматривает возможность узлу получать пакеты на свой адрес, даже если он переместиться в другую сеть. В домашней сети такие узлы ведут себя как обычно, а при перемещении в другие сети получают пакеты через специального агента, расположенного в домашней сети. Реализация мобильных клиентов опирается на специальный протокол и дополнительные заголовки.

Вместе с протоколом IP изменился и протокол ICMP. Новая версия протокола ICMP была впервые определена в 1995 году в RFC1885 [26] и затем обновлена в 1998 году в RFC2463 [27] и в 2006 году в RFC4443 [28]. ICMP v6 фундаментально отличается по возможностям от своего предшественника. В него добавлены такие функции, как управление мультикастовой передачей, «поиск соседей» (включая отображение IPv6 адресов в MAC адреса), обеспечение поддержки мобильных клиентов.

Подробному рассмотрению протокола IPv6 посвящена работа [22].

2 Программирование сетевых приложений для стека TCP/IP

Одна из первых реализаций стека протоколов TCP/IP была разработана в университете Беркли (США), и реализована в операционной системе BSD UNIX. Разработанную тогда библиотеку и API называют Berkley Sockets, или просто Sockets. Последующие реализации, в том числе Windows Sockets в значительной степени повторяют этот интерфейс, что позволяет достаточно просто писать переносимые приложения на языке C с использованием этих библиотек. Так, для обеспечения переносимости между UNIX и Windows потребуется разный набор подключаемых заголовочных файлов и несколько вызовов функций для инициализации библиотеки Winsock, а основная часть программы может оставаться без изменения. Мы сначала рассмотрим классический интерфейс Berkley Sockets, а затем особенности реализации библиотеки Windows Sockets.

2.1 Работа с библиотекой

2.1.1 Подключение библиотеки

Для использования библиотеки Berkley Sockets вам, как обычно потребуется подключить несколько заголовочных файлов в тексте программы. На UNIX-системах все необходимые функции входят в стандартную библиотеку языка C (libc), поэтому подключение дополнительных библиотек не требуется.

Достаточный для простых программ набор заголовочных файлов может выглядеть, например, так[9]:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>     /* для функции inet_addr() */
#include <unistd.h>
```

2.1.2 Создание сокета

Для работы с TCP/IP нам необходим объект, который бы представлял в нашей программе сетевой сокет. Согласно принятому в библиотеке Berkley Sockets соглашению, такие объекты идентифицируются целым числом[10]. Так же, как и при работе с файлами, в первую очередь нам необходимо создать и инициализировать сокет. Делается это при помощи вызова функции `socket`:

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

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

Первый параметр функции socket определяет стек протоколов, который мы хотим использовать. Для стека IP это `AF_INET`. Второй параметр определяет вид соединения, и может принимать два значения: `SOCK_STREAM` для потоковых соединений (например, с протоколом TCP) и `SOCK_DGRAM` для протоколов, передающих отдельные пакеты (указывайте при использовании протокола UDP). Последний параметр определяет

[9] В примерах в этом пособии будет использована функция `memset()`, которая определена в заголовочном файле `string.h`, а также функции ввода-вывода из `stdio.h`.

[10] В Windows для этого был введён специальный тип `SOCKET`, который определён как `int`.

конкретный протокол, который мы хотим использовать. В нашем случае это IPPROTO_TCP для протокола TCP.

Функция socket возвращает значение, идентифицирующее сокет в системе. В случае ошибки возвращается значение INVALID_SOCKET.

2.1.3 Представление сетевых адресов

Библиотека Sockets разрабатывалась с учётом возможности поддержки различных сетевых протоколов (и не только сетевых, она используется, например и для работы с локальными UNIX-сокетами). Поэтому её разработчики должны были предусмотреть возможность поддержки адресов различной структуры. Поскольку библиотека разрабатывалась для языка C без применения объектно-ориентированного программирования, то для решения этой задачи авторы использовали подход, при котором сетевой адрес хранится в специальной структуре, передаваемой в функции библиотеки Sockets по указателю. При объявлении функций, работающих с адресом, используется тип struct sockaddr, который выступает в роли «базового класса» по терминологии ООП[11]. При написании программ вместо структуры этого типа нужно использовать конкретные структуры, соответствующие используемому типу сетевых протоколов. Библиотека определяет тип структуры по её первому элементу и размеру (который передаётся отдельным параметром).

Для TCP/IP v4 структура для хранения адреса имеет тип struct sockaddr_in:

```
#include <netinet/in.h>

struct sockaddr_in{
   uint8_t sin_len;          // отсутствует во многих системах[12]
    sa_family_t sin_family;
    in_port_t sin_port;
    struct in_addr sin_addr;
    char sin_zero[8];         // опционально в POSIX[13]
};
```

Последний параметр применяется для выравнивания размера структуры и должен быть заполнен нулями. Остальные элементы несут смысловую нагрузку: sin_family определяет стек протоколов, и должен быть равен AF_INET для TCP/IP. Порт для соединения (на удаленном компьютере) указывается в параметре sin_port. IP адрес удаленного сервера задается структурой sin_addr, имеющей тип in_addr:

```
struct   in_addr {
  union   {
             struct  {
                        uint8_t s_b1,     // 8 бит
                                s_b2,     // 8 бит
                                s_b3,     // 8 бит
                                s_b4;     // 8 бит
                     }  S_un_b;
             struct  {
```

[11] На самом деле тогда ещё не появился даже стандарт ANSI C, согласно которому этот указатель можно было бы объявить как void *.

[12] Это поле появилось в BSD 4.3 при добавлении поддержки протоколов OSI. Оно является опциональным согласно стандарту POSIX и не реализовано, например, в Windows и Linux. При программировании не требуется заполнять или как-либо использовать это поле. Для корректной работы перед использованием структуры sockaddr_in стоит сделать memset(&addr, 0, sizeof(addr));

[13] Это поле также не является обязательным, но его применение более распространено.

```
                        uint16_t  s_w1,   // 16 бит
                                  s_w2;   // 16 бит
                    }  S_un_w;
            uint32_t  S_addr;             // 32 бита
        } S_un;
};
```

Эта структура позволяет получить доступ к IP-адресу как к одному 32-битному числу (`S_addr`), двум 16-битным (`s_w1`, `s_w2`) или четырем отдельным байтам (`s_b1`,...,`s_b4`).

В некоторых новых реализациях[14] (начиная с BSD 4.2) отказались от использования объединения и определяют эту структуру как содержащую один элемент:

```
#include <netinet/in.h>

struct in_addr {
   in_addr_t s_addr;   // 32 бита
};
```

Необходимо отметить, что все поля в структурах `sockaddr_in` и `in_addr` необходимо заполнять с использованием сетевого порядка байтов. Для преобразования данных из обычного порядка в сетевой предусмотрены специальные функции:

```
#include <arpa/inet.h> // или <netinet/in.h>

uint32_t htonl( uint32_t hostlong);
uint16_t htons( uint16_t hostshort);
```

Функция `htonl()` предназначена для преобразования 32-битных чисел, а `htons()` – для 16-битных. Обратное преобразование осуществляют функциями `ntohl()` и `ntohs()` соответственно.

Еще одна полезная функция - `inet_addr`[15], преобразует IP-адрес, записанный текстовой строкой, в 32-битное число в сетевом порядке:

```
in_addr_t inet_addr(const char *cp);
```

Заметим, что эта функция может преобразовывать только адреса, записанные в форме четырех чисел, разделенных точками (например “`127.0.0.1`”). Преобразование доменных имён она не выполняет.

Все функции библиотеки sockets, которые получают на вход сетевой адрес в виде структуры `sockaddr`, требуют также и передачи размера этой структуры отдельным параметром. При этом если адрес передаётся от пользовательского процесса ОС (как, например, при запросе соединения с удалённым хостом), то передача размера осуществляется по значению:

```
struct sockaddr_in serv;
/* Заполнить структуру serv{} */
connect(sock,(struct sockaddr *) &serv,
                        sizeof(serv)); // передача по значению
```

В случае если подразумевается передача адреса от ОС в пользовательский процесс (система сообщает адрес подключившегося к нам клиента), то передача размера происходит по указателю. При этом перед вызовом необходимо присвоить передаваемому значению размер доступной памяти, иначе функция выдаст ошибку:

---

[14] В Windows используется «старый» вариант с объединением.

[15] Недостатком этой функции является то, что возвращаемый в случае неправильно форматированной строки код `INADDR_NONE` (как правило, `0xFFFFFFFF`) является корректным широковещательным ip-адресом 255.255.255.255. Для избежания этой коллизии рекомендуется использовать более новую функцию `inet_aton` или функцию `inet_pton`, которая может работать как с IPv4 так и с IPv6 адресами. К сожалению обе эти функции не доступны под Windows.

```
Struct sockaddr_in cli;
socklen_t len;
len = sizeof(cli); // в len указываем размер доступной памяти
getpeername(sock,(struct sockaddr *) &cli,
                                  &len);   // передача по ссылке
// значение len может быть изменено
```

Передача адреса всегда осуществляется по ссылке.

2.2 Открытие активного соединения

После того, как мы создали сокет, можно с его помощью соединиться с удаленным компьютером. Для этого используем функцию `connect`:

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

int connect(int s, const struct sockaddr *name, socklen_t
namelen);
```

Первым параметром является сокет, который мы хотим соединить. Второй – адрес сервера, заданный структурой `sockaddr_in` для TCP/IP. Последним параметром передается размер структуры с адресом.

Функция `connect` возвращает `0` при успешной установке соединения. В случае ошибки возвращается отрицательное значение. Необходимо понимать, что функции `connect` необходимо дождаться завершения процесса сетевого соединения, поэтому ее выполнение может занять значительное время, на которое ваша программа будет заблокирована. Также необходимо всегда проверять возвращаемое значение, поскольку процесс установки сетевого соединения часто заканчивается ошибкой.

Собрав все наши знания вместе, мы получим примерно такой фрагмент кода для установки соединения с телнет-сервером на локальной машине:

```
#define CONNECTION_PORT 23      // Telnet

int sock;
sockaddr_in serv_addr;

sock = socket (AF_INET, SOCK_STREAM, IPPROTO_TCP);

memset(&serv_addr, 0, sizeof(serv_addr));

serv_addr.sin_family = AF_INET;
serv_addr.sin_port = htons(CONNECTION_PORT);
serv_addr.sin_addr.s_addr = inet_addr("127.0.0.1");

if(connect(sock,(struct sockaddr *)&serv_addr,
                                   sizeof(serv_addr)) < 0)
   printf("Connection failed.\n");
```

2.2.1 Обмен данными

После того, как установлено соединение, можно обмениваться данными с удаленной системой. Для этого имеются две функции: `send` для отправки данных и `recv` для получения.

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

```
ssize_t send(int s, const void *msg, size_t len, int flags);

ssize_t recv(int s, void *buf, size_t len, int flags);
```

Первым параметром этих функций является сокет, через который будет производиться обмен данными. Второй параметр, `buf`, является указателем на область с данными, которые необходимо передать или принять. Параметр `len` определяет число передаваемых или принимаемых данных. Параметр `flag` позволяет регулировать некоторые особенности обработки данных, так, например, значение `MSG_OOB` позволяет передать или получить «срочные» (out of band) данные, которые могут быть приняты удаленной стороной быстрее, чем еще имеющиеся в сетевых буферах обычные данные. Функция `recv` поддерживает флаг `MSG_PEEK`, позволяющий получить данные, не удаляя их из системного буфера, так что они будут доступны и для следующего вызова `recv`.

В случае успешного завершения операции обе функции возвращают число переданных или отправленных байтов. В случае ошибки возвращается значение `SOCKET_ERROR`. Если соединение было закрыто удаленной стороной и в буфере приема больше не осталось входящих данных, то функция `recv` вернет 0.

Успешный вызов функции `send` на самом деле не гарантирует, что все данные были доставлены удаленной системе. Она возвращает управление после того, как все переданные ей данные будут размещены в системном буфере стека протоколов TCP/IP. При этом часть данных может быть уже принята удаленным компьютером, часть может находиться в пути, а часть оставаться на вашем компьютере. Помешать передать данные до конца может множество событий: от сбоев в сети или на одном из компьютеров, до некорректного закрытия сокета вашей программой.

В обычном режиме функции `send` и `recv` блокируют работу компьютера, до тех пор, пока не будет выполнена запрошенная операция, например, пока не будет полностью получен по сети запрошенный объем данных. Такое поведение может быть неудобно при написании программ. Первый возможный вариант решения – использовать многопоточное программирование. Например, сервер может создавать по отдельному потоку для каждого клиентского соединения. У протокола TCP/IP прием и передача данных абсолютно независимы, поэтому мы можем без каких-либо последствий читать данные из сокета в одном потоке и записывать в другом. Второй вариант – использовать неблокирующий (non-blocking) режим работы, о котором будет рассказано позднее. Кроме того, имеется функция select, которая позволяет определить, имеются ли новые данные для приема, или свободное место в буферах для отправки:

```
#include <sys/select.h>

int select(int nfds, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);
```

Эта функция работает с тремя наборами сокетов: набор `readfds` проверяется на возможность выполнения операции чтения, `writefds` – записи, а `exceptfds` – на наличие ошибок или срочных (OOB) данных. Параметр `nfds` задаёт максимальное значение номера сокета, которое может содержаться в наборах (проверяются значения с `0` по `nfds-1`)[16]. Функция `select` будет ожидать выполнения одного из условий в течение времени, заданного в структуре `timeval`[17]:

```
struct timeval {
```

[16] Некоторые современные реализации игнорируют этот параметр.

[17] В Linux вызов `select` модифицирует этот параметр.

```
    long    tv_sec;         // seconds
    long    tv_usec;        // and microseconds
};
```

Для реализации «бесконечного» ожидания необходимо передать вместо указателя на эту структуру значение NULL.

Однако в первую очередь необходимо передать этой функции сокеты, которые мы хотим проверять. Для этого используются специальные переменные, наборы fd_set, которые позволяют передать одновременно несколько сокетов для каждой операции. Для работы с наборами, необходимо определить переменную типа fd_set и подготовить её с помощью специальных макросов:

```
#include <sys/select.h>

FD_ZERO(*set)
FD_SET(s, *set)
FD_CLR(s, *set)
FD_ISSET(s, *set)
```

Здесь *set – указатель на переменную типа fd_set, а s – SOCKET. Макрос FD_ZERO очищает набор (рекомендуется использовать перед началом работы), FD_SET добавляет сокет в набор, FD_CLR удаляет сокет из набора, а FD_ISSET проверяет, есть ли сокет в наборе (возвращает логическое значение true если s входит в набор set).

Функция select возвращает 0 в случае истечения времени ожидания, SOCKET_ERROR в случае ошибки и число готовых к работе сокетов в случае успешного завершения работы. При этом select модифицирует полученные на вход наборы таким образом, что остаются установленными только те сокеты, которые готовы к выполнению соответствующей операции.

2.2.2 Завершение соединения

Закончив обмен данными, необходимо корректно закрыть соединение. Если соединение будет закрыто некорректно, то могут быть потеряны передаваемые данные и удаленная система своевременно не узнает о закрытии соединения. Корректное завершение работы подразумевает, во-первых, обмен специальными сообщениями о завершении связи и, во-вторых, освобождение системных ресурсов на локальной машине.

Первая стадия этой операции выполняется функцией shutdown:

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

int shutdown(int s, int how);
```

Параметр s идентифицирует сокет, соединение по которому должно быть закрыто. Параметр how определяет, как надо закрыть соединение, он может принимать значения SHUT_WR, SHUT_RD и SHUT_RDWR[18]. Тут вновь проявляется полная независимость приемной и передающей частей протокола TCP/IP: можно закрыть сокет для передачи данных, но оставить его открытым для приема использовав значение SHUT_WR. Или наоборот, вызов функции со значением SHUT_RD закроет сокет для приема данных, оставив его открытым для записи. Значение SHUT_RDWR предписывает закрыть соединение в обе стороны. Например, запрашивая страницу с web-сервера можно отправить запрос, закрыть сокет на передачу, принять данные от сервера и после этого закрыть сокет на прием.

[18] В Windows используются значения SD_RECV,SD_SEND и SD_BOTH соответственно.

После того, как сетевое соединение будет закрыто, можно освободить системные ресурсы, закрыв сокет. Делается это функцией `close`[19]:

```
#include <unistd.h>

int close(int s);
```

Функция принимает в качестве параметра дескриптор закрываемого сокета и возвращает `0` в случае успешного выполнения, `SOCKET_ERROR` в случае ошибки.

2.2.3 Пример программы, устанавливающей активное соединение

Рассмотрим пример программы, устанавливающей соединение с web-сервером, запрашивающей корневую директорию и печатающую ответ сервера.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <string.h>

int main(int argc,char * argv[])
{
   int s;
   sockaddr_in addr;
   int i;
   char str[101];

   if(argc<2)
   {
        printf("enter IP address of server as parameter");
        return  100;
   }

   // создаём сокет
   s=socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);
   if(s==INVALID_SOCKET)
   {
        printf("socket() error\n");
        return 1;
   }

   // заполняем структуру с адресом сервера
   memset(&addr,0,sizeof(addr));
   addr.sin_family=AF_INET;
   addr.sin_port=htons(80); // для web-серверов обычно
                            // используется порт 80
   addr.sin_addr.S_un.S_addr=inet_addr(argv[1]);
                            // IP адрес сервера задаётся
                            // в командной строке

   // устанавливаем соединение
```

[19] В Windows – `closesocket()`

```
    if(connect(s,(const sockaddr*)&addr,sizeof(addr))!=0)
    {
         printf("connect() error\n");
         close(s);
         return 2;
    }

    // посылаем запрос согласно протоколу HTTP
    send(s,"GET /\n\n",7,0);
    shutdown(s,SD_SEND);     // закрываем соединение на передачу

    do
    {
         i=recv(s,str,100,0); // получаем данные
         if(i>0)
         {
              str[i]='\0';
              printf("%s",str);
         }
    } while(i>0);  // при завершении передачи данных сервером
                   // функция recv вернёт 0

    shutdown(s,SD_BOTH); // окончательно закрываем соединение
    close(s);            // закрываем сокет
    return 0;
  }
```

2.3 Прием входящих соединений

Для приема входящих соединений нам также нужно в первую очередь создать сокет с помощью функции `socket`. В случае исходящего соединения нам было не принципиально, какой локальный порт будет использован системой для открытия соединения. В случае же приема входящих соединений нужно указать локальный адрес, используя функцию `bind`:

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

int bind(int s, const struct sockaddr *name, socklen_t namelen);
```

Также как и при вызове функции `connect`, локальный адрес передается с помощью указателя `name`, на структуру типа `sockaddr_in`. Эта функция позволяет задать не только порт, но и адрес локального интерфейса (их может быть несколько), который будет связан с сокетом. Если это не требуется, то можно вместо локального IP-адреса задать значение `INADDR_ANY`, которое укажет системе на необходимость использовать все доступные интерфейсы.

Функция `bind` может быть использована и для задания конкретного значения локального адреса для исходящих соединений, в этом случае она должна быть вызвана до вызова `connect`.

После того, как мы связали сокет с локальным адресом, мы можем перевести сокет в режим приема соединений (режим пассивного соединения протокола TCP). Это делается с помощью функции `listen`:

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

```
int listen(int s, int backlog);
```

Параметр s является дескриптором сокета, который переводится в состояние приема, backlog определяет размер системной очереди входящих соединений.

После успешного вызова listen сокет готов к приему входящих соединений. Как только какая-нибудь удаленная система запросит соединение с нашим адресом, мы сможем установить с ней соединение. Для приема входящего соединения используется функция accept:

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

int accept(int s, struct sockaddr *addr, socklen_t *addrlen);
```

Этой функции необходимо передать дескриптор сокета s, который был переведен в состояние приема соединений. Узнать адрес удаленной системы, с которой происходит соединение, можно передав в параметре addr указатель на структуру sockaddr_in, а в параметре addrlen – указатель на переменную, в которой указан размер этой структуры. Если оба этих параметра заданы, то функция accept запишет адрес удаленной системы по указанному адресу. Если адрес удаленной системы не требуется, то можно указать значение NULL.

При вызове функции accept создается новый сокет, который и должен использоваться для обмена данными с удаленной системой. Его дескриптор является возвращаемым значением этой функции. Исходный сокет (который был переведен в режим приема соединений) для обмена данными не используется, он готов для приема следующего соединения с помощью еще одного вызова accept. Сокет, который возвращает функция accept, полностью готов к передаче данных с помощью send/recv и никаких дополнительных подготовительных операций для работы с ним не требуется. В случае ошибки функция accept возвращает значение INVALID_SOCKET.

По умолчанию, функция accept блокирует работу вызвавшего ее потока до тех пор, пока не будет принято входящее соединение. Если это неприемлемо, то можно проверить, имеются ли входящие соединения, передав дескриптор принимающего сокета функции select в списке сокетов, готовых к чтению (readfds).

2.3.1 Пример приёма соединения

Ниже приведён пример программы, которая принимает входящие соединения на порт 1234, передаёт клиенту текст «Hello!» и закрывает соединение. В момент подключения клиента его адрес и порт печатаются на консоли.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <string.h>

void main(int argc,char *argv[])
{
  int s,client;
  sockaddr_in addr;
  int val,len;

  // создаём сокет для приёма соединений
  s=socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);
```

```
    if(s==INVALID_SOCKET)
    {
         printf("socket() error\n");
         return 1;
    }

    // разрешаем повторное использование локального порта
    val=1;
    setsockopt(s,SOL_SOCKET,SO_REUSEADDR,&val,sizeof(val));

    // заполняем структуру локальным адресом
    memset(&addr,0,sizeof(addr));
    addr.sin_family=AF_INET;
    addr.sin_port=htons(1234);    // порт для приёма соединений
    addr.sin_addr.S_un.S_addr=INADDR_ANY;
                  // используются все локальные интерфейсы

    // связываем сокет с локальным портом
    if(bind(s,(sockaddr*)&addr,sizeof(addr))!=0)
    {
         printf("bind() failed\n");
         close(s);
         return 2;
    }

    // переводим сокет в режим пассивного соединения
    if(listen(s,10)!=0)
    {
         printf("listen() failed\n");
         close(s);
         return 3;
    }

    for(;;)   // соединения принимаются в бесконечном цикле
    {
         // необходимо указать размер структуры адреса
         len=sizeof(addr);

         // принимаем входящее соединение
         client=accept(s,(sockaddr*)&addr,&len);
         printf("incoming connection from %d.%d.%d.%d:%d\n",
              addr.sin_addr.S_un.S_un_b.s_b1,
              addr.sin_addr.S_un.S_un_b.s_b2,
              addr.sin_addr.S_un.S_un_b.s_b3,
              addr.sin_addr.S_un.S_un_b.s_b4,
              ntohs(addr.sin_port));

         // посылаем привет клиенту
         send(client,"hello!",6,0);

         // закрываем соединение и клиентский сокет
         shutdown(client,SD_BOTH);
         close(client);
```

```
            printf("incoming connection closed\n");
        }
        return 0;
    }
```

В этой программе использовалась функция `setsockopt()`, с помощью которой можно устанавливать различные режимы работы сокета. В данном случае она была использована, чтобы включить разрешение на многократное использование локальных портов. Если этого не сделать, то только один сокет может быть связан с одним портом. При попытке использовать порт повторно, функция `bind()` вернёт ошибку.

Во время отладки программ и их аварийного завершения операционная система не всегда сразу освобождает порты, использованные разрабатываемой программой. В этом случае использование режима повторного использования позволяет сразу запустить отлаживаемую программу, не ожидая освобождения портов. Для финальной версии программы такой режим обычно не требуется.

2.3.2 Задания для самостоятельного выполнения

- Проверьте работоспособность программы-сервера.
- Попробуйте запустить несколько копий программы, посмотрите, какая из них обрабатывает входящие соединения. Запускайте новые копии сервера и закрывайте старые во время эксперимента в разном порядке. Какая из копий обрабатывает входящие соединения? Отключите режим совместного использования порта и попробуйте снова.
- Модифицируйте код сервера так, чтобы он распечатывал передаваемые клиентом данные. Соединитесь с вашим сервером с помощью Web-browser'а и посмотрите, какую информацию он передаёт.
- Попробуйте отправить web-browser'у html документ так, чтобы он корректно отобразил его.

2.4 Работа в неблокирующем режиме

По умолчанию, большинство функций, работающих с сокетами, таких как `connect`, `send`, `recv`, `accept` блокируют выполнение вызвавшего их потока до завершения выполнения операции. При работе с сетью такие задержки могут быть очень большими, а в случае функции `accept` – вообще потенциально бесконечными, если никто не захочет соединиться с нашим сокетом. Хотя такой режим может быть удобен для написания простейших программ, во многих случаях он оказывается неприемлем. Функция `select` облегчает ситуацию, однако она не позволяет определить, сколько точно байт доступно для чтения или сколько места имеется в системных буферах отправки данных. Поэтому, если мы запросим больше данных, чем доступно, вызов `recv` (или `send`) может все равно быть заблокирован.

Полностью предотвратить блокирование нашего потока мы можем, переведя сокет в неблокирующий (non-blocking) режим. Для этого используется функция `fcntl`[20]:

```
#include <fcntl.h>
int fcntl(int s, long cmd, ...);
```

Эта функция может выполнять различные операции, в зависимости от значения параметра `cmd`. Для включения неблокирующего режима необходимо установить флаг `O_NONBLOCK`. Текущее значение флагов может быть прочитано с помощью команды

[20] В Windows этот режим может быть включен функцией `ioctlsocket()` с параметром `FIONBIO`.

F_GETFL, а установлено с помощью F_SETFL. Типичный код для переключения сокета в неблокирующий режим выглядит так:

```
int flag;
int sock; // сокет, необходимо сначала создать
...
if ((flag = fcntl(sock, F_GETFL, 0)) >= 0)
    flag = fcntl(sock, F_SETFL, flag | O_NONBLOCK);
if (flag < 0) {
  // ошибка
  }
```

В неблокирующем режиме все функции работы с сокетами возвращают управление немедленно, вне зависимости от того, полностью ли выполнена запрошенная операция. Функции передачи и приема данных в этом режиме вернут число отправленных или переданных данных, которое в данном случае может принимать значение от `0` до запрошенного размера. Программист должен учитывать это и соответствующим образом обрабатывать не полностью выполненные запросы. Завершение установления соединения или поступление нового соединения при работе в неблокирующем режиме может быть проверено с помощью функции `select`.

Функция `setsockopt` также позволяет задать несколько параметров[21], которые воздействуют на блокирующее поведение функций `send` и `recv`.

- `SO_RCVTIMEO`
  Максимальный таймаут для функции `recv`. Функция не будет блокировать процесс на время, больше установленного этим параметром. По истечению таймаута будет возвращено число принятых байт или установлен код ошибки `EWOULDBLOCK`, если никакие данные приняты не были.
- `SO_RCVLOWAT`
  Минимальное число байт, получения которого будет ждать блокирующий вызов `recv`. Значение по умолчанию 1.
- `SO_SNDTIMEO`
  Максимальный таймаут для функции `send`. Функция не будет блокировать на большее время и вернёт число переданных байт или установит код ошибки `EWOULDBLOCK`.
- `SO_SNDLOWAT`
  Минимальное число байт для операций вывода. Функция `select` вернёт сокет как готовый для записи, если возможно без блокирования отправить указанный объём данных.

2.4.1 Пример использования неблокирующего режима и функции select

Рассмотрим пример программы, использующей функцию `select` и неблокирующий режим работы для одновременной работы с большим количеством сетевых соединений в одном потоке. Наша программа будет представлять собой сервер, перекодирующий поступающий от клиентов текст в верхний регистр и отправляющий его назад.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
```

[21] Данные параметры не доступны в Windows.

```
#include <string.h>
#include <vector>   // будем использовать vector для хранения
                    // активных соединений
using namespace std;
```

Для отправки данных через сокет в неблокирующем режиме потребуется буфер, в который будут помещаться данные для отправки. Поскольку детали реализации буфера не имеют прямого отношения к работе с сетью, он не будет рассмотрен подробно, будем считать, что он реализован в классе со следующим интерфейсом.

```
class buffer
{
public:
   buffer()
   ~buffer();

   bool AddData(void *ptr,int len);
     // Добавить данные в конец буфера:
     // ptr – указатель на начало данных
     // len – размер данных в байтах

   int ReadySize();
     // Возвращает размер данных, доступных для отправки.
     // Если данных нет, возвращается 0.

   const char *ReadyData();
     // Возвращает указатель доступные для отправки данные.
     // Размер этих данных возвращается ReadySize().
     // Если данных нет, возвращается NULL.

   void RemoveData(int len);
     // Удаляет данные из начала буфера.
     // len – размер данных для удаления, для корректной работы
     // должен быть <= значения возвращенного ReadySize().
};
```

В сервере будут использоваться два вида сокетов: сокеты для обмена данными с клиентами и один сокет для приёма соединений. Чтобы избежать блокировки все они должны находится в неблокирующем режиме и опрашиваться с помощью функции `select`. Оба вида сокетов будут инкапсулированы в отдельных классах, а следующий класс будет являться для них базовым, определяя общий интерфейс:

```
class PollSocket
{
public:

     PollSocket() { _sock=INVALID_SOCKET; }

     virtual ~PollSocket()    // деструктор – закроем сокет
     {
          if(_sock!=INVALID_SOCKET)
               closesocket(_sock);
     }
```

```
     virtual bool PrepareForSelect(fd_set *read_fs,
                                   fd_set *write_fs, int *max)
     // функция должна поместить сокет в набор на чтение или на
     // запись в зависимости от необходимости, и скорректировать
     // переменную max, в которой находится максимальный номер
     // установленного в наборах дескриптора.
     // Возвращаемое значение: true – операция выполнена
     // false – операция не выполнена, объект надо уничтожить
     // (например после закрытия соединения с клиентом).
     { return false; }
     virtual bool PerformOperation(fd_set *read_fs,
                                         fd_set *write_fs)
     // Выполнить операции чтения или записи если наш сокет
     // присутствует в соответствующих наборах.
     // Возвращаемое значение: true – операция выполнена
     // false – операция не выполнена, объект надо уничтожить
     { return false; }

     bool MakeNonBlocking()
     // переводит сокет в неблокирующий режим
     {
          unsigned long op=1;
          return ioctlsocket(_sock,FIONBIO,&op)==0;
     }

protected:
     SOCKET _sock; // сокет, с которым мы работаем.
};

vector<PollSocket *> Sockets;
// динамический массив, в котором будут хранится рабочие сокеты
```

Теперь приступим к реализации обработчика клиентского соединения. Процедура чтения в этом случае большой сложности не представляет: она будет принимать данные в доступном объеме и выполонять обработку – переводить символы в верхний регистр. Однако с отправкой придётся быть чуть более осторожным – поскольку сокет работает в неблокирующем режиме, мы не можем надеяться что все данные, которые потребуется отправить обратно, удастся сразу же отправить. Поэтому придётся использовать промежуточный буфер для хранения исходящих данных, откуда они и будут по возможности отправлятся.

```
#define RCV_BUFFER 1024 // размер буфера приёма данных

class ClientConnection: public PollSocket
{
public:
     ClientConnection(int accept_sock);
     // конструктор – принимаем соединение
     // accept_sock – дескриптор сокета, с которого нужно принять
     // соединение
     ~ClientConnection();

     virtual bool PrepareForSelect(fd_set *read_fs,
                                   fd_set *write_fs,int *max);
```

```
     // будем регистировать наш сокет на чтение – всегда,
     // на запись если имеются данные в выходном буфере

     virtual bool PerformOperation(fd_set *read_fs,
                                   fd_set *write_fs);
     // будем выполнять операции записи или чтения

protected:
     bool DoRead();
     // Читаем и обрабатываем данные от клиента.
     // Возвращает false если соединение закрыто.

     bool DoWrite();
     // Отправляем данные из буфера.

     buffer _out;              // буфер исходящих данных
     struct sockaddr_in _addr; // адрес клиента
};

ClientConnection::ClientConnection(SOCKET accept_sock)
{
     int len=sizeof(_addr);

     memset(&_addr,0,sizeof(_addr));

     // принимаем входящее соединение
     _sock=accept(accept_sock,(sockaddr*)&_addr,&len);
     if(_sock!=INVALID_SOCKET)
     {
          // печатаем адрес клиента
          printf("Accepted connection from %s:%d\n",
               inet_ntoa(_addr.sin_addr),ntohs(_addr.sin_port));

          // переводим сокет в неблокирующий режим
          MakeNonBlocking();
     }
     else
     {    // ошибка приёма соединения
          printf("Accept() failed!\n");
     }
}

ClientConnection::~ClientConnection()
{
     if(_sock!=INVALID_SOCKET)
     {
          // закрываем сокет и печатаем сообщение
          closesocket(_sock);
          printf("Closed connection with %s:%d\n",
                inet_ntoa(_addr.sin_addr),ntohs(_addr.sin_port));
     }
     else
          printf("Closing invalid client object");
```

```
}

bool ClientConnection::PrepareForSelect(fd_set *read_fs,
                                        fd_set *write_fs, int * max)
{
     // регистрируем наш сокет в наборах для вызова select()

     // если у нас неправильный сокет (не сработал accept)
     // то объект надо уничтожить - возвращаем false.
     if(_sock==INVALID_SOCKET) return false;

     // регистрируем на запись если есть что отправлять
     if(_out.ReadySize()>0)   FD_SET(_sock,write_fs);
     // регистрируем на чтение всегда
     FD_SET(_sock,read_fs);
     // корректируем максимальное значение
     if(_sock>*max) *max=_sock;

     return true;   // всё в порядке
}

bool ClientConnection::PerformOperation(fd_set *read_fs,
                                             fd_set *write_fs)
{
     // выполняем операции, к которым наш сокет готов
     if(_sock==INVALID_SOCKET) return false;

     // проверяем и выполняем чтение.
     if(FD_ISSET(_sock,read_fs))
          if(!DoRead()) return false; // если соединение закрыто
                                      // наш объект можно удалять
     // проверяем и выполняем запись
     if(FD_ISSET(_sock,write_fs))
          DoWrite();

     return true;
}

bool ClientConnection::DoRead() // чтение и обработка
{
     int len,i;
     char buff[RCV_BUFFER];

     // принимаем данные с сокета
     len=recv(_sock,buff,RCV_BUFFER,0);

     if(len>0) // данные приняты
     {
          // обрабатываем - переводим в верхний регистр
          for(i=0;i<len;i++)
               buff[i]=toupper(buff[i]);

          // помещаем обработанные данные в буфер отправки
```

```
          _out.AddData(buff,len);

          return true;
     }
     else      // ошибка или соединение закрыто
          return false;
}

bool ClientConnection::DoWrite()   // отправка данных
{
     int len;

     // отправляем
     len=send(_sock,_out.ReadyData(),_out.ReadySize(),0);
     if(len>0) // len байт отправлено, это может быть меньше,
               // чем мы попросили (_out.ReadySize()).
               // Удаляем из буфера отправленное, остальное,
               // если есть, отправим в следующий раз.
               _out.RemoveData(len);

     return len!=SOCKET_ERROR;
}
```

Класс `Acceptor` реализует работу с сокетом, принимающим соединения. Он обрабатывает только одну операцию – чтение, которая сигнализирует о поступлении нового соединения. Для обработки соединения будем создавать новый экземпляр класса `ClientConnection`.

```
#define SERVER_PORT 3000    // номер порта для приёма соединений

class Acceptor: public PollSocket
{
public:
     Acceptor(); // создаем сокет и переводим его в режим приёма
     ~Acceptor();

     virtual bool PrepareForSelect(fd_set *read_fs,
                                   fd_set *write_fs,int *max);
     // регистрируемся в списке на чтение

     virtual bool PerformOperation(fd_set *read_fs,
                                           fd_set *write_fs);
     // принимаем соединение
};


Acceptor::Acceptor()
{
     // создаём сокет
     _sock=socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);

     if(_sock==INVALID_SOCKET)
     {
          printf("Socket() failed\n");
          return;
```

```
     }

     // разрешаем повторное использование порта
     BOOL val=TRUE;
     setsockopt(_sock,SOL_SOCKET,SO_REUSEADDR,(char*)&val,
                                                  sizeof(val));
     // закрепляемся за портом
     struct sockaddr_in addr;
     memset(&addr,0,sizeof(addr));
     addr.sin_family=AF_INET;
     addr.sin_addr.S_un.S_addr=INADDR_ANY;
     addr.sin_port=htons(SERVER_PORT);

     if(bind(_sock,(sockaddr*)&addr,sizeof(addr))!=0)
     {
          printf("bind() failed\n");
          closesocket(_sock);
          _sock=INVALID_SOCKET;
          return;
     }

     // переходим в режим приёма соединений
     if(listen(_sock,10)!=0)
     {
          printf("listen() failed\n");
          closesocket(_sock);
          _sock=INVALID_SOCKET;
          return;
     }
}

Acceptor::~Acceptor()
{
     if(_sock!=INVALID_SOCKET)     // закрываем сокет
     {
          closesocket(_sock);
          printf("Closed accepting socket\n");
     }
     else
          printf("Closing invalid accepting socket object");
}

bool Acceptor::PrepareForSelect(fd_set *read_fs,
                                fd_set *write_fs,int * max)
{
     if(_sock==INVALID_SOCKET) return false;

     // регистрируемся только на чтение (приём соединения)
     FD_SET(_sock,read_fs);
     if(_sock>*max) *max=_sock;

     return true;
}
```

```
bool Acceptor::PerformOperation(fd_set *read_fs,
                                          fd_set *write_fs)
{
     if(_sock==INVALID_SOCKET) return false;

     // если сокет готов на чтение значит есть соединение
     if(FD_ISSET(_sock,read_fs))
     {
          // создаём клиентский объект, он вызовет accept()
          ClientConnection *client=new ClientConnection(_sock);
          // заносим клиента в список активных сокетов
          Sockets.push_back(client);
     }

     return true;
}
```

Теперь все объекты готовы. В функции `main` остаётся создать объект-обработчик входящих соединений и поместить его в список рабочих.

Далее программа будет в цикле регистрировать рабочие сокеты в списках проверки готовности на чтение и запись и передавать эти списки функции `select`. Когда она вернёт управление, в списках останутся только те дескрипторы, которые готовы к выполнению запрошенной операции. Программа ещё раз пройдет по списку рабочих объектов и попросит их выполнить необходимые действия.

Если один из объектов вернёт false, его надо будет удалить из списка.

```
int main(int argc, char* argv[])
{
     fd_set read_fs,write_fs;
                         // наборы дескрипторов на чтение/запись
     int max;            // максимальный дескриптор в наборе
     int i;

     // создаем обработчик входящих соединений
     Acceptor *a=new Acceptor();
     // добавляем его в список
     Sockets.push_back(a);

     // Пока у нас есть рабочие объекты работаем.
     // Так как наш acceptor() никогда не возвращает false
     // в нормальной ситуации, то выход из этого цикла
     // возможен только в случае ошибки создания принимающего
     // сокета
     while(!Sockets.empty())
     {
          // стираем наборы
          FD_ZERO(&read_fs);
          FD_ZERO(&write_fs);
          max=0;

          // готовим списки сокетов на чтение и запись
          for(i=0; i!=Sockets.size() ; i++)
          {
```

```
                if(!Sockets[i]->PrepareForSelect(&read_fs,
                                                 &write_fs, &max))
                {
                     // уничтожаем текущий объект
                     delete Sockets[i];
                     Sockets.erase(Sockets.begin()+i);
                     i--; // корректируем индекс
                }
           }

           // вызываем select с бесконечным таймаутом
           select(max,&read_fs,&write_fs,NULL,NULL);

           for(i=0; i!=Sockets.size() ; i++)
           {
                if(!Sockets[i]->PerformOperation(&read_fs,
                                                 &write_fs))
                {
                     // уничтожаем текущий объект
                     delete Sockets[i];
                     Sockets.erase(Sockets.begin()+i);
                     i--; // корректируем индекс
                }
           }
     }

     return 0;
}
```

2.5 Работа с протоколом UDP

В отличие от протокола TCP, который подразумевает установку связи между двумя системами, протокол UDP обеспечивает лишь отправку отдельных сообщений (датаграмм). Для работы с протоколом UDP при создании сокета нужно указать тип `SOCK_DGRAM` и код протокола `IPPROTO_UDP`. Поскольку соединения в UDP не предусмотрены, то соответственно функции `connect`, `listen` и `accept` для работы с UDP сокетами не используются. Достаточно использовать функцию `bind` чтобы определить локальный адрес и порт, который будет использоваться для отправки или приема датаграмм.

Связав сокет с портом, можно начинать передавать и принимать данные с помощью функций `sendto` и `recvfrom`:

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

ssize_t sendto(int s, const void *msg, size_t len, int flags,
                   const struct sockaddr *to, socklen_t tolen);

ssize_t recvfrom(int s, void * buf, size_t len, int flags,
                 struct sockaddr * from, socklen_t * fromlen);
```

Наряду с параметрами, знакомыми по функциям `send` и `recv`, тут можно видеть указатель на структуру `sockaddr` (вместо которой подставляется `sockaddr_in`) и её размер. В этой структуре указывается адрес получателя при отправке, а при приеме

система записывает по этому указателю адрес системы, от которой была принята датаграмма. В остальном эти функции действуют так же, как и функции `send` и `recv`.

При передаче данных пакетами, необходимо следить, чтобы размер передаваемых данных не превысил возможностей стека протоколов. Максимальный размер датаграммы можно узнать с помощью функции `sysctl` с параметром `UDPCTL_MAXDGRAM`[22]. В качестве размера данных можно передать ноль, в этом случае на адрес получателя отправляется пакет с пустым полем данных.

Работая с протоколом UDP, нужно учитывать, что он не гарантирует получение данных удаленной системой. Также возможно нарушение порядка пакетов при приеме или их дублирование. В случае, если такие явления нежелательны, необходима дополнительная обработка этих ситуаций в программе. С другой стороны, протокол UDP позволяет организовать передачу данных с минимально возможными задержками, что может быть полезным для приложений, для которых необходимо иметь как можно более свежую информацию, возможно даже за счет потери задержавшихся в пути старых данных, как, например, в компьютерных играх.

2.5.1 Пример передачи данных с помощью UDP

В качестве примера работы с UDP протоколом, рассмотрим реализацию сервиса эхо – программы, принимающей UDP датаграммы и отсылающей их назад отправителю.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <string.h>

#define MAX_LEN 1024  // максимальный размер
                      // принимаемого пакета

void main(int argc,char *argv[])
{
   int s;
   char buff[MAX_LEN+1];
   int len,addr_len;

   // создаём сокет для приёма соединений
   // обратите внимание на 2-ой и 3-ий параметры
   s=socket(AF_INET,SOCK_DGRAM,IPPROTO_UDP);
   if(s==INVALID_SOCKET)
   {
        printf("socket() error\n");
        return 1;
   }

   // заполняем структуру локальным адресом
   memset(&addr,0,sizeof(addr));
   addr.sin_family=AF_INET;
   addr.sin_port=htons(1234);    // порт для приёма соединений
   addr.sin_addr.S_un.S_addr=INADDR_ANY;
```

[22] В Windows с помощью `getsockopt` с параметром SO_MAX_MSG_SIZE.

```
                    // используются все локальные интерфейсы

    // связываем сокет с локальным портом
    if(bind(s,(sockaddr*)&addr,sizeof(addr))!=0)
    {
         printf("bind() failed\n");
         closesocket(s);
         return 2;
    }

    // цикл приёма пакетов
    for(;;)
    {
       addr_len=sizeof(addr);
       // принимаем пакет
       len=recvfrom(s,buff,MAX_LEN,0,
                                  (sockaddr*)&addr, &addr_len);

       if(len>=0)
       {
         // распечатываем принятые данные
         buff[len]='\0';
         printf("Datagramm from %s:%d\n%s\n",
                   inet_ntoa(addr.sin_addr),
                   ntohs(addr.sin_port),buff);
         // посылаем пакет обратно
         sendto(s,buff,len,0, (sockaddr*)&addr,addr_len);
       }
       else
       {
         printf("recvfrom() error\n");
         break;
       }
    }

    return 0;
}
```

Использованная в примере функция `inet_ntoa`[23] преобразует IP-адрес в текстовую строку в стандартной нотации «a.b.c.d». Обратите внимание на указанные в справке ограничения по времени использования возвращаемого этой функцией значения.

2.5.2 Задания для самостоятельного выполнения

- Реализуйте программу для общения с эхо-сервером: она должна запросить у пользователя адрес сервера и текст, отправить текст на указанный адрес и распечатать ответ.
- Проверьте работоспособность сервера и вашего клиента.

[23] Аналогичная функция, поддерживающая IPv6, называется `inet_ntop`.

- Отключите в сервере отправку ответа и попробуйте послать «ответ» самостоятельно с помощью клиента с другого компьютера. Какая информация вам для этого потребуется?
- Реализуйте ожидание ответа от сервера в течение ограниченного времени.

2.6 Использование базы данных DNS

До сих пор наши программы могли работать только с IP адресами в цифровом виде. Однако прикладные программы обычно получают от пользователя сетевые адреса в текстовом виде. К счастью, нет необходимости изучать и самостоятельно реализовывать работу с протоколом DNS. Для выполнения преобразования можно воспользоваться функциями, которые имеются в стандартной библиотеке.

Наиболее простой является функция `gethostbyname`.

```
#include <netdb.h>

int h_errno;

struct hostent * gethostbyname(const char *name);
```

В качестве единственного параметра передаётся имя компьютера, информацию о котором требуется получить. Результат возвращается в виде указателя на структуру `hostent`, которая определена следующим образом.

```
#include <netdb.h>
struct hostent {
    char     *h_name;
    char     **h_aliases;
    int      h_addrtype;
    int      h_length;
    char     **h_addr_list;
  };

#define h_addr  h_addr_list[0]
```

Элементы структуры имеют следующее значение:

- `h_name`
  Официальное доменное имя хоста.
- `h_aliases`
  Список альтернативных имён, заканчивается элементом `NULL`.
- `h_addrtype`
  Тип возвращаемого адреса, обычно `AF_INET`.
- `h_length`
  Длина возвращенного адреса в байтах.
- `h_addr_list`
  Список адресов хоста, в двоичном виде в сетевом порядке байтов. Заканчивается элементом `NULL`.
- `h_addr`
  Первый элемент списка адресов, для совместимости со старым кодом.

В случае ошибки `gethostbyname` возвращает значение `NULL` и помещает код ошибки в глобальную переменную `h_errno`. Предусмотрены четыре возможных варианта:

- `HOST_NOT_FOUND`
  Указанное имя не существует.

- TRY_AGAIN
  Временная ошибка, можно попробовать повторить запрос позже. Возможной причиной может являться отсутствие ответа авторизованного DNS сервера.
- NO_RECOVERY
  Произошла ошибка, повторный запрос не поможет.
- NO_DATA
  Для указанного адреса отсутствует запись об IP адресе. Такое происходит, если в базе данных первичного DNS сервера для данного хоста не указана запись с IP адресом.

Возвращаемый этой функцией адрес является указателем на внутреннюю переменную библиотеки. Благодаря использованию **локальной памяти потока (TLS)** использование функции gethosbyname в многопоточном приложении безопасно. Однако необходимо скопировать возвращенные значения, так как следующий вызов этой функции изменит сохранённую в структуре информацию. Не следует каким-либо образом модифицировать записи в структуре или пытаться освободить занимаемую ей память.

Наряду с наличием символьных имён для хостов, имеются и база данных для текстовых названий служб и соответствующих им номеров портов. На UNIX-системах эта информация о соответствии хранится в файле /etc/services[24]. Преобразовать имя службы в номер порта можно с помощью функции getservbyname.

```
#include <netdb.h>

struct servent *
     getservbyname(const char *name, const char *proto);

struct  servent {
     char    *s_name;      /* официальное имя */
     char    **s_aliases;  /* список синонимов */
     int       s_port;      /* номер порта */
     char    *s_proto;     /* имя протокола */
     };
```

Работа с этой функцией сходна с использованием функции gethostbyname. В параметре name задаётся имя сервиса, которое необходимо преобразовать в порт, а в параметре proto – название используемого протокола. Если в качестве proto передать NULL, то будут возвращены записи для всех протоколов.

Существуют и функции, выполняющие обратное преобразование: функция gethostbyaddr позволяет получить имя хоста по его IP адресу, а getservbyport – узнать имя сервиса по номеру порта.

```
#include <netdb.h>

struct servent *
     getservbyport(int port, const char *proto);

struct hostent *
     gethostbyaddr(const void *addr, socklen_t len, int type);
```

Более развитым вариантом функции, выполняющей преобразование имени хоста и сервиса в адрес и порт, является функция getaddrinfo.

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

[24] В Windows это файл system32\drivers\etc\services.

```
#include <netdb.h>

int getaddrinfo(const char *hostname, const char *servname,
        const struct addrinfo *hints, struct addrinfo **res);

struct addrinfo {
     int ai_flags;
     int ai_family;
     int ai_socktype;
     int ai_protocol;
     socklen_t ai_addrlen;
     struct sockaddr *ai_addr;
     char *ai_canonname;
     struct addrinfo *ai_next;
     };
```

Функция `getaddrinfo` возвращает информацию о хосте, имя которого задано параметром `hostname`. В качестве имени может быть указано имя хоста, текстовая строка, в которой указан разделенный точками цифровой IPv4 адрес или IPv6 адрес. Параметр `servname` задаёт порт: может быть указано имя порта из /etc/services или номер в десятичной системе. Вместо любого из этих параметров можно передать `NULL`.

Полученная функцией информация возвращается в виде связанного списка структур `addrinfo`, указатель на первую из которых помещается в переменную `res`. Указателем на следующую структуру является поле `ai_next`. Значения членов структуры `ai_family`, `ai_socktype` и `ai_protocol` возвращаются в виде, пригодном для использования в вызове функции `socket`. Элемент `ai_addr` указывает на заполненную структуру адреса размером `ai_addrlen`.

Аналогичного вида структура, указатель на которую может быть передан параметром `hints`, используется для передачи системе информации о виде необходимой информации. Поле `ai_family` задаёт семейство протоколов, адреса которого требуется получить. Значение `PF_UNSPEC` говорит о готовности работать с адресами любого типа. Аналогично, значения элементов `ai_socktype` и `ai_protocol` задают тип сокета и протокол. Нулевое значение говорит о готовности получить информацию о любом типе сокета/протоколе. В параметре `ai_flag` могут быть заданы различные флаги, которые позволяют задать режимы работы с IPv4 и IPv6 адресами, отключить преобразование имён хостов/служб или сообщить функции, что возвращённая информация будет использована в пассивном соединении. Остальные элементы этой структуры должны быть равны нулю.

Если в качестве hints передаётся `NULL`, то функция `getaddrinfo` работает так, как если бы ей была передана структура hints, элемент `ai_family` которой равен `PF_UNSPEC`, а остальные – нулю.

Все возвращаемые функцией `getaddrinfo` структуры данных выделяются динамически, поэтому нет необходимости копировать их содержание перед следующим вызовом. Однако эту память необходимо освободить после использования, для чего необходимо воспользоваться функцией `freeaddrinfo`.

```
#include <netdb.h>

void freeaddrinfo(struct addrinfo *ai);
```

2.6.1 Задание для самостоятельного выполнения

- Напишите программу, которая бы выводила всю доступную через DNS информацию о хосте, заданном IP адресом или именем.

2.7 Особенности работы с TCP/IP в Windows

Работа с протоколом TCP/IP в Windows осуществляется с помощью библиотеки Windows Sockets, или Winsock. В этом разделе мы рассмотрим особенности работы со второй версией этой библиотеки, используемой в 32-битных ОС семейства Windows.

Интерфейс библиотеки Winsock реализует все возможности работы с сокетами библиотеки Berkley, что упрощает написание переносимых приложений. Кроме того, в ней имеется ряд специфических функций, которые могут быть удобны для работы в операционной системе Windows, однако не имеют аналогов в UNIX. Все специфичные для Windows функции библиотеки Winsock начинаются на буквы WSA, что позволяет легко отличить их от стандартных функций библиотеки Berkley Sockets.

Имеется и ряд отличий, которые все же потребуют некоторых изменений при переносе программы между Windows и UNIX:

1. в отличие от UNIX нужно подключить только один заголовочный файл winsock2.h и библиотеку ws2_32.lib;
2. перед началом работы необходимо вызвать функцию `WSAStartup` для инициализации библиотеки, а по окончании работы – функцию `WSACleanup` для освобождения ресурсов;
3. могут встретиться отдельные случаи использования специфичных для Windows типов данных и констант, определенных в заголовочных файлах Windows;
4. не совпадают параметры сокетов, устанавливаемые вызовами `setsockopt/getsockopt`, `fcntl`, `ioctl`;
5. вместо функции `close` для закрытия сокета используется функция `closesocket`.
6. используются различные способы получения информации об ошибках.

Во многом отличия между интерфейсами Berkley Sockets и Winsock обусловлены тем, что тогда как в UNIX системах поддержка сети исходно внедрялась как постоянная функция ОС, в Windows она появилась и первое время существовала как независимая система, часто реализованная третьими фирмами. Оставила свой отпечаток и разная архитектура операционных систем.

В целом, эти отличия не принципиальны и, при желании и соблюдении некоторой осторожности, можно написать переносимую между Windows и UNIX программу.

2.7.1 Инициализация и освобождение библиотеки

Итак, перед началом работы необходимо инициализировать библиотеку Winsock, вызвав функцию `WSAStartup`:

```
#include <winsock2.h>
int WSAStartup( WORD wVersionRequested, LPWSADATA lpWSAData);
```

Первый параметр этой функции, `wVersionReauested`, позволяет приложению запросить версию библиотеки, с которой оно хочет работать. Основная версия задается в младшем байте, в старшем байте задается номер ревизии. Текущая основная версия библиотеки – 2. Номера ревизии, которые необходимы для поддержки той или иной функции можно найти в документации.

Вторым параметром передается указатель на структуру типа `WSADATA`, которая заполняется в процессе вызова библиотекой Winsock и содержит информацию о текущей версии и возможностях библиотеки.

В случае удачного вызова функция `WSAStartup` возвращает значение `0`, в случае ошибки – один из нескольких кодов ошибки, с которыми можно ознакомиться в документации.

По окончании работы с библиотекой Winsock нужно вызвать функцию `WSACleanup`.

```
#include <winsock2.h>
int  WSACleanup (void);
```

2.7.2 Соответствие типов

Как в Windows, так и в UNIX используются специфические для этих систем нестандартные типы данных. В следующей таблице делается попытка сопоставить используемые с сетевыми функциями типы данных, имеющие разные названия в этих системах. Название типов для UNIX-систем дано по стандарту POSIX.

Тип POSIX	Тип Windows	Комментарий
	Дескрипторы	
Int	SOCKET	В Windows введён специальный тип для дескриптора сокета.
	Адреса	
in_port_t	unsigned short	Порт UDP/TCP, целое без знака 16 бит.
in_addr_t	unsigned long	IP адрес. Целое без знака 32 бита.
	Целочисленные переменные с заданным числом бит.	
uint8_t	u_char, BYTE	8 бит целое без знака (unsigned char).
uint16_t	u_short, WORD	16 бит целое без знака (unsigned short).
uint32_t	u_long, DWORD	32 бита целое без знака (unsigned long).
	Размеры	
ssize_t	int	Размер буферов для чтения/записи.
socklen_t	int	Размер адреса.
sa_family_t	unsigned char или unsigned short	Тип для номера стека протоколов. 8 бит в системах, поддерживающих поле sin_len в структуре sockaddr_in, 16 бит - в не поддерживающих.
	FAR	Объявляет указатель как дальний, в современных ОС не используется и определён в Win32 как пустая строка.

Надо заметить, что различия могут иметь место и между разными UNIX-системами. Например, в Linux вместо типа данных socklen_t (POSIX, BSD) используется тип sockaddr_len и т.п.

В Windows определены константы INVALID_SOCKET и SOCKET_ERROR, которые используются как возвращаемые в случае ошибок значения функциями socket и другими. В UNIX специальные такие константы не объявляются, об ошибке, как правило[25], сигнализирует значение −1. В Windows SOCKET_ERROR определёно как −1, а INVALID_SOCKET как его беззнаковый аналог: ~0. При портировании кода из Windows в UNIX можно определить обе эти константы как −1.

Обратите внимание, что тип SOCKET в Windows определён как беззнаковое целое (unsigned int), в то время как в операционных системах UNIX функцией socket возвращается целое со знаком (int). Соответственно, следующий код будет корректно работать в UNIX, но не в Windows:

```
....
sock_fd = socket(ai_family,ai_socktype,ai_protocol);
if (sock_fd < 0)
{
.... // обработка ошибки – не работает в Windows
```

В Windows условие sock_fd<0 никогда не будет выполнено, так как беззнаковый SOCKET всегда больше или равен нуля, а константа INVALID_SOCKET в Windows соответствует максимальному беззнаковому целому числу. Код обработки ошибки никогда не получит управление. Такая ошибка может появиться в проектах, код которых перенёсен из UNIX в Windows.

2.7.3 Управление параметрами сокетов

Существенно различаются между Windows и UNIX интерфейсы для управления параметрами сокетов. Те задачи, которые в Berkley Sockets решаются с помощью вызовов ioctl / fcntl, в Winsock возложены на ioctlsocket и WSAIoctl. При этом используются разные наборы команд, а обозначения для одинаковых команд могут не совпадать.

Рассмотрим, например, такую операцию как перевод сокета в неблокирующий режим. В UNIX она осуществляется с помощью вызова fcntl, а в Windows – ioctlsocket.

```
#include <winsock2.h>

int ioctlsocket( SOCKET s, long cmd, u_long FAR *argp );
```

Также как и ioctl в UNIX эта функция может выполнять различные операции, в зависимости от значения параметра cmd. Для управления блокирующим режимом используется значение команды FIONBIO. Параметр argp будет определять, хотим ли мы включить или выключить блокирующий режим. Он должен быть указателем на переменную типа unsigned long, имеющую нулевое значение для блокирующего режима и ненулевое для не блокирующего.

Таким образом, чтобы перевести сокет в неблокирующий режим в Windows нам нужно написать следующий код.

```
#include <windows.h>

SOCKET s;
u_long l;

...
```

[25] Всегда проверяйте по справке возвращаемое в случае ошибки значение в тех случаях, когда вы его точно не помните.

```
l=1;
ioctlsocket(s,FIONBIO,&l);
```

Нужно отметить, что не все опции и режимы работы сокетов предусмотренные Berkley Sockets в принципе могут быть использованы в Windows. Например, не поддерживается асинхронный режим работы сокетов с уведомлениями через механизм сигналов. В Windows похожий режим может быть реализован, например, с применением механизмов Overlapped IO, что, однако потребует использования специфических функций.

Функции `setsockopt / getsockopt` реализованы как в Berkley Sockets, так и в Winsock, однако набор устанавливаемых ими опций также различается. Например, в Windows максимальный размер датаграммы UDP можно узнать, запросив с помощью `setsockopt` параметр `SO_MAX_MSG_SIZE`, а в UNIX для этого нужно воспользоваться системным вызовом, который позволяет узнать параметры ядра.

При написании кросплатформенного кода лучшим решением, вероятно, будет реализовать набор специфичных для каждой системы функций, которые решали какие-то задачи, например, включали или выключали блокирующий режим.

2.7.4 Доступ к кодам ошибки

В UNIX функции библиотеки Sockets, как и другие функции стандартной библиотеки, при возникновении ошибки возвращают специальное значение (как правило `−1`) и записывают код ошибки в глобальную переменную `errno`.

В отличие от Berkley Sockets, Winsock с переменной `errno` не работает, так как не является частью стандартной библиотеки языка C в Windows. Для того чтобы получить код ошибки Winsock нужно вызвать функцию `WSAGetLastError`.

```
#include <winsock2.h>

int  WSAGetLastError (void);
```

2.7.5 Другие функции библиотеки Winsock

Операционная система UNIX создавалась под лозунгом «всё является файлом». Сетевая подсистема в UNIX полностью соответствует этой концепции и дескриптор сокета равноправен с дескриптором файла. Таким образом, при разработке приложения в UNIX мы можем использовать, например функцию `select` для того, чтобы одновременно проверять готовность к вводу-выводу сетевого соединения и наличие ввода пользователя с клавиатуры. Однако подсистема ввода-вывода в Windows построена в соответствии с другими концепциями, и при переносе на эту платформу было невозможно сохранить такую же высокую степень интеграции сетевой подсистемы и подсистемы ввода-вывода. Так функция `select` сохранила свою функциональность применительно к сетевым сокетам, но не может использоваться с обычными файлами или другими объектами.

Наряду с функциями для инициализации и закрытия библиотеки, в библиотеке Winsock есть ряд специфичных для Windows функций. Использование этих функций сильно затруднит портирование кода на UNIX-системы, однако для Windows приложений они обеспечивают доступ к сетевым средствам с использованием базовых возможностей операционной системы Windows, таким как механизм сообщений, средства синхронизации и асинхронного ввода-вывода.

Так, например, функция `WSAAsyncSelect` позволяет приложению получать уведомления о событиях, связанных с сокетом, посредством механизма сообщений.

Функции WSASend, WSARecv, WSASendTo, WSARecvFrom поддерживают наряду с обычными режимами режим Overlapped IO и работу с несколькими буферами приема/передачи данных.

2.7.6 Пример программы, использующей Windows Sockets

В качестве примера рассмотрим запрос корневой директории с web-сервера. Эта программа полностью соответствует рассматриваемому ранее примеру, за исключением необходимых для работы с winsock изменений: использования типа SOCKET, вызова функций инициализации и закрытия библиотеки и использования closesocket вместо socket.

```
#include <winsock2.h>
#include <string.h>
#include <stdio.h>

int main(int argc,char * argv[])
{
   WSADATA wd;
   SOCKET s;      // используем тип SOCKET
   sockaddr_in addr;
   int i;
   char str[101];

   if(argc<2)
   {
        printf("enter IP address of server as parameter");
        return  100;
   }

   // инициализируем библиотеку windows sockets
   if(WSAStartup(MAKEWORD(2,2),&wd)!=0)
   {
        printf("Winsock initialization failed.");
        return 200;
   }

   // создаём сокет
   s=socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);
   if(s==INVALID_SOCKET)
   {
        printf("socket() error\n");
        return 1;
   }

   // заполняем структуру с адресом сервера
   memset(&addr,0,sizeof(addr));
   addr.sin_family=AF_INET;
   addr.sin_port=htons(80); // для web-серверов обычно
                            // используется порт 80
   addr.sin_addr.S_un.S_addr=inet_addr(argv[1]);
                            // IP адрес сервера задаётся
                            // в командной строке
```

```
    // устанавливаем соединение
    if(connect(s,(const sockaddr*)&addr,sizeof(addr))!=0)
    {
         printf("connect() error\n");
         closesocket(s);
         return 2;
    }

    // посылаем запрос согласно протокола HTTP
    send(s,"GET /\n\n",7,0);
    shutdown(s,SD_SEND);     // закрываем соединение на передачу

    do
    {
         i=recv(s,str,100,0); // получаем данные
         if(i>0)
         {
              str[i]='\0';
              printf("%s",str);
         }
    } while(i>0);  // при завершении передачи данных сервером
                   // функция recv вернёт 0

    shutdown(s,SD_BOTH);// окончательно закрываем соединение
    closesocket(s);     // закрываем сокет используя closesocket
    WSACleanup();       // завершаем работу с Windows sockets
    return 0;
}
```

2.7.7 Задание для самостоятельного выполнения

- Модифицируйте код одной из своих предыдущих программ так, чтобы он мог компилироваться и выполняться как под UNIX, так и под Windows.

3 Программы для работы с TCP/IP

Наряду со стандартной библиотекой sockets на большинстве систем, поддерживающих работу со стеком TCP/IP, имеется набор программ, которые могут быть полезны для настройки сети, проверки её работоспособности и отладки приложений, использующих TCP/IP.

3.1 ifconfig

В UNIX-системах программа `ifconfig` используется для конфигурирования ethernet-интерфейсов. Она позволяет задать ip-адрес, маску и другие параметры сетевого интерфейса. Как правило, эта программа запускается одним из скриптов инициализации во время загрузки системы.

Примеры

Присвоить IPv4 адрес 192.168.2.10 с маской 255.255.255.0 интерфейсу ed0:

```
# ifconfig ed0 inet 192.168.2.10 netmask 255.255.255.0
```

3.2 ipconfig.exe

В Windows настройка сетевых интерфейсов осуществляется с помощью одного из приложений панели управления. Посмотреть настройки сетевых интерфейсов можно с помощью программы `ipconfig.exe`, работающей в текстовом режиме. Ключ `/all` включает вывод более подробной информации.

Примеры

Краткая информация ifconfig о настройках IP в Windows 2000. В системе имеются неиспользуемый Ethernet адаптер и соединение с интернет через модем.

```
C:\>ipconfig

Настройка протокола IP для Windows 2000

Адаптер Ethernet Подключение по локальной сети 3:

   Состояние устройства  . . . . . : отсоединен кабель

Адаптер ТвГУ:

   DNS суффикс этого подключения . . :
   IP-адрес  . . . . . . . . . . . . : 82.179.134.52
   Маска подсети . . . . . . . . . . : 255.255.255.255
   Основной шлюз . . . . . . . . . . : 82.179.134.52
```

Подробная информация о той же системе.

```
C:\>ipconfig /all

Настройка протокола IP для Windows 2000

   Имя компьютера  . . . . . . . . . : PC_1
   Основной DNS суффикс  . . . . . . :
   Тип узла  . . . . . . . . . . . . : Широковещательный
   Включена IP-маршрутизация . . . . : Нет
```

```
   Доверенный WINS-сервер  . . . . . : Нет

Адаптер Ethernet Подключение по локальной сети 3:

   Состояние устройства  . . . . . : отсоединен кабель
   Описание  . . . . . . . . . . . . : Attansic L1 Gigabit
                                 Ethernet 10/100/1000 Base-T Adapter
   Физический адрес. . . . . . . . . : 00-18-F3-6F-52-B2

Адаптер ТвГУ:

   DNS суффикс этого подключения . . :
   Описание  . . . . . . . . . . . . : WAN (PPP/SLIP) Interface
   Физический адрес. . . . . . . . . : 00-53-45-00-00-00
   DHCP разрешен . . . . . . . . . . : Нет
   IP-адрес  . . . . . . . . . . . . : 82.179.134.52
   Маска подсети . . . . . . . . . . : 255.255.255.255
   Основной шлюз . . . . . . . . . . : 82.179.134.52
   DNS-серверы . . . . . . . . . . . : 82.179.130.1
                                       62.76.80.1
   NetBIOS через TCP/IP. . . . . . . : отключено
```

3.3 ping

Программу `ping` используют для проверки работоспособности IP сети. Она использует протокол ICMP, отправляя эхо запросы на указанный адрес, и распечатывает время получения ответа. После отправки нескольких запросов печатается статистика о числе полученных/пропущенных ответов и времени задержки.

Если вы не можете связаться с удалённой системой, попробуйте запустить программу ping для проверки связи. Начните с запуска программы ping, указав в параметрах loopback-адрес. Если он не работает, то это говорит о проблемах с настройкой сети на локальной машине. Затем попробуйте получить ответ от других компьютеров, подключенных к той же локальной сети. Если связь в пределах локальной сети работает, проверьте получение ответов от более удалённых компьютеров. Целесообразно начинать с ближайшего маршрутизатора.

Если программа ping сообщает о невозможности определить имя удалённой системы, попробуйте вводить напрямую IP адреса. Если это помогает – проверьте настройки DNS сервера и наличие связи до него с помощью программы ping.

Большое количество потерянных пакетов говорит о плохом качестве работы сети – скорее всего, нужно проверить работоспособность оборудования и нагрузку на сеть.

Примеры

Отправка эхо-запросов на loopback-адрес, который должен всегда работать:

```
C:\> ping 127.0.0.1

Обмен пакетами с 127.0.0.1 по 32 байт:

Ответ от 127.0.0.1: число байт=32 время<10мс TTL=128
Ответ от 127.0.0.1: число байт=32 время<10мс TTL=128
```

```
Ответ от 127.0.0.1: число байт=32 время<10мс TTL=128
Ответ от 127.0.0.1: число байт=32 время<10мс TTL=128

Статистика Ping для 127.0.0.1:
    Пакетов: отправлено = 4, получено = 4, потеряно = 0 (0%
потерь),
Приблизительное время передачи и приема:
    наименьшее = 0мс, наибольшее =  0мс, среднее =  0мс
```

Работа программы ping в условиях потери пакетов:

```
C:\>ping www.google.ru

Обмен пакетами с www.l.google.com [74.125.79.106] по 32 байт:

Ответ от 74.125.79.106: число байт=32 время=266мс TTL=51
Ответ от 74.125.79.106: число байт=32 время=235мс TTL=51
Превышен интервал ожидания для запроса.
Ответ от 74.125.79.106: число байт=32 время=266мс TTL=51

Статистика Ping для 74.125.79.106:
    Пакетов: отправлено = 4, получено = 3, потеряно = 1 (25%
потерь),
Приблизительное время передачи и приема:
    наименьшее = 235мс, наибольшее =  266мс, среднее =  191мс
```

В качестве дополнительного задания определите, в чём состоит ошибка программистов Microsoft, допущенная при определении среднего времени.

3.4 traceroute

Эта программа позволяет не только узнать есть ли связь до удалённой системы, но и посмотреть, через какие хосты проходят ваши пакеты.

Программа `traceroute` отправляет IP пакеты с увеличивающимися значениями поля `TTL`. По мере продвижения пакета по сети это значение уменьшается и, при достижении 0, пакет уничтожается, а в ответ посылается ICMP уведомление. Получая эти уведомления, `traceroute` и узнает маршрут.

С помощью `traceroute` мы можем не только узнать маршрут наших пакетов, но и понять где возникают задержки или потери пакетов.

В Windows эта программа называется `tracert`.

Примеры

Использование программы `traceroute` для локализации проблем со связью. В рассматриваемом случае электронная почта работала нормально, а web-страницы открывались очень медленно или не открывались вообще.

```
C:\>tracert www.google.ru

Трассировка маршрута к www.l.google.com [74.125.79.103]
с максимальным числом прыжков 30:

1  203 ms 203 ms 188 ms  82.179.128.62
2  203 ms 188 ms 187 ms  82.179.128.61
3    *    282 ms   *     m9-1-gw.msk.runnet.ru [194.190.255.101]
4    *      *    204 ms  msk-ix-gw1.google.com [193.232.244.232]
```

```
5  281 ms    *       *     72.14.236.248
6    *       *       *     Превышен интервал ожидания для запроса.
7    *       *       *     Превышен интервал ожидания для запроса.
8  282 ms    *       *     209.85.255.166
9    *       *       *     Превышен интервал ожидания для запроса.
10   *     297 ms    *     74.125.79.103
11 266 ms    *       *     74.125.79.103
12 250 ms    *     235 ms  ey-in-f106.google.com [74.125.79.106]

Трассировка завершена.
```

Программа `traceroute` отправляет по три пакета с одинаковым значением TTL и выводит время возвращения ICMP уведомления об уничтожении пакета, или звездочку, если уведомление не было получено.

Как мы можем видеть, в данном случае потери пакетов начинаются между вторым и третьим маршрутизаторами, так что мы можем сделать вывод, что проблемы возникли где-то на этом участке.

Как правило, `traceroute` выводит и имя транзитного хоста, и его адрес. В нашем примере для многих хостов выводится только адрес. Это может произойти, если для этих адресов не прописаны соответствующие имена в зоне `IN-ADDR.ARPA` (что, видимо, имеет место с первым и вторым хостами), или из-за потерь пакетов DNS протокола (что, вероятно, происходило в этом примере в большинстве случаев).

3.5 arp

Программа `arp` используется для просмотра и модификации таблицы преобразования ip-адресов в MAC-адреса. Ключ `-a` позволяет просмотреть текущую arp-таблицу, с помощью ключа `-s` производится добавление статических записей, а ключ `-d` позволяет удалить записи из таблицы.

Примеры.

Добавить в arp-таблицу, статическую запись, устанавливающую для ip-адреса 192.168.1.212 соответствие с MAC-адресом 00AA0062C609:

```
# arp -s 192.168.1.212   00-aa-00-62-c6-09
```

Вывести arp-таблицу:

```
# arp -a
Интерфейс: 192.168.1.138 on Interface 0x1000003
  Адрес IP             Физический адрес      Тип
  192.168.1.212        00-aa-00-62-c6-09     статический
  192.168.1.1          00-18-39-27-76-44     динамический
  192.168.1.5          00-18-F3-6F-52-B2     динамический
  192.168.1.137        00-18-EA-64-8B-71     динамический
```

Удалить запись для ip 192.168.1.212:

```
# arp -d 192.168.1.212
```

3.6 route

Используется для управления и отображения таблицей маршрутизации. Эта программа поддерживает следующие команды:

- `PRINT` – распечатать таблицу маршрутизации
- `ADD` – добавить запись

- CHANGE – изменить запись
- DELETE – удалить запись

Примеры

Распечатать таблицу маршрутизации:

```
> route PRINT
```

Распечатать только узлы, начинающиеся со 157:

```
> route PRINT 157*
```

Добавить запись “пакеты для сети 157.0.0.0 с маской 255.0.0.0 отправлять через интерфейс 2 на маршрутизатор 157.55.80.1, метрика 3” в таблицу маршрутизации:

```
> route ADD 157.0.0.0 MASK 255.0.0.0 157.55.80.1 METRIC 3 IF 2
```

Если интерфейс (IF) не задан, то будет использован наиболее подходящий интерфейс для указанного шлюза.

Удалить запись из таблицы маршрутизации:

```
> route DELETE 157.0.0.0
```

3.7 netstat

Позволяет просмотреть статистику протоколов и текущие сетевые соединения. По умолчанию, netstat отображает список сокетов, находящиеся в состоянии открытого соединения. Ключ –a позволяет увидеть также и сокеты, находящиеся в состоянии пассивного соединения (ожидающие входящее соединение).

Ключ –e позволяет просмотреть статистику Ethernet контроллеров. Ключ –s – статистику по протоколам.

Примеры

Вывод открытых соединений на системе с ОС Windows 2000 (программа выполнялась на компьютере с именем cf-34):

```
> netstat

Активные подключения

 Имя   Локальный адрес       Внешний адрес          Состояние
 TCP   cf-34:1072            cf-34:1073             ESTABLISHED
 TCP   cf-34:1073            cf-34:1072             ESTABLISHED
```

Вывод всех открытых сокетов на той же системе:

```
> netstat -a

Активные подключения

 Имя   Локальный адрес       Внешний адрес          Состояние
 TCP   cf-34:epmap           cf-34:0                LISTENING
 TCP   cf-34:microsoft-ds    cf-34:0                LISTENING
 TCP   cf-34:1025            cf-34:0                LISTENING
 TCP   cf-34:1027            cf-34:0                LISTENING
 TCP   cf-34:1073            cf-34:0                LISTENING
 TCP   cf-34:1072            cf-34:0                LISTENING
 TCP   cf-34:1072            cf-34:1073             ESTABLISHED
```

```
  TCP    cf-34:1073             cf-34:1072             ESTABLISHED
  TCP    cf-34:8118             cf-34:0                LISTENING
  UDP    cf-34:epmap            *:*
  UDP    cf-34:microsoft-ds     *:*
  UDP    cf-34:1026             *:*
  UDP    cf-34:1028             *:*
  UDP    cf-34:1060             *:*
```

3.8 Netcat

Эта программа не входит в «классический» набор tcp/ip утилит, однако она является очень удобным инструментом для отладки сетевых приложений.

Утилита netcat позволяет принимать и отправлять данные по сетевым соединениям, используя протоколы TCP и UDP. Она может создавать практически любой вид соединения, который может понадобиться для исследования сети или разработки сетевых приложений, и имеет богатый набор встроенных возможностей.

Программа Netcat может работать в трёх основных режимах. В режиме активного соединения можно установить соединение с удалённым сервером и, например, предать на него данные от какой-нибудь локальной программы. Режим приёма соединения может быть использован для получения потока данных от удаленного клиента. Режим туннелирования позволяет перенаправить входящее соединение на другую сетевую систему.

Утилита Netcat существует в нескольких вариантах. Последняя версия оригинальной программы (1.10) была разработана в 1996 году и в настоящее время доступна по адресу http://nc110.sourceforge.net/. Среди альтернативных реализаций можно отметить GNU Netcat, официальный сайт которой находится по адресу http://netcat.sourceforge.net/.

Примеры

Режим активного соединения.

Для установки соединения достаточно указать имя хоста и порт. Например, соединяемся с web-сервером www.tversu.ru:

```
# netcat www.tversu.ru 80
```

После установки соединения можно вводить HTTP запрос и смотреть на ответ сервера.

Режим приёма соединения.

Для включения режима приёма соединений нужно указать ключ –l. С помощью ключа –p указывается локальный порт, на котором будет приниматься соединение. Для приёма соединения на порт 9875 необходимо ввести следующую команду:

```
# netcat -l -p 9875
```

После подключения клиента можно общаться с ним через стандартный ввод и вывод.

Режим туннелирования.

Ключ –L, доступный в версии GNU Netcat, включает режим туннелирования (перенаправления) входящего соединения. Перенаправляем входящие соединения на порт 8080 на web-сервер www.tversu.ru.

```
# netcat –L www.tversu.ru:80 -p 8080
```

В других версиях Netcat аналогчиного результата можно достигнуть, соединив две копии программы с помощью конвеера.

```
# netcat -l –p 8080 | netcat www.tversu.ru 80
```

Работа с протоколом UDP.

Ключ –u включает режим работы через UDP сокеты. Чтобы получить текущее время с сервера времени RFC867 [29] наберите следующую команду и нажмите ввод. Для выхода нажмите Ctrl+C.

```
# netcat -u time.server.com 13

Thu Oct 20 14:41:57 2009
```

Использование Netcat для решения практических задач.

`Netcat` можно использовать для копирования файлов по сети. Для этого сначала на передающем компьютере выполните:

```
# netcat -l -p 1234 < testfile
```

Затем на принимающем:

```
# netcat sender.ip.org 1234 > testfile
```

Если у вас есть версия `netcat` под Windows можно организовать доступ к командной строке по сети. Для этого воспользуемся режимом приёма входящих соединений и ключом –e, предписывающим запустить внешнюю программу для обработки соединения.

```
c:\> nc -l -p1000 -d -e cmd.exe -L
```

При поступлении соединения на порт 1000 `netcat` запустит командный интерпретатор cmd.exe, стандартные ввод и вывод которого будут перенаправлены в сетевое соединение, и можно будет удалённо работать на компьютере. Обратите внимание, что пароль при этом не спрашивают. Опция –d доступна только в Windows версии и позволяет запустить `netcat` в фоновом режиме, чтобы он привлекал меньше внимания.

Сканирование (поиск открытых) портов на компьютере 192.168.1.2.

```
# netcat -v -n -z -w 1 192.168.1.2 1-1000
```

Опция `-v` включает отображение сообщений (в том числе об установленных соединениях), `-n` отключает DNS преобразования, `-z` заставляет закрыть соединение сразу же после его открытия, `-w` устанавливает таймаут 1 секунда, 1-1000 задаёт диапазон портов.

3.9 nslookup

Программа `nslookup` используется для получения информации с DNS серверов и проверки работоспособности DNS системы.

После запуска программы вы можете вводить доменные имена и сможете увидеть выдаваемые сервером ответы. Команда `server` позволяют выбрать используемый DNS сервер. Команда `set` – задать разнообразные опции, например, виды запрашиваемых записей, режим отладки, список доменов для поиска. Встроенная справка доступна по команде `help`.

Примеры

Рассмотрим пример использования программы nslookup. Жирным шрифтом выделен ввод пользователя.

При запуске программы она сообщает об используемом DNS сервере.

```
c:\>nslookup
Default Server:  DD-WRT
Address:  192.168.1.1
```

Чтобы запросить информацию о доменном имени нужно просто набрать его и нажать Enter. Ниже показан запрос информации об имени www.tversu.ru. Сообщение “Non-authoritative answer” говорит о том, что DNS сервер сообщает нам информацию, полученную им от другого сервера.

```
> www.tversu.ru
Server:  DD-WRT
Address:  192.168.1.1

Non-authoritative answer:
Name:    www.tversu.ru
Address:  82.179.130.12
```

По умолчанию программа nslookup запрашивает информацию об ip-адресе, соответствующем введённому имени. Чтобы запросить другую информацию необходимо выполнить команду set querytype. Например, для запроса информации о почтовом сервере домена, нужно установить тип запроса MX. В следующем примере показано, как определить, куда следует отправлять почту, адресованную на адрес вида …@tversu.ru.

```
> set querytype=MX
> tversu.ru
Server:  DD-WRT
Address:  192.168.1.1

Non-authoritative answer:
tversu.ru  MX preference = 10, mail exchanger = mail.tversu.ru

tversu.ru       nameserver = ns.tversu.ru
tversu.ru       nameserver = ns.runnet.ru
mail.tversu.ru  internet address = 82.179.130.2
ns.runnet.ru    internet address = 194.85.32.18
ns.tversu.ru    internet address = 82.179.130.1
```

Для этого адреса указан один почтовый сервер mail.tversu.ru с приоритетом 10. Таких серверов может быть указано много. Попробуем узнать, сколько почтовых серверов обрабатывает почту Google mail.

```
> gmail.com
Server:  DD-WRT
Address:  192.168.1.1

Non-authoritative answer:
gmail.com       MX preference = 20, mail exchanger =
alt2.gmail-smtp-in.l.google.com
gmail.com       MX preference = 30, mail exchanger =
alt3.gmail-smtp-in.l.google.com
gmail.com       MX preference = 40, mail exchanger =
alt4.gmail-smtp-in.l.google.com
gmail.com       MX preference = 5, mail exchanger = gmail-
smtp-in.l.google.com
gmail.com       MX preference = 10, mail exchanger =
alt1.gmail-smtp-in.l.google.com

gmail.com       nameserver = ns1.google.com
gmail.com       nameserver = ns3.google.com
```

```
gmail.com       nameserver = ns4.google.com
gmail.com       nameserver = ns2.google.com
ns4.google.com  internet address = 216.239.38.10
ns1.google.com  internet address = 216.239.32.10
ns2.google.com  internet address = 216.239.34.10
ns3.google.com  internet address = 216.239.36.10
```

Тут можно видеть 5 серверов с разным приоритетом, наиболее предпочтителен сервер gmail-smtp-in.l.google.com.

В ответах также содержится информация о DNS серверах, отвечающих за интересующие нас домены. Программа nslookup позволяет переключится на использование других сервреров с помощью команды `server`. В следующем примере показано, как переключится на использование DNS сервера домена tversu.ru (из предыдущих примеров видно, что он называется ns.tversu.ru и имеет адрес 82.179.130.1) и запросить информацию о хосте www.tversu.ru непосредственно у него.

```
> server 82.179.130.1
Default Server:  ns.tversu.ru
Address:  82.179.130.1

> set querytype=ANY
> www.tversu.ru
Server:  ns.tversu.ru
Address:  82.179.130.1

www.tversu.ru   internet address = 82.179.130.12
tversu.ru       nameserver = ns.tversu.ru
tversu.ru       nameserver = ns.runnet.ru
ns.runnet.ru    internet address = 194.85.32.18
ns.tversu.ru    internet address = 82.179.130.1
```

Отсутствие надписи «Non-authoritative answer» говорит о том, что получен ответ от сервера, который хранит информацию об этом имени.

В следующем примере показано, какую информацию сервер хранит о самом домене tversu.ru.

```
> tversu.ru
Server:  ns.tversu.ru
Address:  82.179.130.1

tversu.ru
        primary name server = ns.tversu.ru
        responsible mail addr = mike.ns.tversu.ru
        serial  = 2009101501
        refresh = 86400 (1 day)
        retry   = 7200 (2 hours)
        expire  = 2419200 (28 days)
        default TTL = 86400 (1 day)
tversu.ru  MX preference = 10, mail exchanger = mail.tversu.ru
tversu.ru       nameserver = ns.runnet.ru
tversu.ru       nameserver = ns.tversu.ru
tversu.ru       text =

        "v=spf1 +mx:tversu.ru -all"
```

```
tversu.ru       internet address = 82.179.130.2
mail.tversu.ru  internet address = 82.179.130.2
ns.runnet.ru    internet address = 194.85.32.18
ns.tversu.ru    internet address = 82.179.130.1
```

Тут присутствует уже знакомая информация о почтовом сервере, а также информация о самом домене: первичным (основным) сервером домена является ns.tversu.ru, резервным – ns.runnet.ru.

Поле `serial` сообщает версию базы данных, это поле должно увеличиваться при каждом изменении информации о домене, для того чтобы резервные сервера знали, что нужно скачать новую версию базы с основного сервера. Часто в качестве значения этого поля принято использовать дату внесения изменения + порядковый номер изменения за день. Скорее всего, информация о домене tversu.ru последний раз менялась 15 октября 2009 года.

Параметры `refresh`, `retry` и `expire` задают временные параметры взаимодействия между основным и резервными серверами. Резервный сервер будет пытаться синхронизовать свою базу данных с базой данных основного сервера через каждые `refresh` секунд. Если ему не удастся установить связь с основным сервером, то он должен повторять свои попытки с интервалом, заданным параметром `retry`. Если ему не удастся синхронизовать базу данных в течение `expire` секунд, то он признает имеющиеся у него данные устаревшими и перестанет отвечать на запросы о данном домене до тех пор, пока ему не удастся синхронизовать базу данных.

`Default TTL` указывает допустимое время кэширования информации другими DNS серверами. Это значение может быть изменено для отдельных записей заданием параметра `TTL`.

И, наконец, поле `responsible mail addr` сообщает нам e-mail ответственного за работу DNS сервера. Первую точку следует заменить символом ‘@’.

3.9.1 Задания для самостоятельного выполнения

- Изучите протокол отправки почты SMTP (RFC2821 [30], осенью 2008 года обновлён в RFC5321 [18]). Используя `nslookup`, узнайте адрес почтового сервера, принимающего вашу почту. Соединитесь с ним, используя `netcat`, и отправьте себе письмо.

4 Задачи для программирования

Задание 1

Разработайте пару приложений для передачи файлов по сети. Одно приложение отправляет указанный пользователем файл, используя активное соединение с сервером. Сервер принимает соединения клиентов и сохраняет переданные им файлы в текущей директории.

Задание 2

Модифицируйте программу из задания 1 таким образом, чтобы клиент мог передать серверу несколько файлов, не прерывая соединения.

Задание 3

Разработайте программу для общения двух пользователей через сеть (Talk). Пользователи должны иметь возможность выбрать режим работы – приём или активное соединение. После установки соединения пользователи могут обмениваться текстовыми сообщениями.

Задание 4

Разработайте систему многопользовательского чата. Система обслуживается сервером, ведущим учёт подключенных клиентов. Протокол обмена между клиентом и сервером должен предусматривать:

- возможность указания имени пользователя при соединении,
- получение текущего списка пользователей,
- прием и передачу текстовых сообщений,
- возможность изменения имени пользователя в процессе работы.

Полученные от одного клиента сообщения передаются всем другим пользователям.

Задание 5

Реализуйте систему распределенного чата, не использующую центральный сервер. Клиенты, использующие систему, могут подключаться друг к другу произвольным образом, при этом должна обеспечиваться доставка сообщений до всех подключенных пользователей. При реализации не должны использоваться механизмы широковещательной передачи.

Задание 6

Добавьте в систему распределённого чата поддержку общения по группам (темам). Сообщения должны доставляться только тем пользователям, которые подключены к соответствующей группе (или через которые должен проходить маршрут доставки сообщений).

Список литературы

[1] В.Г. Олифер, Н.А. Олифер. Компьютерные сети. Принципы, технологии, протоколы. Учебник для ВУЗов. 3-е издание. – СПб.:Питер, 2006. – 958 с: ил. ISBN 5-469-00504-6

[2] Т. Паркер, К. Сиян. TCP/IP для профессионалов. 3-е издание. — СПб.: Питер, 2004. — 859 с: ил. ISBN 5-8046-0041-9

[3] Э. Таненбаум. Компьютерные сети. 4-е издание. – СПб.: Питер, 2003.-992 с.

[4] Behrouz A. Forouzan. Data Communications and Networking. - 4th edition. McGraw-Hill, 2007. ISBN-13 978-0-07-296775-3 ISBN-10 0-07-296775-7

[5] J. Postel. DoD standard Transmission Control Protocol. January 1980. [Электронный ресурс] — Режим доступа: http://www.ietf.org/rfc/rfc761.txt

[6] Y. Rekhter, B. Moskowitz, D. Karrenberg, G.J. de Groot, E. Lear. Address Allocation for Private Internets. Febuary 1996. [Электронный ресурс] — Режим доступа: http://tools.ietf.org/html/rfc1918

[7] M.Cotton, L. Vegoda. Special Use IPv4 Addresses. January 2010. [Электронный ресурс] — Режим доступа: http://www.ietf.org/rfc/rfc5735.txt

[8] J. Postel. Internet Protocol. September 1981. [Электронный ресурс] — Режим доступа: http://www.ietf.org/rfc/rfc791.txt

[9] J. Reynolds, J. Postel. Assigned Numbers. October 1994. [Электронный ресурс] — Режим доступа: http://www.ietf.org/rfc/rfc1700.txt

[10] Protocol Numbers. http://www.iana.org/assignments/protocol-numbers/protocol-numbers.xhtml

[11] R.T. Braden, D.A. Borman, C. Partridge. Computing the Internet checksum. September 1988. [Электронный ресурс] — Режим доступа: http://www.ietf.org/rfc/rfc1071.txt

[12] J. Postel. Internet Control Message Protocol. September 1981. [Электронный ресурс] — Режим доступа: http://www.ietf.org/rfc/rfc792.txt

[13] Port Numbers. http://www.iana.org/assignments/port-numbers

[14] J. Postel. User Datagram Protocol.August 1980. [Электронный ресурс] — Режим доступа: http://www.ietf.org/rfc/rfc768.txt

[15] J. Postel, J. Reynolds. File Transfer Protocol. October 1985. [Электронный ресурс] — Режим доступа: http://www.ietf.org/rfc/rfc959.txt

[16] J. Myers, M. Rose. Post Office Protocol - Version 3. May 1996. [Электронный ресурс] — Режим доступа: http://www.ietf.org/rfc/rfc1939.txt

[17] M. Crispin. INTERNET MESSAGE ACCESS PROTOCOL - VERSION 4rev1. March 2003. [Электронный ресурс] — Режим доступа: http://www.ietf.org/rfc/rfc3501.txt

[18] J. Klensin. Simple Mail Transfer Protocol. October 2008. [Электронный ресурс] — Режим доступа: http://www.ietf.org/rfc/rfc5321.txt

[19] C. Feather. Network News Transfer Protocol (NNTP). October 2006. [Электронный ресурс] — Режим доступа: http://www.ietf.org/rfc/rfc3977.txt

[20] P.V. Mockapetris. Domain names - concepts and facilities. November 1987. [Электронный ресурс] — Режим доступа: http://www.ietf.org/rfc/rfc1034.txt

[21] P.V. Mockapetris. Domain names - implementation and specification. November 1987. [Электронный ресурс] — Режим доступа: http://www.ietf.org/rfc/rfc1035.txt

[22] P. Loshin. IPv6: Theory, Protocol, and Practice. -2nd ed. Morgan, 2004.

[23] S. Bradner, A. Mankin. The Recommendation for the IP Next Generation Protocol. January 1995. [Электронный ресурс] — Режим доступа: http://www.ietf.org/rfc/rfc1752.txt

[24] S. Kent, K. Seo. Security Architecture for the Internet Protocol. December 2005. [Электронный ресурс] — Режим доступа: http://www.ietf.org/rfc/rfc4301.txt

[25] S. Thomson, T. Narten, T. Jinmei. IPv6 Stateless Address Autoconfiguration. September 2007. [Электронный ресурс] — Режим доступа: http://www.ietf.org/rfc/rfc4862.txt
[26] A. Conta, S. Deering. Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6). December 1995. [Электронный ресурс] — Режим доступа: http://www.ietf.org/rfc/rfc1885.txt
[27] A. Conta, S. Deering. Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification. December 1998. [Электронный ресурс] — Режим доступа: http://www.ietf.org/rfc/rfc2463.txt
[28] A. Conta, S. Deering, M. Gupta, Ed.. Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification. March 2006. [Электронный ресурс] — Режим доступа: http://www.ietf.org/rfc/rfc4443.txt
[29] J. Postel. Daytime Protocol. May 1983. [Электронный ресурс] — Режим доступа: http://www.ietf.org/rfc/rfc867.txt
[30] J. Klensin, Ed.. Simple Mail Transfer Protocol. April 2001. [Электронный ресурс] — Режим доступа: http://www.ietf.org/rfc/rfc2821.txt

[Stevens] W. Richard Stevens, Bill Fenner, Andrew M. Rudoff UNIX® Network Programming Volume 1, Third Edition: The Sockets Networking API. Addison Wesley, 2003

Сорокин Сергей Владимирович

Разработка сетевых приложений с использованием стека протоколов TCP/IP

Учебное пособие