Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Кафедра информатики

С.И. Сиротко, А.А. Волосевич

КОМПЬЮТЕРНЫЕ СЕТИ

Учебное пособие
для студентов специальности I-31 03 04 «Информатика»
всех форм обучения

Минск 2006

УДК 004.7(075.8)
ББК 32.973.202 я 73
С 40

Рецензенты:
доцент кафедры математического обеспечения ЭВМ БГУ,
канд. техн. наук Л.Ф. Зимянин;
доцент кафедры микропроцессорных систем и сетей ИИТ БГУИР,
канд. техн. наук В.Н. Мухаметов

Сиротко С.И.

С 40 Компьютерные сети: Учебное пособие для студ. спец. I-31 03 04 «Информатика» всех форм обуч. / С.И. Сиротко, А.А. Волосевич. – Мн.: БГУИР, 2006. – 95 с.: ил.

ISBN 985-444-904-1

Учебное пособие составлено в соответствии с рабочей программой курса «Компьютерные сети». В него включены базовые сведения из теории вычислительных (компьютерных) сетей, основы построения и функционирования сетей Internet, рассмотрены протоколы IP, TCP и UDP, основы программирования сетевых приложений в среде TCP/IP, а также с использованием интерфейсов Mailslots и NetBIOS. Приводятся учебные примеры сетевых приложений.

Пособие может быть рекомендовано студентам и магистрантам технических специальностей для изучения базовых технологий компьютерных сетей.

УДК 004.7(075.8)
ББК 32.973.202 я 73

ISBN 985-444-904-1

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Компьютерные сети являются неотъемлемой частью современных информационных технологий. Значительная часть разрабатываемого и уже эксплуатируемого программного обеспечения, в том числе web-базирующиеся информационные системы, распределенные вычисления и другие перспективные направления так или иначе связаны с работой в сетях.

Целью курса «Компьютерные сети» является изучение теоретических основ построения и функционирования компьютерных (вычислительных) сетей, в первую очередь использующих стек TCP/IP, их базовых технологий, протоколов и служб, а также приобретение знаний и практических навыков разработки сетевых приложений.

Пособие ориентировано в большей степени на практическую часть учебного курса – сетевое программирование. Оно тесно связано с преподаваемыми кафедрой курсами «Системное программирование» и «Операционные системы и среды», а также может служить в качестве основы для дальнейшего изучения информационных сетевых технологий более высокого уровня.

В пособии рассматриваются следующие вопросы.

1. Элементы теории сетей в пределах, необходимых для освоение изложенного далее материала.
2. Стек протоколов TCP/IP, сети IP и протокол IP.
3. Транспортная система сетей и транспортные протоколы TCP и UDP.
4. Программирование с использованием API сокетов в среде TCP/IP, включая приложения-серверы в режиме множественного доступа.
5. Программирование с использованием других интерфейсов (Windows Mailslots и NetBIOS).

Два последних раздела сопровождаются в качестве примеров исходными текстами программ, реализующих описываемые подходы и концепции.

Подбор материала для практической части курса учитывает как актуальность тем, так и реальные возможности проведения лабораторных работ.

1. КОМПЬЮТЕРНЫЕ СЕТИ – ОБЩАЯ ХАРАКТЕРИСТИКА

1.1. Основные определения

Компьютерная (вычислительная) сеть представляет собой совокупность *узлов*, или *хостов* (*hosts*), соединенных каналами передачи данных. Часть узлов предназначена для взаимодействия с конечными пользователями услуг сети – это *терминальные* (*оконечные*) узлы, или *терминалы*. Если сеть достаточно велика и сложна, то выделяются также специализированные управляющие (служебные) узлы, необходимые только для обеспечения ее функционирования: маршрутизаторы, шлюзы, серверы и т.д.

Каналы связи и служебные узлы образуют *базовую сеть передачи данных* (*СПД*). Базовая СПД – основа вычислительной сети, она, как правило, более сложна, более консервативна и более универсальна, чем подключенные к ней устройства пользователей (терминалы), то есть одна и та же базовая СПД используется многими различными пользователями и их группами, которые могут решать с ее помощью различные задачи.

Сказанное иллюстрируется рис. 1.1, где 1 – базовая СПД, 2 – управляющие узлы, 3 – каналы связи, 4 – оконечные узлы.

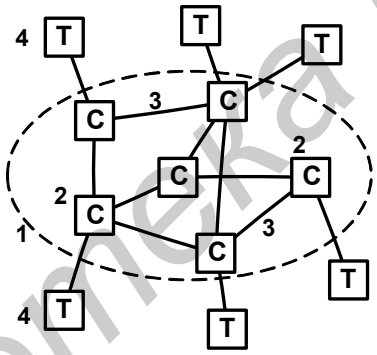

Рис. 1.1. Общая структура вычислительной сети

В свою очередь внутри базовой СПД могут выделяться магистральные каналы с большей пропускной способностью и обслуживающие их узлы – *опорная сеть* (*backbone*). К опорной сети подключаются уже не отдельные узлы, а целые подсети. Иерархия построения позволяет оптимизировать и упорядочить функционирование сети: если передаваемый поток данных не может быть замкнут в пределах подсети, его целесообразно передавать не через соседние подсети, а через опорную сеть.

Как разновидность систем обработки данных (СОД) вычислительная сеть характеризуется следующими признаками:

– неоднородность – состоит из разнотипных компонентов с различающимися функциями;

– децентрализованное управление – каждый компонент сети достаточно самостоятелен и сам контролирует свое функционирование;

– распределенность в пространстве – компоненты сети территориально удалены друг от друга.

Важным признаком, отличающим вычислительные сети от прочих СОД, является то, что в сетях основное внимание уделяется размещению, поиску и транспортировке данных в пределах сети безотносительно к их обработке конкретными узлами. В силу этого для сетей, по сравнению с другими СОД, менее свойственно единство цели: одна и та же сеть может служить решению многих задач, в том числе никак не связанных друг с другом.

Помимо общих классификационных признаков вычислительные сети имеют деление по некоторым специфическим категориям, которые рассматриваются ниже.

1.2. Распределение функций узлов в сети

Как отмечалось выше, сеть может быть представлена состоящей из своего рода ядра – базовой СПД – и подключенных к нему терминалов. В простейшем случае СПД состоит только из каналов передачи данных, и тогда все узлы сети играют роль терминалов, которые равноправны между собой и более или менее однородны по характеристикам. Такая сеть называется *одноранговой*. Терминалы (рабочие станции) такой сети могут при этом предоставлять сетевой доступ к своим ресурсам и выполнять некоторые сервисные функции в рамках всей сети, но это не становится основным их назначением. Одноранговые сети просты в построении и обслуживании, но для них возможности наращивания размеров и расширения круга решаемых задач ограничены.

Усложнение требований к сети приводит к необходимости выделения специализированных узлов (*серверов*), выполняющих только служебные функции и предоставляющих свои ресурсы остальным пользователям, причем гораздо более надежно и эффективно, чем рабочие станции, делающие это «по совместительству». Функции сервера могут касаться как обслуживания пользователей, так и обеспечения работы самой сети; во втором случае его правильнее относить уже к базовой СПД. Сеть с выделенными серверами называют *двухранговой*, и она является явно неоднородной. Такие сети сложнее, но обладают гораздо бóльшими возможностями.

Более глубокая иерархия с более чем одним уровнем распределения функций уже не добавляет принципиальных особенностей и отдельно не рассматривается.

1.3. Локальные и глобальные сети

Как распределенная СОД вычислительная сеть может состоять из различного, возможно, очень большого числа узлов, размещенных на определенной территории, причем эти количественные показатели существенно влияют на многие особенности её построения и функционирования. По территориальному и структурному признаку сети традиционно делятся на локальные и глобальные, между которыми возможен промежуточный тип.

Локальные сети объединяют относительно небольшое или по крайней мере прогнозируемое ограниченное количество близких по назначению и харак-

теристикам рабочих мест на небольшой территории (в одном помещении, здании или группе зданий), используя простое сетевое оборудование и дешевые, но достаточно быстродействующие каналы передачи данных. Как правило, самостоятельные подсети не выделяются, и задача выбора путей транспортировки информации не возникает. Локальные сети характеризуются простотой, малыми затратами на постройку и эксплуатацию, однако имеют ограниченные возможности.

Глобальные сети объединяют не столько отдельных абонентов, сколько сети более низкого ранга (*подсети*), и охватывают большую территорию (в настоящее время – практически весь земной шар и ближайшее околоземное пространство). Каждая из подсетей сохраняет относительную самостоятельность и сама может состоять из подсетей. Другая важная особенность глобальной сети – состав ее участников и круг выполняемых ею задач заранее не определены, поэтому возможность её перестройки, расширения и развития предусматривается изначально.

Региональные или *городские* сети занимают промежуточное положение: они сложнее и крупнее локальных, имеют внутреннюю иерархию, но не достигают масштабов глобальных. Примерами могут служить сети крупных предприятий, ведомств, административных единиц. Такие сети, впрочем, не всегда выделяются в отдельную группу.

Альтернативная классификация рассматривает в комплексе признаков также и назначение сетей: сети рабочих групп, сети отделов, сети кампусов и корпоративные.

Сети *рабочих групп* или *отделов* предназначены для обслуживания небольших обособленных групп пользователей, решающих общие задачи или относящихся к одному подразделению. Основная цель такой сети – обеспечение информационного обмена между пользователями и совместное использование ресурсов (например принтера или архива файлов), а главные характеристики – простота построения и обслуживания. Специальных средств эффективного структурирования обычно не требуется, хотя сеть отдела может включать в себя несколько подсетей отдельных рабочих групп.

Сети *кампусов* возникли как объединение более мелких сетей, что первоначально имело место в студенческих городках при университетах (откуда и название). В настоящее время к этому классу относят сети, объединяющие сети отделов одного предприятия (организации), находящиеся в одном или нескольких близко расположенных зданиях. Сеть кампуса должна обеспечивать обмен информацией между отделами и совместный доступ к ресурсам *без выхода* в глобальную сеть, то есть используя соединения только внутри этой сети. Соединения с глобальной сетью обычно также имеются, но рассматриваются как один из разделяемых ресурсов.

Корпоративные сети, или сети масштаба предприятия, отличаются от прочих как количеством узлов и подсетей, так и территориальной удаленностью их друг от друга, вследствие чего связь между ними может осуществлять-

ся с использованием глобальных соединений – корпоративная сеть использует существующую глобальную сеть как среду транспортировки данных. В этом случае особое значение приобретают функции управления неоднородной системой, учета и контроля пользователей и обеспечения безопасности информационного обмена.

Классификации не исключают друг друга и не всегда являются однозначными и исчерпывающими. Например, сеть БГУИР следует считать сетью кампуса, но при этом ей свойственны и черты корпоративных систем, например, наличие информационных систем со средствами управления пользователями или предоставление удаленного доступа посредством Internet.

В любом случае сети более низкого уровня входят, как правило, в состав сетей более высокого уровня в качестве их подсетей.

1.4. Топологии вычислительных сетей

Очень важным классификационным признаком, характерным именно для сетей, является *топология* – конфигурация сети «на местности», способ взаимного расположения узлов и соединений между ними. От топологии принципиальным образом зависят многие технические и эксплуатационные параметры сети. Выделяется несколько основных топологий (рис. 1.2).

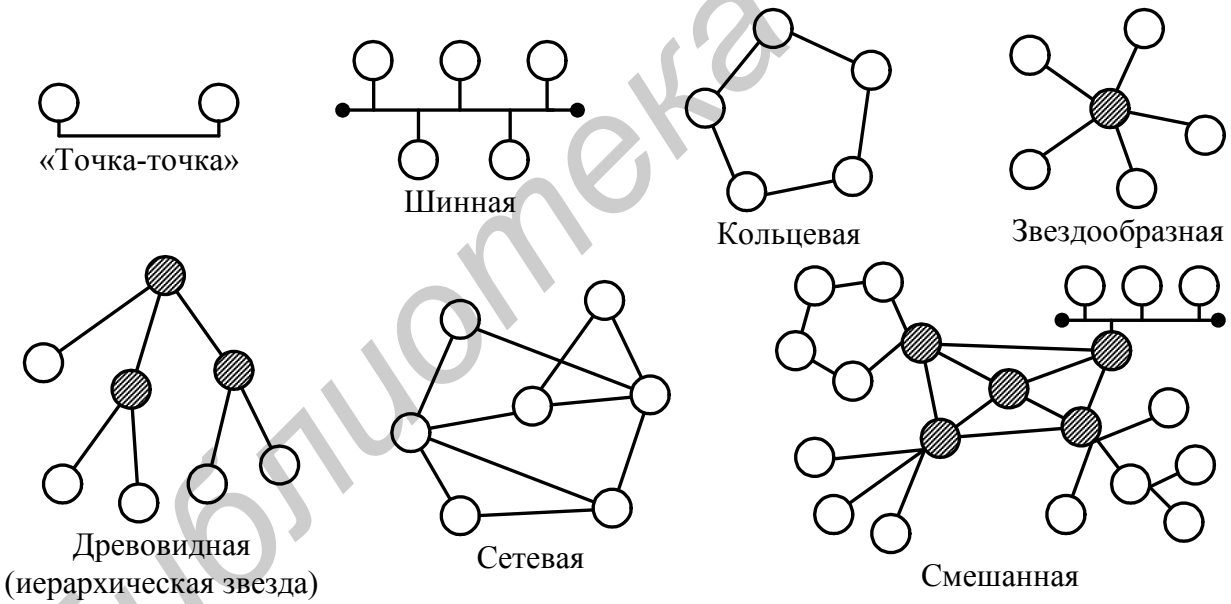

Рис. 1.2. Основные топологии вычислительных сетей

Топология *«точка-точка»* – простое соединение всего двух узлов. Полученную систему можно отнести к сетям лишь условно; адресация узлов не требуется, но может поддерживаться с целью унификации. К соединениям такого рода относится, например, временная связь, организованная между двумя ЭВМ посредством любого подходящего интерфейса. В более сложных сетях соединения «точка-точка» устанавливаются между маршрутизаторами опорной сети, а также между оконечными узлами и базовой СПД при использовании некоторых типов каналов, например телефонных линий.

Шинная топология – все узлы однотипным образом присоединяются к единому каналу передачи данных (*моноканалу*), пропускная способность которого разделяется между всеми пользователями. Реализация проста и экономична, хотя могут возникать сложности при прокладке кабеля, так как он должен «обойти» все узлы. Очевидные недостатки – понижены надежность (обрыв кабеля, как правило, выводит из строя всю сеть, хотя исправность отдельных узлов обычно безразлична) и пропускная способность (в каждый момент времени может функционировать не более одного передатчика). Как следствие, шинная топология эффективна при небольшом количестве подключенных узлов или относительно невысокой интенсивности передачи. Она характерна для сетей на коаксиальном кабеле (например ранние версии Ethernet 10Base5 и 10Base2) и для беспроводных технологий.

Кольцевая топология – напоминает шинную, но единый канал замкнут в кольцо. Так как нормальная передача возможна только в одном направлении, соединения узлов с кольцом не могут быть пассивными – узел включается в разрыв канала и имеет вход и выход, передача данных между которыми «внутри» узла обеспечивается им самим. Для двунаправленной передачи требуется второй параллельный канал. Такой подход наиболее естественен для оптоволоконных линий связи, но может применяться и в обычных проводных. Работоспособность «кольца» зависит от состояний как канала, так и всех узлов, что снижает надежность, а монтаж СПД обычно оказывается сложным и дорогостоящим. Основным же преимуществом кольцевой топологии является удобство обеспечения гарантированной пропускной способности, практически не зависящей от нагрузки на сеть, вследствие чего она эффективна для высокоскоростных магистралей, связывающих вычислительные сети. Кольцевую топологию имели и некоторые локальные сети, например TokenRing, CambridgeRing и им подобные.

Звездообразная топология – в сети выделяется центральный, обычно специализированный узел, к которому присоединяются все остальные (периферийные) узлы отдельными линиями связи. Условно можно рассматривать ее как множество соединений «точка-точка», сходящихся в одном узле. Стоимость базовой СПД может повышаться, так как увеличивается суммарная длина линий, но прокладка ее проста и технологична. Пропускная способность может быть высокой, так как периферийные узлы обмениваются данными с центральным независимо друг от друга. Критические факторы работоспособности сети – безотказность и производительность центрального узла, в то время как состояния периферийных узлов сказываются на ней незначительно. Принципиально реализуется на любых средах передачи, но наиболее характерна для обычных проводных, особенно на так называемой *витой паре*.

Древовидная (*звездообразная иерархическая*) топология – развитие звездообразной, от которой отличается наличием более чем двух уровней подчиненности: любой узел может становиться «центральным» для узлов следующего уровня иерархии. Часть узлов могут быть специализированными и служить

исключительно для поддержания соединений (коммутаторы различного типа). Такая топология позволяет распределить управляющие функции и трафик, разгружая «корневой» узел, использовать каналы с различной пропускной способностью и в результате более эффективно наращивать размеры сети. Наиболее характерная в настоящее время топология локальных сетей.

Сетевая топология – дальнейшее развитие древовидной при отступлении от принципа иерархического подчинения и допущении произвольных множественных связей между узлами. Наличие альтернативных путей доставки данных повышает пропускную способность и надежность системы за счет возможности перераспределения потоков, однако при этом возникает достаточно сложная задача *маршрутизации*. Вследствие этого сетевая топология практически не применима для локальных сетей, но эффективна для базовой СПД, причем маршрутизаторы внутри нее фактически связываются попарно соединениями «точка-точка».

Смешанная топология – сочетание перечисленных разновидностей, естественным образом возникает в глобальных сетях, объединяющих разнородные подсети. Наиболее вероятный вариант – опорная сеть с кольцевой или сетевой топологией (или комбинация таких сетей), к которой присоединяются подсети с любыми топологиями.

Примечание. В ряде случаев следует различать физическую топологию, область фактического распространения сигналов и логическую структуру связей. Так, типичная сеть Ethernet на витой паре имеет звездообразную или древовидную топологию физических линий связи, но все узлы одного сегмента, не разделенного переключателями или шлюзами, «слышат» друг друга и не могут вести передачу одновременно, то есть характер распространения данных соответствует моноканалу. Другой пример – централизованное «звездообразное» управление узлами, которое может быть реализовано в сети с любой топологии, например шинной.

1.5. Принципы передачи данных в сетях

Различают два основных способа организации передачи данных в сетях: с установлением соединения и без установления соединения.

При передаче *без установления соединения* данные передаются в виде законченных самостоятельных блоков – *датаграмм* (*datagram*). Каждая датаграмма доставляется к получателю по произвольному маршруту и независимо от других датаграмм, причем подтверждения о получении (*квитирование*) не предусматриваются, поэтому не гарантируется ни порядок следования датаграмм, ни единственность доставленного экземпляра, ни сам факт доставки – контролируются обычно только искажения каждой отдельной датаграммы. Такой способ передачи прост, экономичен, но для многих применений недостаточно надежен.

Передача *с установлением соединения* обладает бóльшими возможностями. Главное отличие от датаграммной передачи – обеспечение целостности и

упорядоченности потока передаваемых данных. Независимо от способа организации потока данных (см. ниже) порции данных доставляются получателю строго в том порядке, в котором они были отправлены, а прерывание потока своевременно распознается. Это достигается за счет нумерации порций данных и организации встречного потока подтверждений о получении (*квитанций*). Таким образом образуется *виртуальный канал* передачи данных, для прикладных программ близкий по своим свойствам файлу или потоку ввода-вывода. Вместе с тем потоковая передача сложнее, требует специальных процедур установления соединения и дополнительных затрат на контроль его состояния, создает дополнительную нагрузку на линии связи в виде встречного потока квитанций.

Для большинства задач, связанных с обменом значительными объемами данных, более подходит взаимодействие с установлением соединения. Основное применение датаграммной передачи – однократные сообщения ограниченной длины при условии, что надежность доставки не слишком критична.

Если контакт между двумя узлами необходимо поддерживать в течение длительного времени, как в случае передачи с установлением соединения, возникает задача *коммутации* – распределения имеющихся физических каналов передачи данных для создания каналов виртуальных. Различают три основных метода коммутации: коммутация каналов, сообщений и пакетов.

Коммутация каналов (*circuit switching*) – предоставление виртуальному каналу цепочки физических каналов, сохраняющейся на все время жизни соединения и находящейся в монопольном владении этой пары абонентов. Такой канал в простейшем случае представляет собой отдельную линию связи, но может быть организован и с помощью временно́го или частотного разделения сигнала в общей для множества каналов среде передачи. Типичным примером первого служит обычная коммутируемая телефонная сеть, второго – разделение «общего» эфира между радиопередатчиками. Коммутация каналов позволяет достичь максимальной для конкретного типа канала производительности, предоставляемой отдельно взятой паре абонентов. Крупными недостатками являются непроизводительный простой канала в случае, если обмен не является непрерывным, и длительная процедура установления соединения или его восстановления после сбоя.

Коммутация сообщений (*message switching*) отличается тем, что поток данных разбивается на отдельные завершенные сообщения и реальное (физическое) соединение создается только на ограниченное время – для передачи одного сообщения, в остальное время те же ресурсы могут быть использованы для передачи сообщений других потоков. Неудобством такого подхода оказалась переменная и часто слишком большая длина сообщений, затрудняющая создание коммуникационного оборудования, и в настоящее время этот вид коммутации не используется, однако он послужил прототипом современной коммутации пакетов

Коммутация пакетов (*packet switching*) – наиболее часто используемый метод. Основная идея состоит в разбиении всех передаваемых данных на пор-

ции одинакового или хотя бы ограниченного небольшого размера – *пакеты*. Каждый пакет доставляется как датаграмма, то есть независимо от остальных пакетов и, возможно, по собственному маршруту, благодаря чему достигается более равномерная загрузка сети и бóльшая устойчивость против сбоев – доставка может быть выполнена другим маршрутом, и повторить при этом придется лишь отдельные пакет. Размер пакета выбирается удобным для коммутаторов, что упрощает и удешевляет аппаратуру. К проблемам пакетной коммутации могут быть отнесены бóльшая доля служебной информации в трафике, бóльшая нагрузка на маршрутизаторы и необходимость сложных процедур сборки пакетов. Однако преимущества метода оказываются в большинстве случаев более существенными.

1.6. Иерархическое построение сетей. Открытые системы

Необходимость унификации построения разнородных систем и сетей и взаимодействия их друг с другом привела к переносу на них концепции *открытой системы* – системы, построенной на основе открытых спецификаций. Применительно к сетям под открытостью понимают открытость средств взаимодействия систем, связанных в сеть, и открытой является такая система, которая готова взаимодействовать с другими системами с использованием стандартных (унифицированных) правил. Такая система описывается *моделью взаимодействия открытых систем* (*Open System Interconnection – OSI*), принятой Международной организацией по стандартизации (ISO) в 1983 г.

Стандартная модель OSI состоит из семи уровней, функции которых регламентированы. Уровни представляют собой функционально завершенные модули, поддерживающие связь непосредственно с соседними модулями и через них – с модулями другой системы. Для связи между составными частями (уровнями, модулями) систем определяются соответствующие протоколы и интерфейсы (рис. 1.3).

Протокол – набор правил и процедур взаимодействия между одноименными уровнями различных систем. Протоколы обеспечивают корректную связь участников взаимодействия в сети.

Интерфейс – набор правил и средств их реализации для взаимодействия между соседними уровнями одной системы. Интерфейсы обеспечивают возможность модульного построения системы.

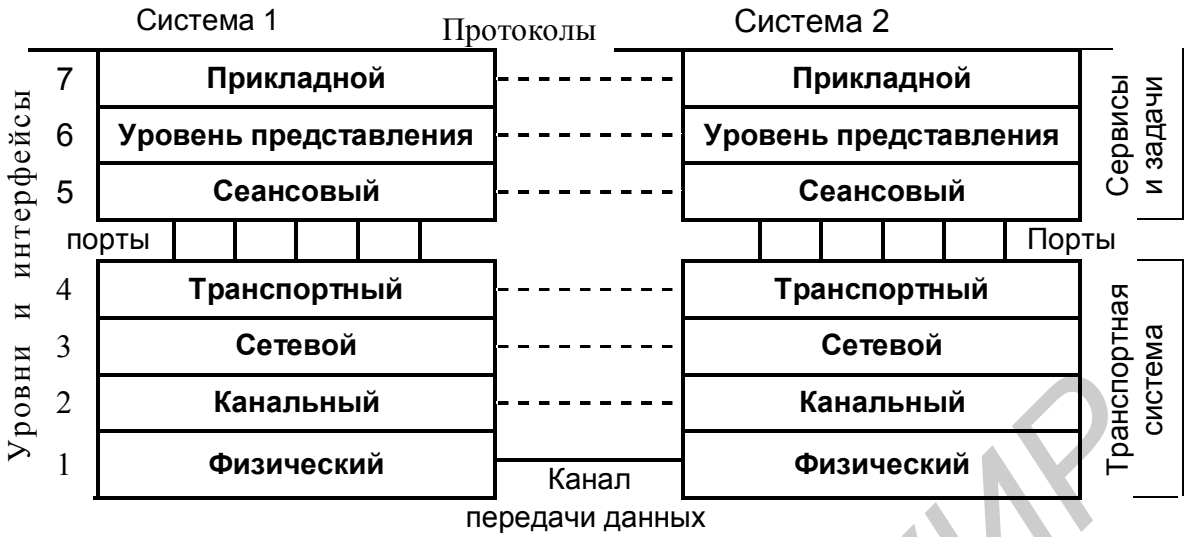

Рис. 1.3. Модель взаимодействия открытых систем OSI

Ниже описаны уровни иерархической модели.

Физический уровень (1) – аппаратура подключения к сети. Этот уровень обеспечивает взаимодействие со средой передачи данных на уровне сигналов и обычно характеризуется существенной вероятностью возникновения ошибок, связанных с физикой процессов в реальной среде.

Канальный уровень (2) – осуществляет логическое управление физическими устройствами и повышение достоверности передачи: контроль и, возможно, исправление ошибок.

Сетевой уровень (3) – организует поиск адресов в сети и перенаправление передаваемых адресованных данных (маршрутизация). Протокол IP принято относить к транспортному уровню, хотя он же может решать и задачу доставки сообщений.

Транспортный уровень (4) – выполняет передачу от одной точки (адреса) к другой с необходимым контролем и (возможно) дополнительным сервисом. Примеры протоколов транспортного уровня – TCP и UDP (TCP решает также и задачи сеансового уровня).

Все нижние уровни до транспортного включительно называют *транспортной системой*, она является основой для вышестоящих уровней. Для них транспортный уровень создает («прокладывает») так называемые *порты* – унифицированные точки доступа к функциям транспортной системы.

Сеансовый уровень (5) – обеспечивает установление соединений (сеансов) между взаимодействующими системами (процессами) и управление ими. Задачи этого уровня могут решаться также и транспортными протоколами (пример – соединения с использованием TCP).

Уровень *представления* (6) – служит для преобразования форматов данных (например, порядок байт в словах, вид кодировки символов, форму записи адресов и т.д.) в соответствии с правилами конкретного программного обеспечения следующего 7-го уровня в конкретной системе.

Прикладной уровень (7) – конечные приложения, как чисто прикладные (программы пользователя), так и служебные (так называемые *службы* или *сервисы*), возможно, используемые другими программами.

В зависимости от назначения узла реализация уровней модели может быть различной и необязательно полной. Так, специализированные служебные узлы базовой СПД, например маршрутизаторы, могут ограничиваться лишь уровнями транспортной системы, зато с мощной аппаратной поддержкой для повышения производительности. В оконечных «прикладных» узлах, наоборот, верхние уровни присутствуют обязательно, но аппаратная поддержка есть только у нижних: физического и частично канального. Уровни выше транспортного реализуются почти исключительно программно.

В случае локальной сети модель претерпевает модификацию. Как было отмечено выше, существенное отличие ЛВС состоит в том, что используется простейшая топология (шинная, кольцевая или древовидная, но с трансляцией сигнала по всем «ветвям») и соответствующая среда передачи данных – *моноканал*. Вследствие этого сетевой уровень (3) естественным образом упраздняется, а функции канального (2) усложняются и могут быть разделены на 2 подуровня (рис. 1.4).

Рис. 1.4. Модификация модели OSI в ЛВС

Подуровень 2.1 – *управление доступом к моноканалу*, или *Media Access Control* (*MAC*), осуществляет идентификацию узлов, выбор моментов для передачи данных и контроль ее результативности в сети шинной (или сводящейся к ней) топологией. В результате образуется *логический канал* передачи данных, используемый вышестоящими уровнями.

Подуровень 2.2 – *управление логическим каналом*, осуществляет обычные функции канального уровня: формирование пакетов, управление их приемом и передачей, контроль ошибок.

В рамках модели отдельные системы обмениваются между собой блоками данных, которые последовательно передаются от верхних уровней системы-источника к нижним, далее через физическую среду передачи данных на нижний уровень системы-приемника и через аналогичную иерархию на ее верхний уровень. На каждой ступени блок подвергается изменениям, в основном связанным с его разбиением на подблоки, требуемые соответствующим уровнем, или сборкой из таких подблоков, а также кодированием или декодированием информации. Кроме того, при перемещении от уровня к уровню (рис. 1.5) про-

исходит *инкапсуляция* блоков: при передаче каждый из уровней снабжает полученный «сверху» блок заголовком (Зг) и/или концевиком (Кн), которые идентифицируют начало и конец блока и несут служебную информацию о нем, например, адрес, контрольную сумму и прочее. При приеме одноименный уровень анализирует заголовок и/или концевик, выделяет блок, выполняет необходимые обратные преобразования содержимого и передает его «вверх».

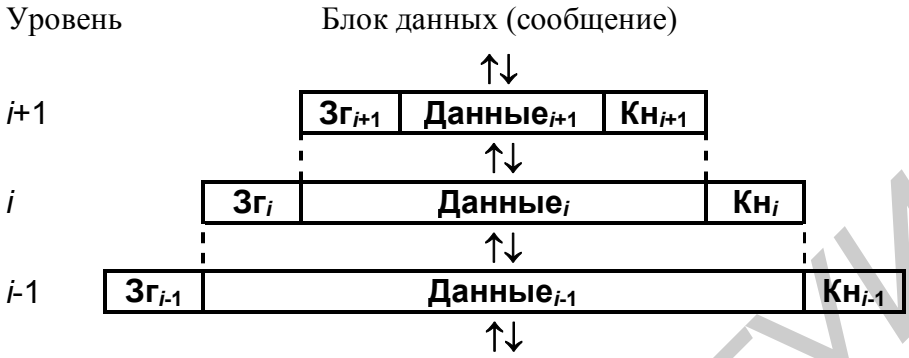

Рис. 1.5. Инкапсуляция блоков данных

Такую иерархию принято называть *стеком протоколов*. Действительно, в системе «виден» только самый «верхний» протокол, и, чтобы «добраться» до «нижних» протоколов, нужно последовательно «снимать» «верхние». Стеки протоколов различных реальных систем в силу многих технических, исторических и организационных причин могут не вполне соответствовать модели OSI, отличаясь от нее количеством уровней и распределением их функций, но сам иерархический принцип в той или иной мере соблюдается обязательно.

Существует также и стек протоколов собственно OSI, полноценно реализующий концепцию модели OSI и призванный заменить старые системы, но, несмотря на поддержку со стороны правительства США и некоторых крупных компаний, например AT&T, его применение в настоящее время ограничено; в рамках данного пособия не рассматривается

Теоретически строгое соблюдение принципов построения и унификация протоколов и интерфейсов позволяет строить системы с полностью взаимозаменяемыми модулями одноименных уровней и обеспечивать связь между ними. На практике серьезной проблемой становится большое разнообразие реальных систем с различными внутренними стандартами, а также снижение эффективности вследствие затрат на поддержание совместимости. Поэтому правильнее будут сказать, что уровни и протоколы взаимодействия между ними в рамках конкретной модели в той или иной мере унифицированы (стандартизованы) и контролируются соответствующими организациями.

1.7. Идентификация участников взаимодействия

Участниками взаимодействия в компьютерных сетях являются программы различного назначения, выполняемые на различных узлах сети. Количество таких участников в крупной сети может быть очень велико, поэтому система их идентификации должна быть надежной, эффективной, гибкой и при этом

имеющей возможность унификации между различными сетями. Схема идентификации, удовлетворяющая этим требованиям, должна быть многоуровневой и опираться на иерархическую модель взаимодействия.

В наиболее общем виде принято считать, что идентификатор состоит из двух частей: идентификатора (*адреса*) узла и идентификатора программы на этом узле (рис. 1.6).

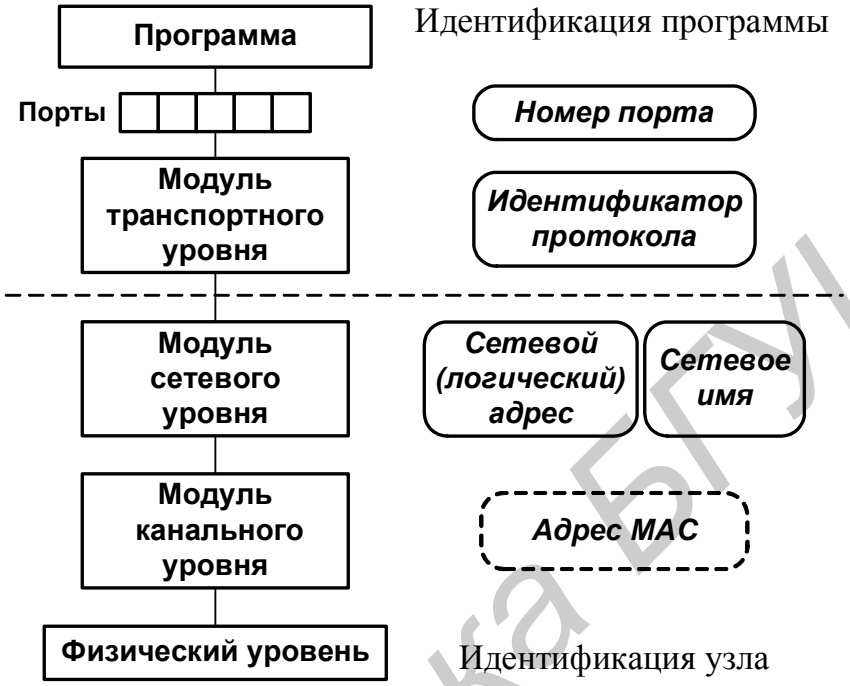

Рис. 1.6. Идентификация участника взаимодействия

В свою очередь, адрес узла обычно бывает представлен как минимум двумя уровнями:

– *логический* (*протокольный*) адрес – адрес, которым оперирует протокол сетевого уровня, поэтому часто называется также *сетевым* адресом;

– физический (MAC) адрес – используется на канальном уровне (низший физический уровень адресации как таковой не предусматривает).

Преобразование физических адресов в логические и обратно называют *разрешением* адресов. Оно прозрачно для прикладных программ, которые обычно не видят MAC-адреса́ и напрямую их не используют.

Для удобства пользователя (человека) и повышения гибкости системы узлы могут получать символические обозначения – *сетевые имена*. Преобразование сетевых имен в адреса выполняется обычно специальными службами, например в сетях Internet это *DNS* и *NAT*.

В качестве идентификатора программы принято использовать номер порта транспортной системы, который предоставлен этой программе. Сетевой порт, в отличие, например, от портов ввода-вывода в адресном пространстве процессора или портов подключения внешних устройств, является чисто логическим объектом и не имеет отражения в аппаратуре: он создается заново, когда необходимо предоставить его программе. Различные транспортные протоколы создают порты независимо друг от друга, и одновременно могут сущест-

вовать несколько портов с одним и тем же номером, но только у разных протоколов; в пределах одного протокола номер порта всегда уникален. Поэтому необходим также и идентификатор транспортного протокола.

Таким образом, полный идентификатор, используемый программами для доступа друг к другу, включает три звена:

<сетевой адрес>:*<протокол>*:*<порт>*,

причем протокол часто бывает задан неявно.

Части этого идентификатора содержатся в заголовках сообщений (пакетов) соответствующего уровня стека протоколов и обрабатываются модулями своего уровня.

Иерархическая архитектура средств поддержки сети позволяет иметь в одном узле несколько экземпляров модуля каждого уровня, отличающихся реализуемыми ими протоколами. Например, система может включать несколько физических интерфейсов, каждому из которых сопоставлены собственные адреса́ (MAC и логические), единый модуль сетевого уровня и поверх него – несколько транспортных. В этом случае выполняется *мультиплексирование* сообщений в рамках системы, основываясь на разборе идентификаторов. Подробнее об использовании идентификаторов см. в разделах, посвященных протоколам семейства TCP/IP и их программированию.

1.8. Стандарты компьютерных сетей и источники информации

Учебники, справочники и периодические публикации не могут дать исчерпывающих сведений о построении и функционировании сетей, поэтому в большинстве случаев необходимо обращаться также к первоисточникам: спецификациям и стандартам. В силу особенностей развития компьютерных сетей в этой области существует целый ряд стандартов, сформированных организациями различного уровня, производителями сетевого оборудования и отдельными инициативными группами, причем одни из них являются результатом целенаправленной и централизованной разработки, другие же складываются фактически, в ходе развития той или иной технологии. Перечислим некоторые из разрабатывающих и контролирующих стандарты организаций.

Международная организация по стандартизации (*International Organization for Standardization* или *International Standards Organization*, *ISO*) и ее представитель в США Американский национальный институт стандартов (*American National Standards Institute*, *ANSI*) – занимаются вопросами стандартизации в самых различных областях, в том числе и коммуникаций.

Международный союз электросвязи (*International Telecommunications Union*, *ITU*) и, в частности, входящий в его состав Сектор телекоммуникационной стандартизации (*ITU Telecommunication Standardization Sector*, *ITU-T*), ранее Международный консультативный комитет по телефонии и телеграфии *МККТТ* (*Consultative Committee on International Telegraphy and Telephony*, *CCITT*), результаты работы которого регулярно издаются в виде так называемых Книг.

Например, том VIII содержит спецификации семейств *V* (V.*x*, передача данных по телефонным сетям) и *X* (X.*x*, сети пакетной коммутации).

Институт инженеров по радиотехнике и радиоэлектронике (*Institute of Electrical and Electronics Engineers*, *IEEE*); в частности, группой 802 этой организации были разработаны стандарты для физического и канального уровня локальных сетей – спецификации *802.x*.

Общество по вопросам Internet (*Internet Society*, *ISOC*) – основная организация, отвечающая за стандартизацию в сетях Internet.

Очень важную роль для сетей Internet играют *RFC – Requests for Comments*. Это большой набор документов (сейчас около четырех тысяч), посвященных самым разным аспектам: обсуждение концепций, исследования отдельных проблем, описание конкретных протоколов и служб и т.д. RFC контролируются группой *Internet Engineering Task Force* (*IETF*) в составе Совета по архитектуре Internet (*Internet Architecture Board*, *IAB*) под управлением ISOC. Документы не имеют строгой схемы именования, а просто получают порядковые номера. Изначально они рассматривались как разъяснения и рекомендации, но часть из них получила статус официальных стандартов Internet: например, набор протоколов TCP/IP и большинство базовых элементов Internet целиком описаны именно в RFC. Эти документы в электронном виде находятся в свободном доступе (ресурс www.rfcs.org и ряд других).

2. СЕТЕВОЙ ПРОТОКОЛ IP И СЕТИ IP

2.1. Краткая история IP-сетей

Дальнейший материал данного пособия относится в основном к сетям, ориентированным на протокол *IP* – IP-сетям. В первую очередь, к этой категории относится глобальная сеть *Internet*.

Сеть Internet является в настоящее время доминирующей глобальной сетью, фактически объединяющей большинство существующих сетей. Несмотря на сложность и неоднородность этой «сети сетей», можно выделить элементы, составляющие её основу и обеспечивающие её единство. В первую очередь это пакетный принцип передачи данных и реализующий его протокол IP с группирующимися вокруг него протоколами и службами.

Особенности пакетной передачи – индивидуальная доставка небольших порций данных, обеспечивающая своевременное обнаружение ошибок и сбоев и достаточно эффективное устранение их последствий, – оказались востребованы на начальном этапе построения глобальных сетей. Это было вызвано как объективной ненадежностью существовавших каналов связи и оборудования, так и желанием обеспечить работоспособность сети при повреждении отдельных ее участков. Последнее особенно интересовало военных, которые играли существенную роль в инициативе создания глобальной сети.

С целью изучения и последующего использования возможностей сетей пакетной коммутации в 1969 г. началась экспериментальная разработка сети

ARPANET. Ею руководило агентство Министерства Обороны США DARPA (ARPA – Advanced Research Project Agency), основными исполнителями стали Калифорнийский университет в Лос-Анжелесе (UCLA), Калифорнийский университет в Сан-Бернардино (UCSB), Стэнфордский исследовательский институт (SRI) и университет штата Юта (UU), а также фирма Bolt, Beranek and Newman (BBN). Созданная сеть получила название ARPAnet. Ее технической основой стали изготовленные BBN процессоры сообщений IMP (Interface Message Processor), а протоколом сетевого взаимодействия – NCP (Network Control Program).

С 1975 по 1979 г. DARPA разработало для замены NCP новое семейство протоколов, в первую очередь протоколы межсетевого взаимодействия *IP* (*Internet Protocol*) и управления передачей *TCP* (*Transport Control Protocol*). С 1983 г. по требованию DARPA все узлы, входящие в ARPAnet, должны были использовать в качестве основы именно эти протоколы. Более того, рекомендации и предложения правительства США, касающиеся сетей, также предусматривали использование TCP/IP, что практически сделало его стандартом. В том же году произошло разделение ARPAnet на сеть военного назначения DDN (Defense Data Network), или MILNET, и сеть DARPA Internet.

Параллельно с ARPAnet развивался и целый ряд сетей, также базирующихся на TCP/IP и связанных с ARPAnet и посредством его друг с другом: CSNET, BITNET, UUCP, USENET и т. д. На основе CSNET (Computer Science Network) при поддержке Национального научного фонда NSF в 1986 г. была создана NSFnet – опорная сеть, объединяющая поначалу сети научных и учебных учреждений, а затем и региональные сети. NSFnet быстро развивалась и уже в ближайшие годы несколько раз модернизировалась, в результате чего к 1991 г. пропускная способность опорной сети возросла почти на три порядка (с 56 кбит/с до 45 Мбит/с), а число региональных узлов достигло 16. Участниками сети становились также и коммерческие организации и сети.

По многим техническим и экономическим показателям NSFnet была эффективнее, чем ARPAnet. К 1990 г. обслуживание ARPAnet фактически прекратилось, и вскоре проект был официально закрыт.

В 1993 г. контроль за расширяющейся NSFnet был передан множеству организаций (*провайдеров*), которые могли строить собственные опорные сети, взаимодействующие посредством «точек доступа» *NAP* (*Network Access Points*) на основе концепции равноправного информационного обмена (*peering*). Базовые функции управления сетью как целостной системой, например, контроль за выделением адресов и сетевых имен, также были распределены между несколькими организациями. Наконец, в 1995 г. опорная сеть собственно NSFnet была закрыта, и Intenet фактически приобрел современный вид.

2.2. Иерархия протоколов в IP-сетях

Архитектура IP-сетей сложилась до стандартизации модели OSI и состоит всего из четырех уровней, имеющих собственные наименования и образующих

стек протоколов IP (или *TCP/IP*). Тем не менее уровни стека протоколов IP в целом соотносятся и с уровнями OSI (рис. 2.1). Отметим, что излагаемый далее материал более ориентирован на стек IP, а не модель OSI.

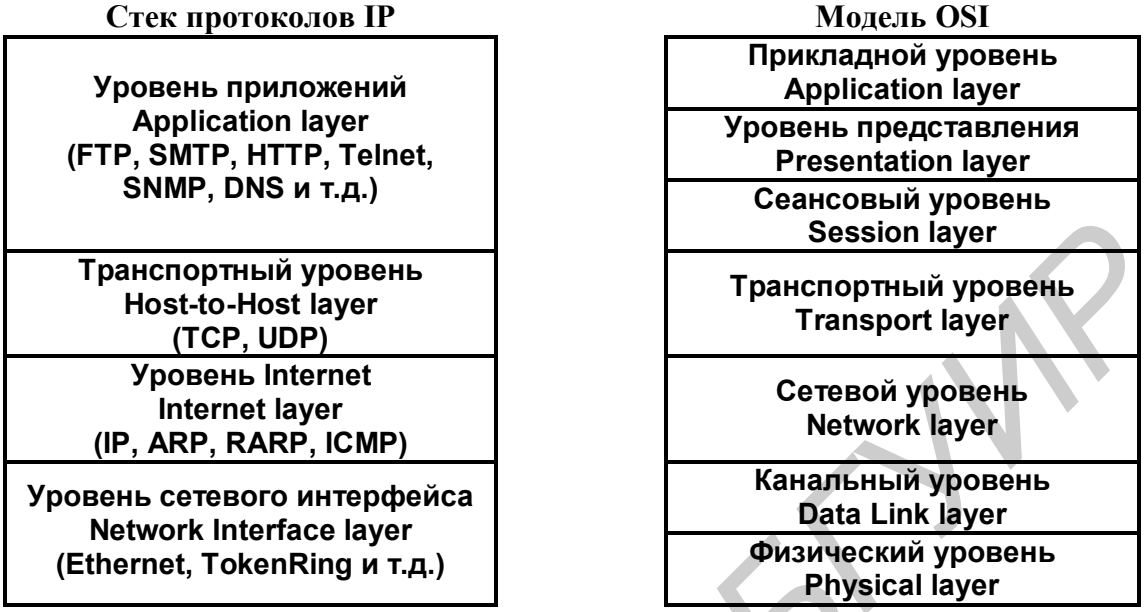

Рис. 2.1. Стеки протоколов IP и OSI

Отметим основные несоответствия: два нижних уровня объединены, протоколы транспортного уровня выполняют функции отсутствующего в стеке сеансового, уровень представления как самостоятельный не выделен.

Протоколы и службы IP-сетей образуют аналогичную иерархию. Некоторые из них схематично показаны на рис. 2.2.

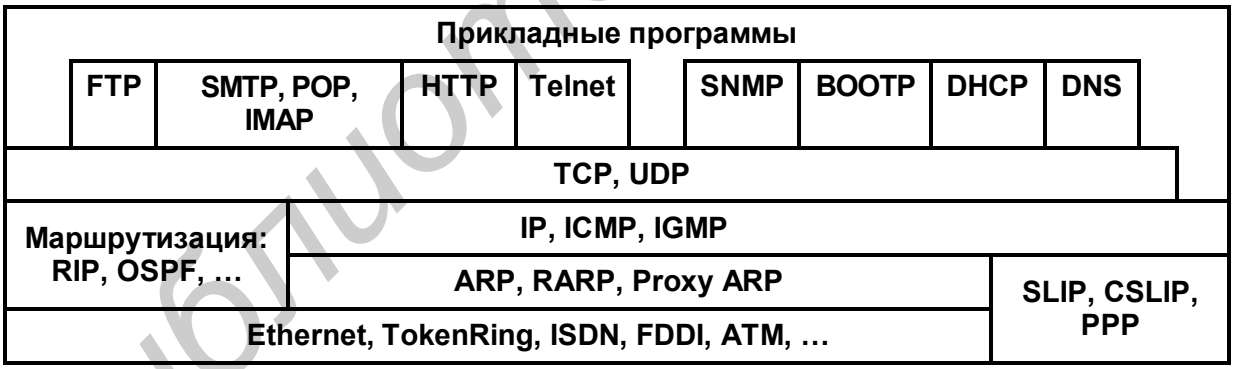

Рис. 2.2. Иерархия протоколов и служб в IP-сетях

Центральное место в иерархии занимают сетевой протокол IP и транспортные TCP и UDP. Хотя они могут быть реализованы не только поверх IP, в подавляющем большинстве случаев используются совместно с ним как устоявшаяся комбинация. Нижний и верхний уровни уже не связаны однозначно именно с IP, TCP и UDP, однако многие из них развивались совместно с IP-сетями и могут рассматриваться как составная часть современного Internet.

Формальная принадлежность протокола к тому или иному уровню не всегда соответствует его фактической роли. Например, на данной схеме:

– протоколы семейства ARP относятся к тому же уровню, что и IP, однако функционально они стоят между уровнями Internet и Network Interface;

– протоколы ICMP и IGMP выполняют сугубо служебные функции и обеспечивают расширения IP, но их сообщения инкапсулируются в IP;

– ряд служебных протоколов и сервисов, обеспечивающих функционирование IP, TCP и UDP, сами используют транспорт TCP или UDP.

Следует также отметить две обособленные группы протоколов, подробное рассмотрение которых выходит за рамки настоящего пособия:

– протоколы маршрутизации – реализуются только служебными узлами (маршрутизаторами) и выполняют только служебные задачи, для прикладных программам они совершенно прозрачны;

– протоколы поддержки соединений «точка-точка» – актуальны для узлов, использующих традиционные телефонные линии связи, последовательные интерфейсы типа RS 232 и прочие подобные соединения; реализуются функции нижних уровней и унификация со стандартным стеком.

2.3. Протокол IP

IP – Internet Protocol (version 4)
Идентификатор при инкапсуляции в кадр Ethernet – 0800h.
Идентификатор при инкапсуляции в пакет IP – 4.
Описание – RFC 791 (основное), RFC 760 (первое описание), RFC 950, 1122, 1123, 1812, 1918, 1517-1520 и др.

IP определяют как «протокол межсетевого взаимодействия»: его основная задача – доставка сообщений (пакетов) между абонентами, в том числе находящихся в различных сетях (подсетях), для чего необходимо обеспечить идентификацию абонентов и построение пути (маршрута) между ними. В IP реализуется пакетный принцип передачи данных.

Примечание. Слово «internet» здесь еще не является термином, и название протокола может быть переведено как «межсетевой протокол», или «протокол общения сетей». Таким образом, Internet в современном значении вторичен по отношению к IP: не IP есть протокол для использования в Internet, а Internet – сеть, использующая IP. В иерархической модели TCP/IP уровень, обозначенный как «internet», принято называть «сетевым» или «межсетевым».

2.3.1. Адресация IP

Для идентификации любого участника взаимодействия в сети принято использовать три компоненты: сетевой адрес, идентификатор протокола и номер порта. Протоколы сетевого уровня, в том числе и IP, отвечают за адресацию узлов, оперируя с сетевыми адресами, называемыми также *логическими*, или *протокольными*. Процедуру преобразования их в адреса *канального* уровня (MAC), или аппаратные принято называть *разрешением* адреса. Так как принципы адресации и протоколы в различных сетях могут сильно отличаться, вве-

дено понятие *семейства адресов* (*address family*), каждому из которых присвоены уникальные идентификаторы. Например, согласно *RFC 1700* семейству адресов IP версии 4 (*IPv4*) присвоен идентификатор 1 – первый после резервированного нулевого, IPv6 – 2, и т. д.

Роль протокольного адреса в сетях Internet играет IP-адрес.

Адреса́ IP (версия 4) – 32-разрядные, принятая форма записи – *точечная*: значения октетов (байт) а́дреса записываются в виде десятичных чисел, отделенных друг от друга точками, порядок октетов – начиная со старшего. Такая запись легко читаема для человека, но, к сожалению, сравнительно сложно преобразуется в двоичное представление, хотя интерпретация адреса предполагает представление его битовыми полями (см. ниже).

Адрес IP изначально принято представлять состоящим из двух частей: *идентификатора сети* и *идентификатора хоста* (узла в сети). Предполагалось, что такая схема позволит гибко приспосабливать протокол к условиям адресации в глобальной сети, структурированной на ряд подсетей. Она сохранилась в основном и сейчас, но претерпела ряд модернизаций (см. ниже).

В зависимости от конкретного числа разрядов, отводимых под каждую из частей адреса, выделяются *классы адресов* IP, формально различающиеся первыми (старшими) разрядами и соответственно занимающие строго заданные диапазоны значений. Всего определено 5 классов адресов, первые 3 из которых считаются основными (рис. 2.3).

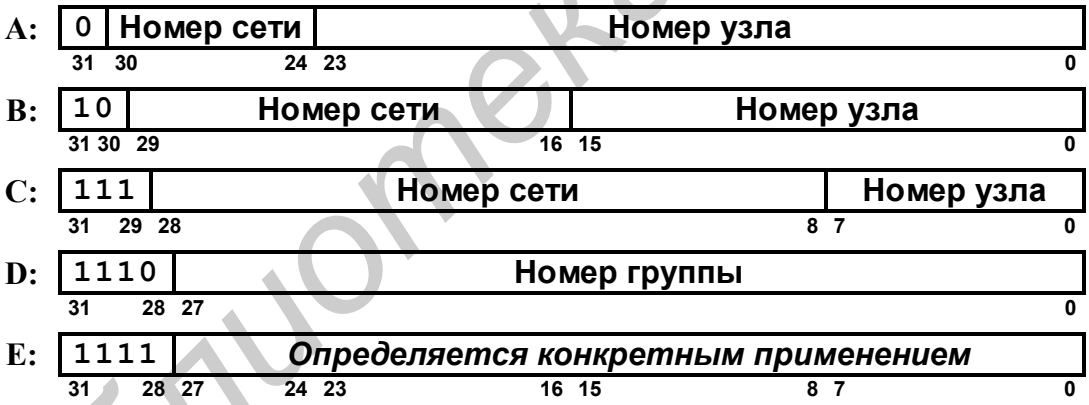

Рис. 2.3. Структура адресов IP

Класс *A*: старший разряд 0, идентификатор сети – 7 разрядов, идентификатор узла – 24 разряда (3 октета), диапазон адресов – от 0.0.0.0 до 127.255.255.255 (реально меньше, так как некоторые адреса отнесены к *зарезервированным*, см. ниже). Предназначался для немногочисленных сетей с потенциально очень большим количеством узлов, был быстро исчерпан и в настоящее время для новых сетей не используется.

Класс *B*: старшая пара разрядов 10, идентификатор сети – 14 разрядов, идентификатор узла – 16 разрядов (2 октета), диапазон адресов от 128.0.0.0 до 191.255.255.255. Предназначен для «средних» сетей, в настоящее время также в значительной мере исчерпан.

Класс *C*: старшая триада 110, идентификатор сети – 21 разряд, идентификатор узла – 8 разрядов (1 октет), диапазон адресов от 192.0.0.0 до 223.255.255.255. Предназначен для очень многочисленных «малых» сетей.

Класс *D*: старшая тетрада 1110, остальные 28 разрядов – идентификатор *группы*. Используются для *групповой адресации*. Многие адреса класса D закреплены за известными пользователями-группами (см. *RFC 1700*).

Класс *E*: старшая тетрада 1111. Считается экспериментальным.

Изначально деления на классы не существовало, и под идентификатор сети отводилось только восемь разрядов, но очень скоро стало очевидным, что количество сетей превзойдет возможности такой идентификации. Лучшим с точки зрения совместимости оказалось решение ввести три класса адресов, различающихся старшими битами, с которых и начинается интерпретация адреса. Так как номера сетей выделялись в порядке возрастания, уже выданные «доклассовые» адреса автоматически оказались принадлежащими классу *A* (вполне очевидно, что они заняты в основном «родоначальниками» ARPANET и Internet). Еще позже были добавлены классы *D* и *E*.

Кроме того, ряд адресов и их диапазонов имеют специальное значение (см. также *RFC 1700*). Ниже перечислены некоторые из них.

0.0.0.0 – «этот узел», идентифицирует сам узел – источник пакета или узел с неопределенным адресом (например в процессе инициализации).

Если нулями заполнен только адрес хоста, а адрес сети задан, то адрес воспринимается как «эта сеть» – любой узел указанной сети (подсети).

255.255.255.255 – «все узлы», идентифицирует любой узел сети и позволяет организовать *широковещательную* (*broadcast*) рассылку; во избежание засорения сети широковещание ограничивается локальной сетью (подсетью), за пределы которой пакеты с таким адресом не передаются, поэтому адрес называется *ограниченным* широковещательным.

Если единицами заполнен только адрес хоста, а адрес сети задан, то адрес воспринимается как «все узлы» указанной сети (подсети). Можно сказать, что ограниченный широковещательный адрес является частным случаем, когда идентификатор сети тоже не определен.

127.x.x.x – *локальный* адрес, идентифицирующий тот узел, на котором выполняется программа, ассоциируется с сетевым именем *localhost*; обычно используется значение 127.0.0.1. Сообщения, адресованные *localhost*, не передаются IP-модулем вниз по стеку протоколов, а обрабатываются им же. Адрес отправителя для таких сообщений будет совпадать с адресом получателя.

10.x.x.x, 172.16.x.x – 172.31.x.x (префикс 172.16/12 – см. ниже) и 192.168.x.x (префикс 192.168/16) – «внутренние» адреса, или адреса «локального Internet» (*private internets*, *private address space*): пакеты, направленные на эти адреса, действуют только в локальной сети, никогда не отсылаются из нее в глобальную и соответственно никогда из нее не приходят. Следовательно, уникальность таких адресов необходимо поддерживать только в преде-

лах локальной сети, но при этом, очевидно, узел не может иметь прямого выхода в глобальную сеть (см. *RFC 1918*).

Примечание. Перечисленные выше требования и соглашения актуальны для сетей, входящих в глобальную сеть. Изолированные сети и обладающие сетевым интерфейсом отдельные узлы могут использовать любые значения IP-адреса, однако явно противоречить принятым нормам нежелательно.

Серьезной проблемой адресации IP является истощение адресного пространства, что обусловлено как объективной ограниченностью пространства 32-разрядных адресов, так и неравномерным и нерациональным их расходованием. Если первое ограничение грозит стать реальным лишь в перспективе, хотя и недалекой, то второе потребовало решения уже достаточно давно.

Так, например, число возможных идентификаторов сетей класса *C* пока может считаться достаточным, но для многих организаций количество узлов, допускаемое сетью этого класса, слишком мало. «Емкость» сетей класса *B* избыточна для типичной «средней» организации, но идентификаторов самих этих сетей не хватает уже достаточно давно. Сети класса *A* потребляют половину адресного пространства, реально используя лишь небольшую его часть.

Таким образом, источником проблемы являются как ограниченная длина адреса, так и недостаточная гибкость схемы, основанной на классах адресов. Возможны 3 пути решения проблемы:

1) увеличение длины протокольного адреса, что является наиболее радикальным решением, но связано с многочисленными трудностями внедрения новой технологии в масштабах глобальной сети; реализовано в версии 6 протокола IP (*IPv6*);

2) применение «внутренних» адресов, для которых не требуется соблюдать уникальность в рамках всей глобальной сети; это практически общепринятое решение, однако оно связано с определенными ограничениями;

3) повышение гибкости «классовой» адресации за счет дополнительных средств; таким средством является *маска подсети*.

Маска подсети – битовая маска той же разрядности, что и адрес, накладываемая на него и специфицирующая «сетевую» и «хостовую» его части, в том числе вне зависимости от принадлежности к классу. Старшая часть маски состоит из подряд идущих единиц, младшая – из нулей. Та часть адреса, которой соответствуют единичные разряды маски, идентифицирует сеть, нулевые – узел. Выделенную маской сетевую часть адреса называют *префиксом*. Префикс обычно записывается в «точечной» форме, но с указанием и его длины. Например, маска 255.255.254.0 расщепляет адрес 192.168.123.123 на 23-разрядный префикс (номер сети) 192.168.122/23 и номер узла 1.123 (порядковый номер в подсети – 379). Частный случай – *обычная* (*natural*) маска, выделяющая сетевую часть адреса в соответствии с его классом.

Маска активно используется в маршрутизации пакетов IP, «маскируя» различия в адресах узлов при определении их принадлежности к той или иной подсети, чем достигаются следующие две основные цели.

1. Выделение нескольких *подсетей* внутри заданной сети: нулевые разряды маски покрывают часть идентификатора узла в адресе, остальная его часть идентифицирует подсеть в сети. Например, маска 255.255.255.224 описывает в рамках одной сети класса C до 8 подсетей с полными идентификаторами от x.x.x.32 до x.x.x.224 (с шагом 32), каждая из которых может содержать до 30 узлов с идентификаторами от 1 до 30. Этот механизм выделения описан в *RFC 950* и применяется для структурирования сетей.

2. Объединение нескольких сетей (обычно класса *C*) в *надсеть* (*суперсеть*) – «нули» маски покрывают часть идентификатора сети, которая, в свою очередь, приписывается к идентификатору узла, и все сети, идентификаторы которых различаются только «замаскированной» частью, образуют единую надсеть. Например, маска 255.255.240.0 объединяет до 16 сетей класса *C* в надсеть, которая может содержать до 4094 узлов с номерами от 0.1 до 15.254. Надсети создаются, когда возможностей доступного класса адресов недостаточно, а более высокий класс недоступен. Маршрутизация в них становится *внеклассовой* (*classless*) и использует метод *CIDR* (см. подразд. 2.3).

На практике часто сочетается использование «внутренних» адресов и масок подсетей: на локальную сеть выделяется «вырезанный» маской ограниченный диапазон полноценных «внешних» IP-адресов (например класса *C*), которые закрепляются за несколькими *пограничными* узлами, или *шлюзами* (*gateway*), внутри этой сети действуют исключительно «внутренние» адреса, а для ее структурирования на подсети снова применяется маскирование.

2.3.2. Сообщения IP

Протокол IP предусматривает обмен сообщениями, которые принято называть *пакетами,* или *датаграммами* (формат пакета см. в табл. 2.1). Напомним, что в рамках пакетного принципа передачи каждое такое сообщение доставляется независимо от остальных и, возможно, по отдельному маршруту.

Таблица 2.1

Формат пакета IP

Поле	**Размер, бит**	**Описание**
Version	4	Версия IP (здесь рассматривается версия 4)
Internet Header Length (IHL)	4	Длина заголовка в 32-разрядных словах
Type of Service (TOS)	8	Тип обслуживания (табл. 2.2)
Total Length	16	Полная длина пакета с заголовком в байтах
Identification	16	Идентификатор пакета
Flags	3	Флаги (табл. 2.3)
Fragment Offset	13	Указатель фрагмента
Time To Live (TTL)	8	Время жизни пакета (рекомендуется 64)
Protocol	8	Идентификатор вложенного протокола
Header Checksum	16	Контрольная сумма заголовка
Source Address	32	Адрес отправителя

Destination Address	32	Адрес получателя
Options	?	Опции (если есть)
Padding	0…31	Заполнитель до границы квартета

Примечание. В спецификациях сетевых протоколов разряды, поля и слова обычно нумеруются и изображаются на диаграммах в естественном порядке, то есть в том, в котором осуществляется их передача. В рассматриваемых протоколах старший разряд (*most significant bit*) передается первым, байты (октеты) также следуют в порядке убывания их старшинства в слове. Таким образом, например, поле *Version* занимает старшие 4 разряда старшего октета первого слова заголовка, и передается оно первым, начиная со старшего своего разряда. В памяти слова размещаются аналогично – стандартным «сетевым» форматом хранения считается так называемый «*big-endian*», или прямой («старший первый», старший байт по младшему адресу).

Номер версии протокола вынесен в начало заголовка, так как все дальнейшее его содержимое специфично для конкретной версии. Здесь рассматривается версия 4 протокола IP (*IPv4*).

Длина заголовка (*IHL*) составляет минимум 5 32-разрядных слов (20 байт), что имеет место при отсутствии опций и заполнителя (см. ниже). Максимальная длина – 15 слов (60 байт). Для сравнения, минимальный размер кадра Ehernet – 46 байт.

Слово типа сервиса (*TOS*) содержит требования особого, отличного от стандартного порядка обслуживания (транспортировки) пакета (табл. 2.2).

Таблица 2.2

Формат слова типа сервиса (TOS) заголовка IP

Поле	**Размер, бит**	**Описание**
PR – Precedence	3	Приоритет: 111 – сетевое управление (Network Control); 110 – межсетевое управление (Internetwork Control); 101 – CRITIC/ECP; 100 – Flash Override; 011 – Flash; 010 – немедленный (Immediate); 001 – приоритетный (Priority); 000 – нормальный маршрутизируемый (Routine).
D – Delay	1	Требования малой задержки
T – Throughput	1	Требование максимальной пропускной способности
R – Reliability	1	Требования высокой надежности
–	2	Зарезервировано (=0)

И приоритет, и флаги обслуживания не являются обязательными к исполнению и могут игнорироваться IP-модулями как оконечного узла (отправителя), так и промежуточных. Однако там, где они учитываются, необоснованное тре-

бование специального обслуживания может приводить к неэффективному расходованию ресурсов сети. *RFC 1700* содержит рекомендованные значения приоритетов и типов сервиса для некоторых видов приложений.

Общая длина пакета подсчитывается вместе с заголовком в байтах. Предельный размер пакета – 64 Кбайт – на практике не приемлем. Желательно, чтобы пакет IP целиком помещался в поле данных кадра протокола канального уровня; наибольший размер такого пакета называют *Maximum Transmission Unit* (*MTU*) – максимальный размер передаваемого блока. Спецификации требуют, чтобы любой узел обеспечивал значение MTU как минимум 576 байт; этот же размер рекомендуется для пакетов, передаваемых за пределы подсети.

Пакеты слишком большой длины могут быть разбиты (*фрагментированы*) на несколько меньших пакетов, причем прозрачно для прикладных программ. Фрагментация выполняется автоматически и зависит от характеристик сети и промежуточных узлов.

Идентификатор фрагмента служит для распознавания фрагментов, относящихся к одному исходному пакету.

Флаги описывают фрагментацию пакета: *DF* указывает IP-модулям на возможность или невозможность фрагментации данного пакета, *MF* отмечает реально произошедшую фрагментацию, то есть наличие продолжения в следующих фрагментах (табл. 2.3).

Таблица 2.3

Формат флагов фрагментации заголовка IP

Поле	**Размер, бит**	**Описание**
–	1	Зарезервировано (=0)
DF – Don't fragment	1	Запрет фрагментации пакета
MF – More fragments	1	Признак наличия последующих фрагментов пакета

Смещение фрагмента указывает на расположение данного фрагмента в пределах пакета.

Время жизни (*TTL*) пакета представляет собой счетчик, который в процессе транспортировки уменьшается на единицу каждую секунду или (в зависимости от реализации) в каждом промежуточном узле, по достижении нулевого значения счетчика пакет уничтожается (не перенаправляется дальше). Таким образом реализуется механизм «самоликвидации», или «уборки мусора», предохраняющий от засорения сети «заблудшими» пакетами. В частности, пакет с нулевым временем жизни не может выйти из подсети. Максимальное значение – 255, рекомендуемое для большинства применений – 64.

Идентификатор инкапсулированного протокола указывает, кто является «пользователем» данного пакета. Например, значение 6 соответствует протоколу TCP, 17 – UDP, 4 – IP (инкапсуляция IP в IP) и т. д.

Контрольная сумма защищает только заголовок IP-пакета. Она вычисляется обычным способом – арифметическим суммированием и дополнением (подробнее см. подразд. 3.2) всех 16-разрядных слов заголовка, однако специ-

фикация допускает использование более совершенных алгоритмов, например *CRC*. Если в процессе передачи пакета IP-модулем некоторые поля изменяются (например TTL), контрольная сумма должна быть пересчитана.

Адресá отправителя и получателя – служат идентификаторами узлов, участвующих в обмене. Допустимы все перечисленные разновидности адресов, включая имеющие специальное значение.

Опции в заголовке могут быть использованы для различных целей, например для отладки, в обычных пакетах данных они не требуются. Заголовок без опций занимает ровно пять 32-разрядных слов, но если опции присутствуют, *заполнитель* (нулевые биты) дополняет их до границы слова.

IP используется в большинстве случаев как носитель сообщений других протоколов, для которых IP-адресá включаются в схему идентификацию абонентов. Поэтому идентификатор протокола и оба áдреса из заголовка IP могут рассматриваться также и как часть заголовка инкапсулированного протокола – так называемый *псевдозаголовок* (см. разд. 3).

2.4. Базовые сведения о маршрутизации в IP-сетях

В общем случае задача *маршрутизации* может быть определена как задача построения маршрута (пути доставки данных) от известного узла-отправителя до заданного его сетевым адресом узла-получателя. Сам маршрут описывается адресами узлов, через которые он проходит, включая конечный. Эта задача решается на различных уровнях и различными средствами.

В первую очередь можно выделить два основных типа маршрутизации:

– *локальная*, или *прямая,* маршрутизация (*direct routing*) – получатель находится в одной сети с отправителем, достаточно лишь преобразовать сетевой адрес в адрес уровня MAC, но применимость ограничена «плоскими» сетями;

– *косвенная* маршрутизация (*indirect routing*) – получатель в другой сети (подсети), маршрут включает промежуточные узлы, в каждом из которых предстоит выбирать адрес следующего узла (или сетевой интерфейс).

Ввиду невозможности хранить персональный маршрут для каждого целевого адреса сеть структурируют так, чтобы для всех узлов одной подсети маршрутизация выполнялась одинаково, тогда в перенаправлениях участвуют только маршрутизаторы или шлюзы. Желательно, чтобы при этом максимум необходимых для маршрутизации сведений нес сам сетевой адрес.

В сетях IP роль таких сведений играет номер сети в адресе в соответствии с его классом или *маской подсети* (см. выше). Маршрутизатор должен по совпадению номера сети распознать принадлежность áдреса одной из подсетей и перенаправить пакет соответствующему соседнему маршрутизатору, либо опознать адресата как находящегося внутри «своей» подсети. Использование масок дает возможность выстроить иерархию сетей, на каждом уровне которой эта процедура повторяется для различной длины префикса.

Рассмотрим в качестве примера поиск маршрута к узлу с адресом 192.168.123.123 во «внутренней» сети 192.168/16, разделенной на подсети

маской 255.255.254.0 (23-разрядные префиксы). Тогда искомый адрес находится в подсети 192.168.122/23. Пусть эта подсеть также делится на подсети маской 255.255.255.224 (27-разрядные префиксы), тогда искомый адрес находится в подсети 192.168.123.96/27. Возможны и комбинации масок различной длины.

Для сетей с выделенными в них подсетями (см. *RFC 950*) действует метод маршрутизации с *масками переменной длины* (*Variable Length Subnet Masks – VLSM*). Похожим образом, но уже для объединения подсетей использует маски и метод *внеклассовой междоменной маршрутизации* (*Classless Inter-Domain Routing – CIDR*, *RFC 1517..1520*).

Для более сложных задач, включая поиск оптимальных маршрутов, существуют специальные алгоритмы и протоколы маршрутизации, подробное рассмотрение которых выходит за рамки настоящего пособия.

3. ПРОТОКОЛЫ ТРАНСПОРТНОГО УРОВНЯ TCP И UDP

3.1. Роль и место транспортных протоколов

Протокол IP, используя пакетный принцип передачи, достаточно успешно решает задачу маршрутизации и доставки пакетов данных. Однако целостность и безошибочность передаваемой информации при этом обеспечиваются недостаточно, кроме того, прямое использование протокола сетевого уровня прикладными программами может быть неудобно и даже небезопасно. Поэтому поверх IP реализуются *транспортные* протоколы, которые предоставляют приложениям услуги соответствующего качества, образуя, таким образом, завершенную *транспортную систему*. В стеке TCP/IP эти функции исполняют протоколы TCP и UDP. Как «пользователи» IP, транспортные протоколы имеют собственные *идентификаторы протоколов* (см. 2.2.3, 3.2, 3.3).

Основное назначение этих протоколов – обеспечить для вышестоящих уровней удобный интерфейс к среде передачи данных через сеть в виде соединений (виртуальных каналов) либо датаграмм. В качестве идентификатора точки доступа к транспортной системе традиционно используются *порты* – фактически номера экземпляров интерфейсов к транспортной системе. С точки же зрения сети (а также пользователей удаленных узлов) порт играет роль идентификатора конкретной программы, для которой он был создан. Каждый транспортный модуль конкретной системы выделяет номера портов из собственного пространства, в пределах которого их значения уникальны, но порты различных протоколов могут иметь совпадающие номера и оставаться независимыми. В сочетании с IP-адресом и идентификатором протокола номер порта образует законченную систему идентификации абонента сети (сетевого приложения) (см. подразд. 1.6, 1.7, 4.2).

В качестве программного интерфейса портов и далее транспортной системы служат *сокеты*. Подробнее о программировании сетевых приложений и использовании сокетов см. разд. 4 настоящего пособия.

Протоколы TCP и UDP интегрированы с IP настолько тесно, что часть заголовочной информации этих протоколов рассматривается совместно – так называемый *псевдозаголовок*, содержащий некоторые поля заголовков пакета IP и инкапсулированного в него сообщения транспортного протокола.

3.2. Протокол передачи датаграмм UDP

UDP – User Datagram Protocol
Идентификатор протокола при инкапсуляции в пакет IP – 17.
Описание – RFC 768

«Протокол пользовательских датаграмм» UDP – наиболее простой транспорт в стеке TCP/IP. Как следует из названия, он обеспечивает передачу данных *без установления соединения*, в виде завершенных самостоятельных сообщений, при этом успешность доставки, ее срок, порядок сообщений и единственность доставленных их экземпляров протоколом не гарантируются.

Формат сообщения UDP см. в табл. 3.1.

Таблица 3.1

Формат датаграммы UDP

Поле	Размер, бит	Описание
Source port	16	Порт источника сегмента
Destination port	16	Порт получателя сегмента
Length	16	Длина сообщения (включая заголовок), байт
Checksum	16	Контрольная сумма
Data	[*Length*] – 8	Данные сообщения

Примечание. Порядок следования разрядов и октетов в словах полностью аналогичен описанному ранее для протокола IP (см. подразд. 2.2.2).

Размер данных датаграммы, очевидно, не может превышать 65528 байт, но конкретные системы могут налагать и более жесткие ограничения на размер сообщений UDP (обычно несколько килобайт). В предельном случае датаграмма может быть пустой (длина сообщения 8).

Контрольная сумма вычисляется как инверсия «суммы по модулю 1» всех двухоктетных слов псевдозаголовка (см. ниже), заголовка UDP и данных датаграммы, при необходимости дополненных заполнителем 0. «Суммой по модулю 1» здесь называется следующая операция: арифметическая сумма усекается по разрядной сетке, но затем перенос из старшего разряда прибавляется к сумме как обычное число. Например, если арифметическое суммирование значений AB01h и CD02h дает 17803h (старшая 1 – перенос за пределы разрядной сетки), то «сумма по модулю 1» –7804h. Само поле контрольной суммы при вычислении её отправителем датаграммы считается нулевым. Если контрольная сумма не вычислялась, в это поле записывается значение 0; если вычисление дало значение 0, то оно заменяется на 65535 (FFFFh).

Псевдозаголовок (*pseudo header*) UDP составляют отдельные поля заголовков как сообщения UDP, так и пакета IP, в который это сообщение инкапсу-

лировано: оба IP-áдреса, идентификатор протокола (здесь это UDP) и размер сообщения UDP. Формат псевдозаголовка приведен в табл. 3.2, его общий размер составляет 12 байт, поля перечислены в порядке их следования.

Таблица 3.2

Псевдозаголовок UDP

Поле	Размер, байт	Исходное расположение
Source Address	4	Заголовок IP
Destination Address	4	Заголовок IP
Заполнитель (=0)	1	—
Protocol (=17)	1	Заголовок IP
Length	2	Заголовок UDP

UDP представляет собой простой и удобный способ передачи не очень больших объемов информации, когда требования к надежности невысоки. Полезным свойством UDP является возможность посылки датаграммы на широковещательный или групповые адресá. Подробнее об использовании UDP в сетевых приложениях см. разд. 5 настоящего пособия.

3.3. Протокол потоковой передачи TCP

TCP – Transmission Control Protocol
Идентификатор при инкапсуляции в пакет IP – 6.
Описание – RFC 793 (основное), RFC 1323 и др.

3.3.1. Общая характеристика и формат сообщений

Более сложный и обладающий более богатыми возможностями «протокол управления передачей»TCP обеспечивает надежную связь между абонентами посредством виртуального канала, используя метод передачи с установлением соединения. В связи с этим протокол имеет несколько характерных особенностей.

1. Для каждой пары взаимодействующих абонентов создается отдельное соединение. Порядок функционирования включает: запрос соединения, подтверждение соединения, обмен данными через него, затем разрыв соединения. На всех стадиях существенно участие обеих взаимодействующих сторон.

2. Так как поток данных предполагается непрерывным, а порция, передаваемая одним обращением к протоколу, конечна, поток представляется состоящим из *сегментов* – блоков данных соответствующего формата, вкладываемых в IP-пакеты.

3. Надежность передачи обеспечивается наличием подтверждений приема данных, которые отсылаются (тоже как поток) в обратном направлении – от приемника к источнику. По этой причине соединение на самом деле всегда является двунаправленным, даже если прикладной уровень использует его как однонаправленное. В случае дуплексного канала подтверждения могут совмещаться с встречными данными.

4. Потоки рассматриваются как непрерывные, а составляющие их сегменты могут подвергаться перекомпоновке, поэтому как для идентификации сегментов данных, так и для подтверждений их приема используется не номер сегмента, а позиция в потоке.

Формат сегмента TCP приведен в табл. 3.3.

Характерно, что в заголовке TCP не предусмотрено поле его длины сегмента. Реальный размер всего сегмента TCP содержится в псевдозаголовке (см. ниже). Количество данных в сегменте может быть и нулевым, тогда он целиком состоит из одного заголовка.

Таблица 3.3

Формат сегмента TCP

Поле	**Размер, бит**	**Описание**
Source port	16	Порт источника сегмента
Destination port	16	Порт получателя сегмента
Sequence number	32	Номер позиции начала данного сегмента в потоке (смещение от начала потока) в байтах
Acknowledgement number	32	Номер позиции первого байта, прием которого еще не подтвержден
Data offset	4	Смещение начала области данных сегмента от его начала, пропуск длины заголовка
Reserved	6	Зарезервировано (=0)
Flags	6	Флаги (табл. 3.4)
Window	16	Оставшийся на данный момент размер *окна приема*
Checksum	16	Контрольная сумма
Urgent pointer	16	Указатель «срочных» данных – позиция в сегменте первого «несрочного» байта данных
Options	?	Дополнительные необязательные опции
Padding	0…31	Заполнитель до границы 32-разрядного слова
Data	?	Данные сегмента

Примечание. Порядок следования разрядов, битовых полей и октетов в словах здесь и в табл. 3.4 полностью аналогичен описанному ранее для протокола IP (см. подразд. 2.2.2).

Флаги играют роль признаков специальной интерпретации отдельных полей заголовка или специальной обработки сегмента в целом (табл. 3.4). Все флаги одноразрядные, перечислены в естественном порядке их следования, то есть начиная со старшего (см. описание формата пакетов IP, п. 2.2.2).

Таблица 3.4

Флаги сегмента TCP

Флаг	**Описание**
URG	Наличие в сегменте экстренных (*urgent*) данных
ACK	Наличие в сегменте подтверждения (*acknowledgement*)

PSH	Немедленно передать («вытолкнуть» – *push*) буферизованные данные
RST	Требование закрыть канал (*reset*)
SYN	Наличие в сегменте данных синхронизации (*synchronize*)
FIN	Окончание передачи, подтверждение разрыва соединения (*final*)

Примечание. Представление слов и порядок передачи октетов полностью аналогичны описанным ранее для протокола IP (см. подразд. 2.2.2).

Контрольная сумма вычисляется аналогично UDP, но является обязательной.

Псевдозаголовок TCP также формируется аналогично UDP и имеет тот же размер – 12 байт (табл. 3.5). Поле длины TCP учитывает размеры заголовка сегмента и его данных, причем его значение *вычисляется*, а не отражает реально переданное количество байт.

Таблица 3.5

Псевдозаголовок TCP

Поле	**Размер, байт**	**Источник**
Source Address	4	Заголовок IP
Destination Address	4	Заголовок IP
Заполнитель (=0)	1	—
Protocol (=6)	1	Заголовок IP
Length	2	Вычисляется отправителем

Ниже рассматриваются некоторые аспекты функционирования TCP и применяемые при этом механизмы. Следует учесть, что обсуждаемые процедуры исполняются системными модулями поддержки протокола и прозрачны для прикладных программ, которые здесь выступают в роли пользователей.

3.2.2. Сегментация и квитирование потока данных TCP

Остановимся подробнее на сегментации потока данных и механизме подтверждений (*квитирования*) TCP. Простейший случай, когда прием и подтверждение выполняются целыми сегментами, иллюстрируется рис. 3.1 (обозначение SEQ соответствует полю *Sequence number*, ACK – *Acknowledgement number* при выставленном флаге *ACK*).

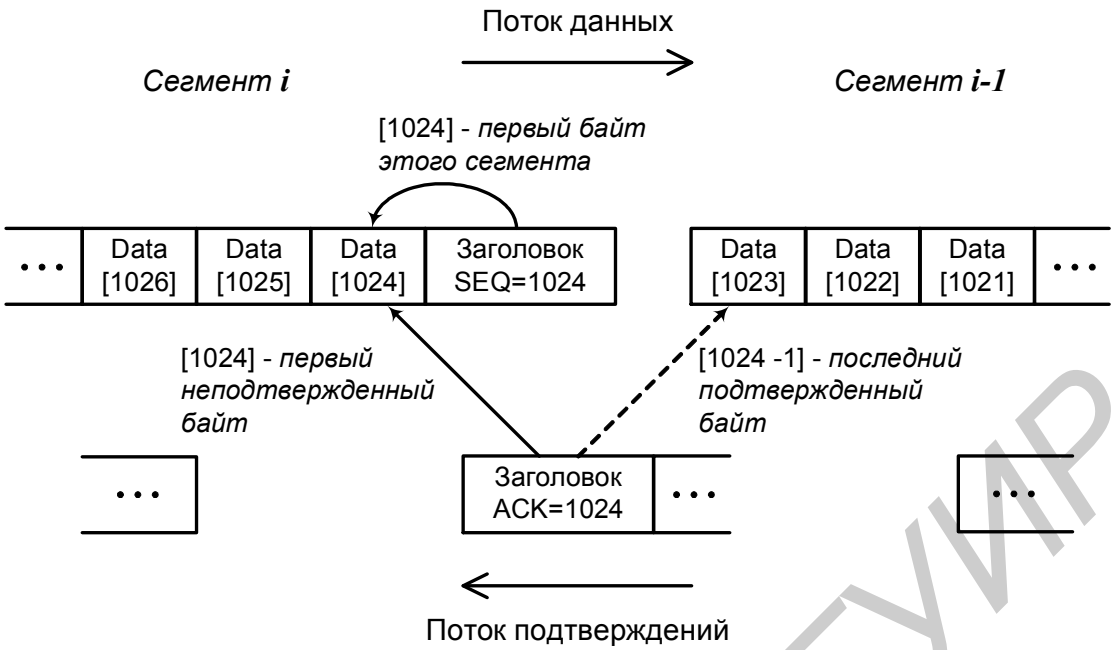

Рис. 3.1. Сегментация и квитирование потока данных TCP

Здесь предполагается, что получатель принимает сегменты целиком. Это имеет место, если он запрашивает достаточно большое количество данных, а еще лучше – согласованное с размером сегмента. На практике, однако, приложение может, например, запрашивать данные из модуля TCP и небольшими порциями, например, посимвольно. В этом случае подтверждения необязательно будут «выровнены» на сегмент. На рис. 3.2 иллюстрируется случай, когда данные запрашиваются двухбайтовыми словами.

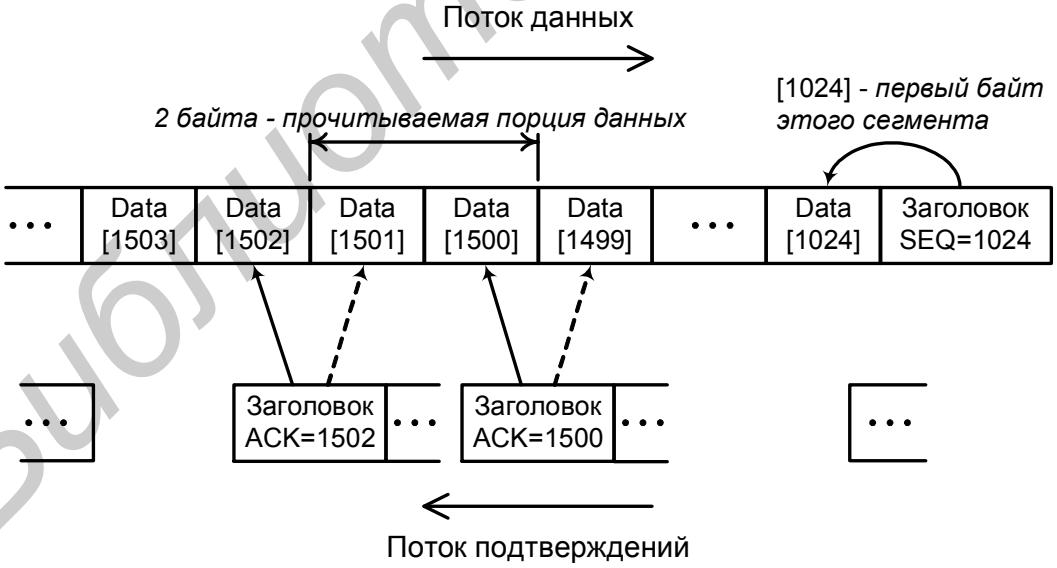

Рис. 3.2. Квитирование TCP: «частичные» подтверждения

Механизм подтверждений обеспечивает необходимый уровень надежности поддержания соединения. Но он же порождает встречный трафик в канале, увеличивает количество передаваемой служебной информации и вносит дополнительные задержки. Один из способов избежать чрезмерного роста затрат

предлагает алгоритм Нагля (*John Nagle*, *RFC 896*): задержка отсылки нескольких мелких порций данных с целью формирование из них одного сегмента бóльшей длины. Этот алгоритм реализуется модулями TCP прозрачно для прикладных программ. Применение его можно запретить, если задача не допускает задержек: флаг *PSH* в заголовке требует немедленно «вытолкнуть» накопившиеся данные, а флаг *URG* запрещает задержку части данных потока независимо от сегментации.

3.2.3. Скользящее окно TCP

И все же требование обязательного подтверждения приема переданной порции данных до передачи следующей может существенно снизить скорость обмена, особенно в глобальной разнородной сети, где задержки могут быть велики. Для преодоления этой проблемы был применен алгоритм так называемого *скользящего окна* (*sliding window*).

Окном называют количество данных, в пределах которого можно вести передачу, не дожидаясь получения подтверждений о приеме уже переданных байт. Окно называют скользящим, потому что в процессе работы соединения оно постоянно сдвигается «вдоль» потока по мере передачи данных и подтверждений и меняется в размере (рис. 3.3). Инициатором в определении размера окна выступает приемник – именно от него поступают сегменты с полем *window* в заголовке (для обоих направлений объявляются собственные окна).

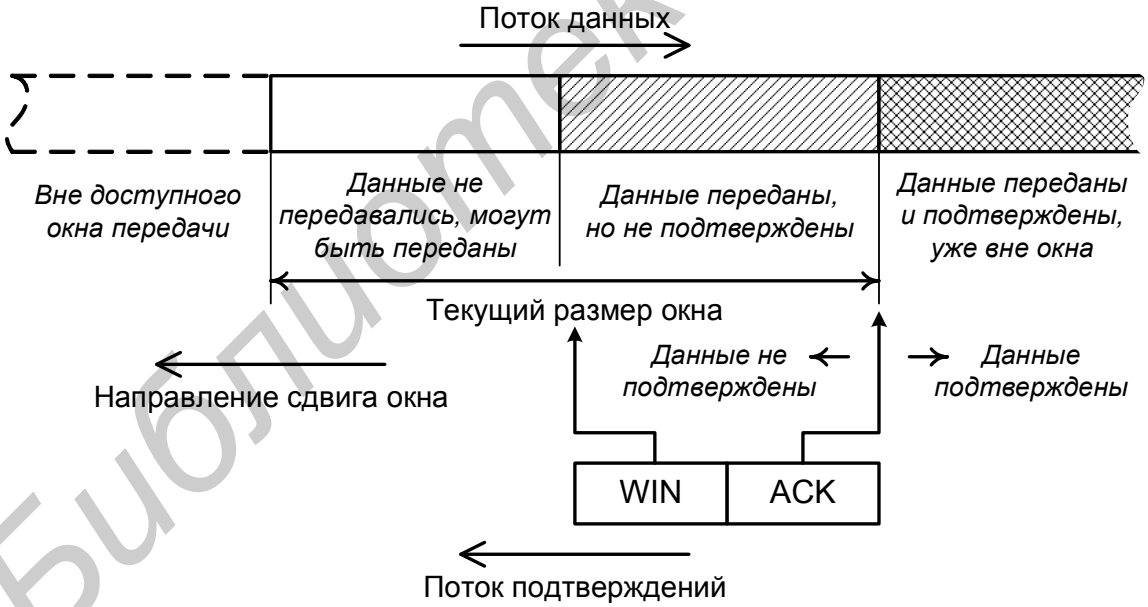

Рис. 3.3. Скользящее окно TCP

Так как приемник и передатчик не знают точных текущих значений окон друг друга, они оперируют как бы собственными его «версиями»:

– *окно приема* – количество байт, которое приемник может (или хочет) принять (например свободное место в его буфере);

– *окно передачи* – количество байт, которое на данный момент может быть передано в соединение, вычисляется передатчиком как разность размера

окна, объявленного в последний раз приемником, и количества байт, уже переданных после этого объявления.

Иными словами, приемник *предлагает* приемнику *возможное* окно приема, а передатчик *вычисляет* для себя *доступное* окно передачи. Границы окон перемещаются в случаях:

– окно приема: увеличивается при считывании прикладной программой порции данных, уменьшается при поступлении очередного сегмента;

– окно передачи: уменьшается при передаче очередного сегмента, увеличивается при поступлении подтверждения, пересчитывается в соответствии со значением поля *window* ответного сегмента.

В результате образуется обратная связь между приемником и передатчиком, позволяющая им согласовывать свои состояния. Можно заметить, что эта обратная связь может быть управляющей: уменьшая заявляемый размер окна, приемник вынуждает передатчик снизить интенсивность отсылки данных вплоть до полной остановки (нулевой размер окна), а увеличивая его, разрешает повысить темп (об использовании скользящего окна для управления процессом передачи см. также п. 3.2.5).

Порядок функционирования механизма скользящего окна при передаче потока данных иллюстрируется рис. 3.4.

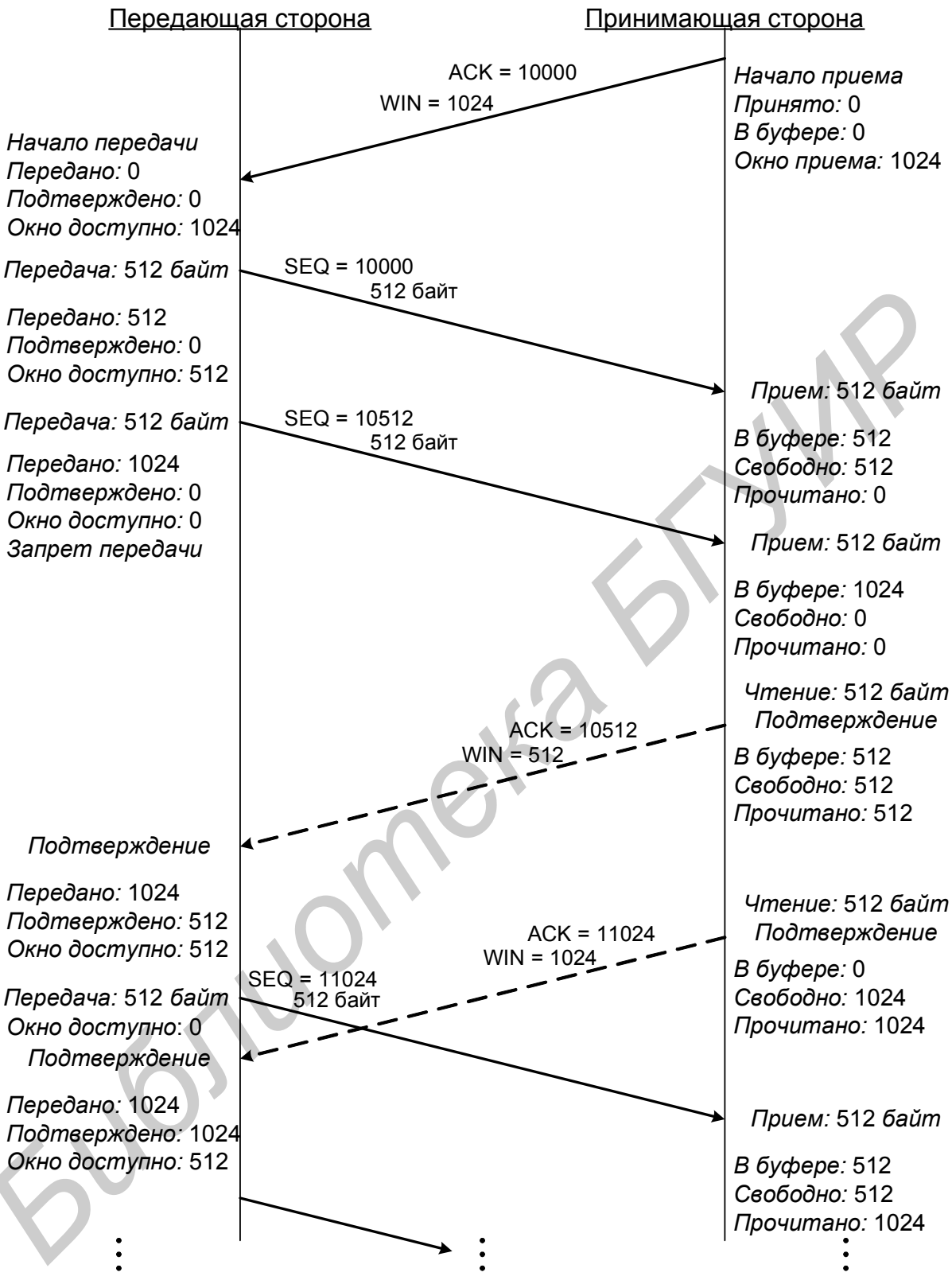

Рис. 3.4. Функционирование скользящего окна

На этом рисунке процесс обмена сильно затянут за счет длительности обработки поступивших данных на принимающей стороне, поэтому эффект от отсрочки подтверждений не слишком заметен. Однако реально заботиться о производительности соединения приходится там, где оба участника сами по себе

могут поддерживать высокую интенсивность обмена, сравнимую с пропускной способностью канала, а время доставки пакетов достаточно велико по сравнению с длительностью как передачи законченного сегмента, так и обработки его приемником. В этом случае алгоритм скользящего окна может давать очень существенный выигрыш (рис. 3.5).

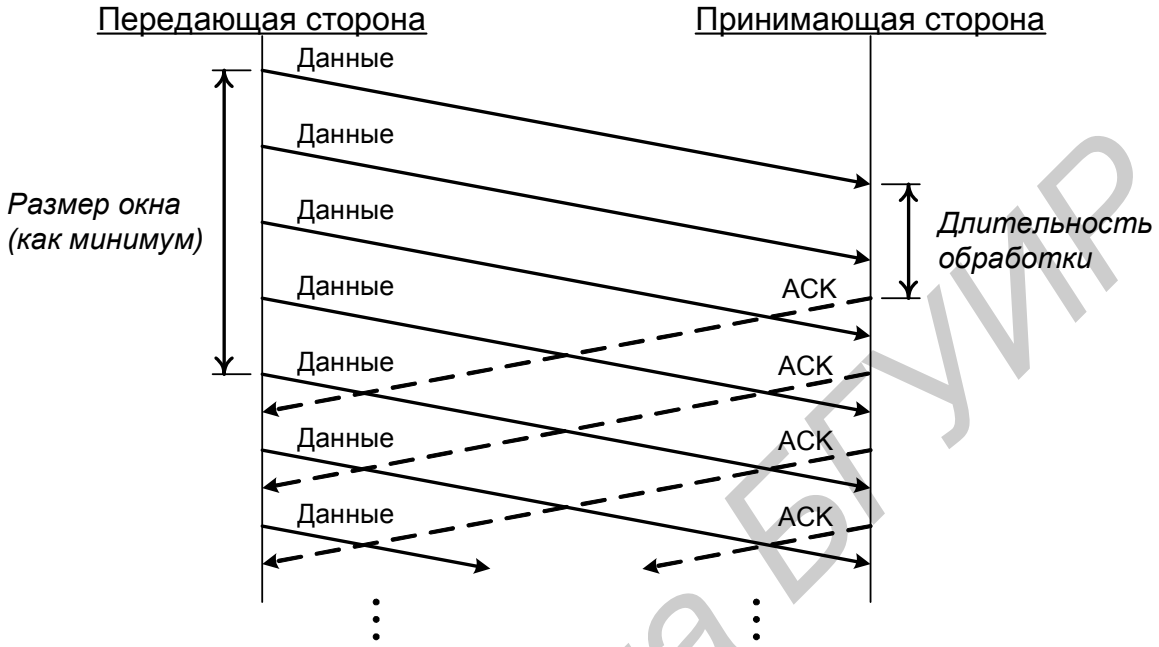

Рис. 3.5. Повышение производительности TCP с помощью скользящего окна

Таким образом, механизм скользящего окна обеспечивает согласованное взаимодействие передатчика и приемника, позволяет повысить реальную пропускную способность соединения и дает возможность дополнительного гибкого управления процессом передачи. К сожалению, при этом возникают также и некоторые побочные эффекты (см. п. 3.2.5).

3.2.4. Установление и разрыв соединений

Как уже говорилось выше, процесс создания соединения предусматривает участие обеих сторон – *клиента* и *сервера* (здесь сервером считается абонент, готовый принимать соединения, а клиентом – инициирующий создание соединения; подробнее об архитектуре «клиент-сервер» см. раздел 4). При этом модули поддержки TCP должны настроиться на взаимодействие между собой и согласовать параметры передачи, в первую очередь размер окна и стартовую позицию в потоке данных.

Размер окна приема передается соответствующим полем заголовка TCP-сегмента. Отправитель данных не может повлиять на этот параметр и должен принять его, в процессе работы соединения размер окна может быть изменен, но тоже по инициативе приемника. Для обоих направлений размер окна устанавливается индивидуально.

Стартовую позицию в потоке задает *начальный номер в последовательности* (*ISN – initial sequence number*), который также устанавливается индивиду-

ально для обоих направлений потоков. В качестве значения ISN выбирается случайное число. Это снижает вероятность ошибочного приема сегментов, оставшихся в канале после разрыва соединения и еще не уничтоженных, а также затрудняет возможному злоумышленнику «подхват» соединения и подмену сегментов (один из вариантов так называемого *спуффинга*). Несовпадение полей Sequence number вынуждает считать сегмент ошибочным и отбрасывать его (возможно, впрочем, с перезапуском соединения), а случайный выбор 32-разрядного числа обеспечивает достаточно малую вероятность случайного совпадения для одной и тоже пары абонентов.

Процедура установления соединения выполняется в три этапа и называется «тройным рукопожатием» – *three-way handshake* (рис. 3.6).

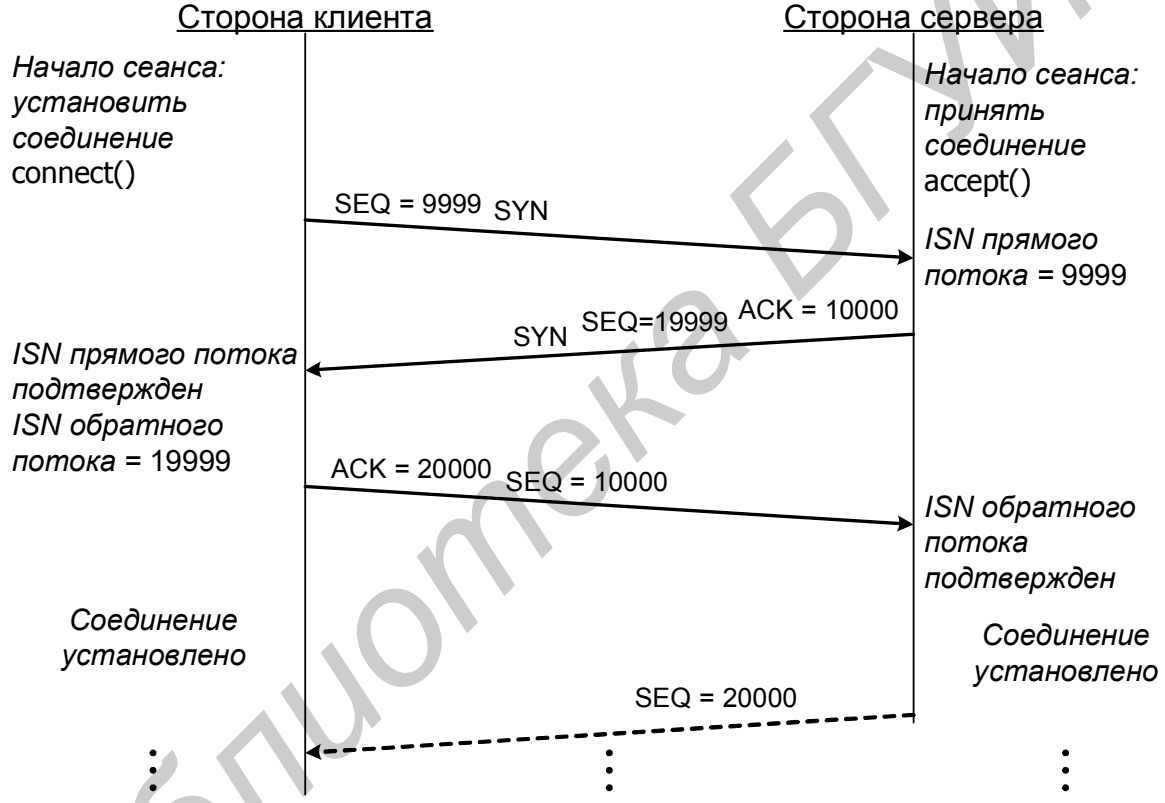

Рис. 3.6. Процедура three-way-handshake

При разрыве соединения необходимо учесть возможное наличие в канале данных, которые были переданы, но еще не прочитаны, поэтому прекращение сеанса взаимодействия тоже требует специальной процедуры. Она выполняется в четыре этапа (инициатором может быть как клиент, так и сервер):

1) инициатор разрыва (первый абонент) высылает сегмент с флагом FIN;

2) второй абонент подтверждает получение этого сегмента;

3) второй абонент повторяет подтверждение на то же количество байт, но уже с выставленным флагом FIN;

4) инициатор подтверждает получение этого сегмента.

Естественно, подтверждения высылаются после того, как сегмент будет принят, прочитан и обработан.

По завершении процедуры оба взаимодействующих TCP-модуля считают соединение закрытым и освобождают выделенные для него ресурсы. Если подтверждения не поступают в течение заданного промежутка времени, процедура закрытия продолжается по тайм-ауту.

Соединение может быть закрыто в одностороннем порядке – например, в результате обрыва связи или аварийного завершения одного из абонентов. Система обычно имеет средства для обнаружения и уничтожения «мертвых» соединений, но тайм-ауты при этом составляют несколько часов.

3.2.5. Особые ситуации TCP и побочные эффекты

Хотя механизмы подтверждений, скользящего окна и специальных флагов справляются с управлением соединением и передачей данных вполне успешно, в такой сложной и распределенной системе, какой является транспортная система TCP, неизбежно возникают проблемные ситуации, решить которые зачастую очень непросто. Ниже рассмотрены некоторые из них.

Сразу после установления соединения буферы приемника пусты, и окно приема полностью доступно. Имея много готовых к отсылке данных, передатчик должен постараться использовать это окно как можно полнее. Однако вызванный этим резкий всплеск трафика может создавать локальные перегрузки промежуточных узлов и мешать оптимизации маршрутов, особенно если они содержат «проблемные» участки. Для предотвращения дестабилизации сети служит алгоритм *медленного старта* (*slow start*), заключающийся в ограничении *передатчиком* степени использования пропускной способности: вместо предлагаемой приемником величины доступного окна он руководствуется вычисляемым *окном перегрузки*. Первоначально оно выбирается равным одному сегменту и затем удваивается после каждого получения подтверждения, пока не сравняется с окном, предлагаемым приемником. С этого момента начинается описанный ранее «стационарный» порядок управления передачей.

Затор (*congestion*) может возникать не только при старте, но и в процессе работы соединения, когда оно не успевает передавать данные. Окно приема при этом не задействуется, так как проблема возникает не у приемника, а на пути к нему. Обычно затор обнаруживается по косвенным признакам: задержкам подтверждений или их дублированию из-за повторов передачи по тайм-ауту. В этом случае передатчик должен снижать темп передачи, а затем пытаться восстановить его. Медленный старт недостаточно удобен для регулирования интенсивности вблизи состояния затора, так как предусматривает экспоненциальный ее рост. В отличие от него алгоритм *устранения затора* (*congestion avoidance*) изменяет количество передаваемых данных линейно и с учетом полученного ранее размера окна перегрузки. Для выбора между алгоритмами вводится так называемый *порог медленного старта* (*slow start treshold*), равный половине найденного размера окна передачи: при «сжатии» доступного окна не ниже этого порога следует использовать устранение затора, ниже – медленный старт (фактически означает приостановку и перезапуск).

Как уже упоминалось, признаком ошибки в соединении может служить получение дублированных или неупорядоченных подтверждений. Приемник может воспользоваться этим для *немедленного запроса* повторной передачи «испорченного» сегмента (в противном случае он просто не высылает подтверждение достаточно долго, чтобы передатчик повторил сегмент по тайм-ауту). С другой стороны, передатчик, анализируя моменты отсылки данных и получения подтверждений, может прогнозировать ожидаемое время доставки и более оптимально выбирать величины тайм-аутов.

Описанные механизмы составляют в Internet единый комплекс, обеспечивая гибкое и эффективное управление соединениями и передачей данных.

Наконец, рассмотрим случай «невыровненных» подтверждений (см. рис. 3.2) или передачи сегментов неравной длины, включая короткие. Если данные считываются из соединения небольшими порциями, возникает дилемма: подтверждать каждую из них сразу после передачи в прикладную программу или сразу несколько – с задержкой. Дальнейшие планы прикладной программы модулю TCP неизвестны, равно как неизвестны и перспективы поступления новых данных. С одной стороны, раннее подтверждение позволяет быстрее передать бóльшее количество данных, и модуль обычно считает приоритетным именно этот фактор. С другой стороны, когда у передатчика приготовлено много данных, «приоткрывание» окна даже на небольшую величину спровоцирует скорее всего посылку сегмента, использующего это окно. Но прием данных осуществляется, как правило, порциями не более сегмента, хотя бы по причине дискретности самого процесса доставки данных, поэтому та же порция будет передана и в прикладную программу, и окно снова приоткроется на небольшую величину – появление коротких сегментов становится самоподдерживающимся, и соединение непроизводительно загружается множеством пакетов с большим количеством служебной информации. Эта ситуация получила название «*синдром глупого окна*» (*silly window syndrome – SWS*). Модуль TCP не может самостоятельно выйти из состояния SWS. Для предотвращения SWS рекомендуется воздерживаться от приема и передачи коротких сообщений без особой на то необходимости (это касается и прикладных программ), а также по возможности и от слишком «поспешных» подтверждений. К счастью, описанная проблема становится актуальной только в случае интенсивного обмена, а для больших объемов данных так или иначе более эффективно именно укрупнение передаваемых порций.

4. ОСНОВЫ ПРОГРАММИРОВАНИЯ TCP/IP

4.1. Схема взаимодействия «клиент-сервер»

Большинство сетевых приложений строится по схеме «*клиент-сервер*» (рис. 4.1), которая предусматривает несимметричное взаимодействие между двумя основными участниками: клиентом и сервером.

Сервер – программа, исполняющая поступающие запросы.

Клиент – программа, отсылающая серверу запросы и принимающая результаты их обработки.

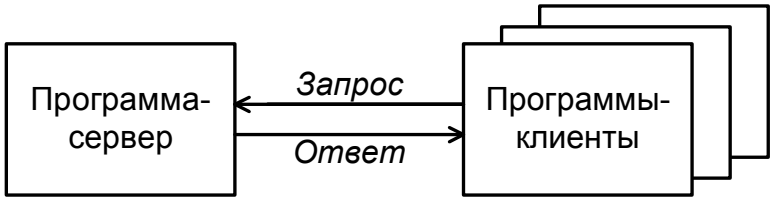

Рис. 4.1. Схема взаимодействия «клиент-сервер»

Один сервер обычно может обслуживать множество клиентов, а клиент – обращаться сразу к нескольким серверам. Одна и та же программа может играть роль и клиента, и сервера, например, обращаясь к другому серверу в процессе исполнения запросов своих клиентов.

Архитектура «клиент-сервер» является очень гибкой и имеет множество разновидностей. Здесь это понятие рассматривается в более узком смысле – применительно к функционированию сетевых протоколов различного уровня и сетевых приложений.

4.2. Основные понятия программного интерфейса TCP/IP

Как отмечалось выше, в IP-сетях принята схема идентификации любого абонента (программы) в сети, состоящая из трех элементов:

1) сетевой адрес, или адрес хоста – идентификатор конкретной ЭВМ или иного узла (точнее, конкретного сетевого интерфейса, например сетевой карты этого узла), с которым связан абонент; для сетей Internet (протокол IPv4) это 32-разрядный IP-адрес (как правило, дополнительные средства позволяют наряду с адресом использовать *сетевое имя*);

2) идентификатор протокола;

3) номер порта – идентификатор программы или иного конкретного пользователя в рамках этого узла: целое число, традиционно 16-разрядное без знака.

Сетевые порты не имеют никакого физического отображения в аппаратной архитектуре узла. Это чисто виртуальные объекты, существующие лишь постольку, поскольку их номера опознаются программными средствами. Их можно представить как индексы в таблице, которая ставит в соответствие каждому известному номеру ссылку на программный объект, использующий этот порт. Номер порта `0` считается зарезервированным (*reserved*), имеет специальное значение (см. ниже) и ни в каких соединениях не участвует. Порты с `1` по `1023` – привилегированные: они закреплены (*assigned*) за различными сетевыми службами, считающимися стандартными, или зарезервированы. По соображениям безопасности захват привилегированных портов должен регулироваться в рамках политики администрирования, обычно в зависимости от прав пользователя, от имени которого выполняется программа. (Однако, например, в ОС Windows 9x такого контроля не предусмотрено.) Подключение программ к портам в качестве клиента не ограничивается либо, если это необходимо, контролируется на другом уровне, с помощью специальных программ – *сетевых экра-*

нов, или *брэндмауэров*. Порт 1024 считается первым «пользовательским», однако и выше него имеются особые номера и диапазоны, использование которых следует считать уже определенным – так называемые «*well-known*» порты, порты более новых служб и широко распространенных приложений. Подробнее о назначении номеров портов см. *RFC 1700*.

Каждый из протоколов сетевого уровня формирует свое множество портов, то есть адресные пространства портов каждого из протоколов независимы, и значения номеров портов различных протоколов могут совпадать.

В свою очередь, сетевые адреса должны быть уникальны в рамках заданной сети (подсети), а идентификаторы протоколов являются константами, описанными в соответствующих спецификациях (см. *RFC 1700*).

Распознавание каждого из элементов выполняется программными модулями поддержки протоколов:

– номер порта – модули транспортного уровня;

– идентификатор протокола – модуль межсетевого уровня, интерфейс с транспортным уровнем;

– адрес – модуль межсетевого уровня, интерфейс с уровнем сетевого интерфейса.

Примечание. Здесь и далее обсуждаются вопросы программирования в стеке TCP/IP, поэтому используется его терминология, а в качестве сетевых адресов подразумеваются адреса́ IP.

Важно отметить, что в данном случае речь идет о *локальном* IP-адресе, присвоенном одному из доступных сетевых интерфейсов системы, и *локальных* портах, создаваемых в системе. Программа, функционирующая в этой системе, использует адрес как селектор сетевого интерфейса, через который она передает и принимает данные, и по собственному усмотрению назначает номер порта. Роль собственно идентификатора абонента-адресата эти же значения адреса и номера порта будут играть для внешних программ, выполняющихся в других системах, для которых эти адрес и порт будут *удаленными*.

Для обозначения безразличного локального адреса, например, если предполагается принимать данные от всех имеющихся сетевых интерфейсов, служит константа INADDR_ANY, численно равная нулю.

В ходе передачи потоков данных от уровня к уровню между портом и сетевым интерфейсом происходит их объединение для унифицированного обслуживания одним и тем же модулем либо, наоборот, разделение в соответствии с идентификаторами. Этот процесс называют *мультиплексированием* и *демультиплексированием*. Например, модуль транспортного уровня при передаче мультиплексирует данные от всех своих портов для передачи их модулю уровня Internet, а при приеме – демультиплексирует получаемый поток в свои порты в соответствии с их номерами в заголовках сообщений. Центральное место в механизме мультиплексирования и демультиплексирования занимает модуль уровня Internet layer, то есть в нашем случае модуль IP, а роль программного интерфейса для доступа к стеку протоколов играют *сокеты* (рис. 4.2).

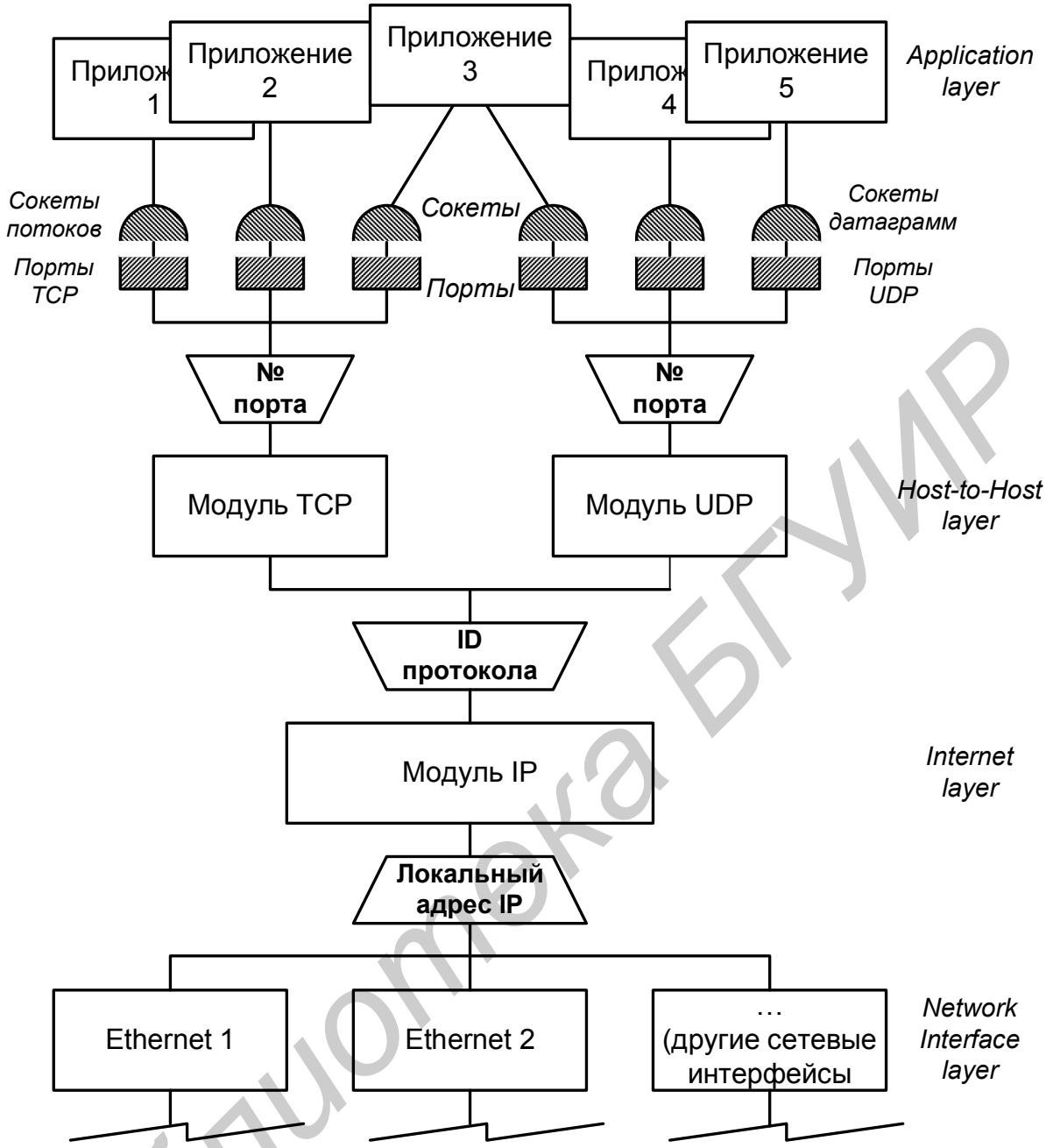

Рис. 4.2. Мультиплексирование сообщений в стеке протоколов TCP/IP

Примечание. На схеме применен ряд упрощений: показаны только те модули, которые представляют наибольший интерес в плане программирования сетевых приложений (IP, TCP и UDP), не учитываются так называемые «*сырые*» (*raw*) сокеты, не соотносимые с транспортными протоколами, и т.д.

Отметим, что мультиплексирование может происходить и на прикладном уровне – если одно приложение использует (обслуживает) несколько каналов обмена данными с сетью одновременно (например прил. 3 на рис. 4.2). Часто так же называют процесс опроса состояний нескольких точек подключения к сети с помощью вызова select() (см. подразд. 4.7).

Сокет (*socket*, букв. *гнездо*, *соединитель*) – программный объект, обычно системный, абстрагирующий точку доступа к транспортной системе. В типич-

ном случае сокет сопоставляется одному из портов одного из транспортных протоколов и служит удобной для прикладного программирования унифицированной надстройкой над ним. Реже используются сокеты, связанные с другими протоколами, либо «сырые».

Сокеты в рассматриваемом здесь виде появились в BSD Unix, в настоящее время они входят в стандарт POSIX и являются практически общепринятым интерфейсом межпроцессного взаимодействия, в первую очередь в сетях. Их поддержка в различных системах может осуществляться непосредственно ядром как часть подсистемы ввода-вывода, либо специальными библиотеками. Благодаря стандартизации, принципы программирования с использованием сокетов в различных ОС в достаточной мере подобны, хотя и имеют некоторые особенности.

Для прикладных программ сокеты обычно доступны через их *описатели* (*descriptor*, *handle*). Часто описатели сокетов совместимы с описателями файлов, что позволяет унифицировать процедуры ввода-вывода.

В целом API сокетов позволяет реализовать все задачи межпроцессного взаимодействия: идентификацию участников, синхронизацию, взаимное исключение, обмен данными с установлением соединения (посредством *виртуального канала*) и без него (*пакетная*, или *датаграммная,* передача), а также низкоуровневое (прямое) управление интерфейсом.

Далее рассматриваются сокеты Windows, предоставляемые подсистемой *Windows Sockets* (*Winsockets*). Среди функций работы с ними могут быть выделены универсальные, почти полностью соответствующие стандарту POSIX, и специфические, введенные в WinSock API. Кроме того, среды программирования, например, MFC в Visual C++, VCL в Delphi и т. д., могут предлагать собственные надстройки над Winsockets. Существуют две версии Winsockets: WinSock 1 и WinSock 2 (более точно, версии спецификаций 1.1 и 2.2 соответственно), их поддержку обеспечивают модули ОС WSOCK32.DLL и WS2_32.DLL соответственно. В среде MS Visual C++ для доступа к ним служат включаемые в проект библиотеки WSOCK32.LIB и WS2_32.LIB соответственно, а необходимые декларации находятся в заголовочных файлах WINSOCK.H и WINSOCK2.H; создавая проект, необходимо выбрать один из вариантов.

Здесь и далее, если явно не оговорено иное, рассматриваются возможности версии Winsockets 1.1, которые считаются базовыми и поддерживаются во всех Win 32-системах.

4.3. Основные типы и структуры данных API подсистемы сокетов

Сокет Win 32 в общем напоминает описатель открытого файла. Для его идентификации служит тип описателя SOCKET, совместимый с описателем общего вида HANDLE, который в свою очередь является своеобразным указателем и физически сводится к 32-разрядному целому числу. Все обращения к сокету осуществляются через этот описатель. Правила обращения с ним в общем те же, что и для других описателей Windows, однако в качестве недопустимого

или ошибочного значения вместо NULL используется специальная константа INVALID_SOCKET, а закрытие сокета выполняется специальной функцией closesocket() вместо универсальной CloseHandle().

В Unix-системах сокеты включены в пространство описателей открытых файлов и, следовательно, идентифицируются целыми числами в формате int, в современных версиях – 32-разрядными. При этом сокеты Unix во многих аспектах близки к файлам также и функционально.

Каждый сокет обладает следующими основными характеристиками.

1. *Семейство протоколов* (*protocol family*), или *семейство адресов* (*address family*), в некоторых источниках называется также *доменом* (*domain*) – идентифицирует используемые тип протокола и схему адресации. Описывается библиотечными константами с соответствующими префиксами, например PF_INET, AF_INET. В настоящее время понятия семейства адресов и семейства протоколов можно считать синонимичными. Существует множество семейств, однако поддержка их конкретными системами может быть ограниченной. Здесь интересны следующие варианты:

– PF_/AF_INET – Internet-адресация, то есть схема «IP-адрес – порт» плюс неявно заданный идентификатор протокола;

– PF_/AF_UNIX – адрес представляет собой имя файла и путь *в локальной файловой системе*; как следует из названия семейства, этот тип адреса характерен для Unix, где применяется широко;

– PF_/AF_UNSPEC – семейство не определено, автоматической обработки нет, за формирование и обработку пакетов отвечает использующая сокет программа; применяется для прямого низкоуровневого программирования.

2. *Тип сокета* – характеристика способа обработки данных, фактически неявный выбор протокола транспортного уровня, обслуживающий порт, с которым будет связан этот сокет. Необязательно каждый тип сокета совместим со всеми семействами протоколов. Ниже приведены основные типы сокетов:

– SOCK_STREAM – сокет для передачи данных с установлением соединения (*потоковый*), в сетях Internet обслуживаются протоколом TCP; в Unix такие сокеты могут использоваться в вызовах read() и write() подобно обычным файловым дескрипторам;

– SOCK_DGRAM – сокет для передачи и приема датаграмм (без установления соединения), в сетях Internet обслуживаются протоколом UDP;

– SOCK_RAW – так называемый «*сырой*» сокет, обеспечивает передачу через порт потока байт без автоматической обработки, применяется для прямого низкоуровневого программирования.

3. *Локальный* адрес сокета – идентификация точки привязки, ресурса в своей *локальной* системе (механизм мультиплексирования, см. выше), описывается специальными структурами, которые будут рассмотрены ниже.

Сочетание этих трех характеристик уникально и однозначно идентифицирует сокет и описывает его место в иерархии сетевых приложений, обеспечивая при этом гибкость и многоплатформенность интерфейса.

Многоплатформенность интерфейса сокетов потребовала также и универсальной формы представления адреса – как локального, так и удаленного (то есть адреса абонента, с которым устанавливается связь). Такой формой стали структуры: структура «абстрактного» адреса сокета sockaddr и приводимые к ней специализированные, соответствующие конкретным семействам адресов, например, sockaddr_in – адресá Internet, состоящие из IP-адреса хоста и номера порта. «Базовая» структура содержит поле идентификатора семейства адресов и зарезервированный массив байт, предполагаемый достаточным для размещения адреса любого типа. Специализированные структуры содержат то же поле идентификатора, но область данных адреса у них организована в соответствии с правилами конкретного семейства.

Для объявления реальных констант и переменных служат только специализированные структуры. Они передаются в вызываемые функции по указателю, тип которого приводится к «абстрактному», и обязательно вместе с дополнительным аргументом – размером конкретного экземпляра структуры. Если в структуре возвращается результат, например адрес абонента, то и размер также передается как указатель на переменную, в которой будет возвращен реальный размер заполненной структуры. Такой способ использования очень похож на наследование в объектно-ориентированном программировании, однако интерфейс сокетов создавался задолго до широкого признания ООП, поэтому он был реализован традиционными «до-объектными» средствами.

В качестве примера рассмотрим структуры sockaddr и sockaddr_in (все декларации взяты из заголовочных файлов MS Visual C++, имеющиеся очевидные подмены типов дополнительно не раскрываются).

```
struct sockaddr {
  u_short sa_family; //идентификатор семейства адреса
  char sa_data[14];  //резерв 14 байт для размещения адреса
};
struct sockaddr_in {
  short sin_family; //идентификатор семейства адреса, должен быть AF_INET
  u_short sin_port; //номер порта
  struct in_addr sin_addr; //Internet (IP) адрес
  char sin_zero[8]; //дополнение до 16 байт
};
```

В свою очередь сам IP-адрес также представлен иерархией структур, обеспечивающих гибкий доступ к нему:

```
struct in_addr {
  union { //различные формы представления IP-адреса:
    struct { u_char s_b1,s_b2,s_b3,s_b4; } S_un_b; //4 байта
    struct { u_short s_w1,s_w2; } S_un_w; //2 слова
    u_long S_addr; //32-разрядное целое (двойное слово)
  } S_un;
};
```

В дополнение к этому специфичному для MS Visual C++ набору полей вводится более удобное и одновременно более универсальное (соответствующее стандарту POSIX) объявление:

```
#define s_addr  S_un.S_addr
```

Таким образом, например обращение к естественной 32-разрядной форме представления IP-адреса, содержащегося в переменной *SockAddr* типа struct sockaddr_in, будет иметь вид *SockAddr*.in_addr.S_un.S_addr или *SockAddr*.in_addr.s_addr, причем второй вариант будет действителен также и в Unix-системах (напомним, что порядок байт в этом слове будет «сетевым»). Для самих структур определены синонимы SOCKADDR и SOCKADDR_IN соответственно. Применяются также и другие сокращенные обозначения (макроподставновки, см. заголовочные файлы WINSOCK.H, WINSOCK2.H).

Функции, предоставляющие информацию о хостах (интерфейсах), используют структуру hostent:

```
struct hostent {
  char * h_name; //основное («официальное») имя хоста
  char ** h_aliases; //список альтернативных имен
  short h_addrtype; //тип адресов (все адреса однотипны)
  short h_length; //размер одного адреса в байтах
  char ** h_addr_list; //список адресов хоста (интерфейса)
};
```

Списки здесь представляют собой массивы указателей на строки или структуры, признаком конца массива служит значение NULL (*null-terminated*). Несмотря на то что адреса представлены структурами, в качестве указателей используются универсальные char*, поэтому потребуется приведение типа. Список адресов используется, в частности, при разрешении имени хоста в его адрес. Для более удобного и совместимого с другими реализациями доступа к первому элементу списка адресов h_addr_list[0] служит макроподстановка h_addr.

Кроме перечисленных выше, потребуется использующаяся в библиотеке Winsockets структура WSADATA:

```
typedef struct WSAData {
  WORD wVersion;
  WORD wHighVersion;
  char szDescription[WSADESCRIPTION_LEN+1];
  char szSystemStatus[WSASYS_STATUS_LEN+1];
  unsigned short iMaxSockets;
  unsigned short iMaxUdpDg;
  char * lpVendorInfo;
} WSADATA;
```

Обычно интерес представляют поля wVersion и wHighVersion – текущая (активизированная) и наивысшая поддерживаемая системой версии Winsockets.

4.4. Основные функции API подсистемы сокетов

Здесь кратко рассматриваются наиболее часто используемые функции (системные вызовы) API подсистемы сокетов, необходимые для понимания основ сетевого программирования и написания несложных приложений. Преимущественно это функции, совместимые со стандартом POSIX; дополнительно рассмотрен минимально необходимый набор расширений, специфичный для Windows sockets (функции с префиксом WSA-). Формат вызовов не приводится, а описания функций предельно упрощены. За более подробной информацией следует обращаться к соответствующим справочникам и справочным системам.

4.4.1. Общее управление сокетом

socket() – создание сокета. Аргументы – семейство адреса, тип сокета и протокол. Если для данного типа сокета существует единственный протокол (наиболее типичный случай для обычного стека TCP/IP), то последний аргумент игнорируется. Возвращает созданный сокет, при ошибке – значение INVALID_SOCKET.

closesocket() – закрытие сокета, точнее, его описателя. Описатель становится недействительным. Если это был единственный описатель данного сокета, то вся активность на нем прекращается, ресурсы освобождаются. Если этот же сокет был открыт и другими пользователями, то для них он продолжает функционировать.

bind() – «привязка» сокета, явное связывание его с определенным *локальным* адресом, то есть адресом в «своей» системе. Аргументы – заранее созданный сокет, желаемый локальный адрес в виде структуры семейства sockaddr_*x* и размер этой структуры. Для IP-сетей используется структура sockaddr_in, содержащая номер локального порта (если 0, то он будет выбран системой автоматически) и IP-адрес локального сетевого интерфейса (если INADDR_ANY, то используются все доступные сетевые интерфейсы).

listen() – включение «прослушивания» порта, что дает возможность обнаруживать запросы на соединение. Начиная с этого момента порт, к которому был «привязан» сокет, становится доступным. Аргументы – уже созданный и «привязанный» сокет и максимальная длина очереди запросов на соединение. Запросы, помещенные в очередь, будут ожидать обслуживания, после ее заполнения очередной запрос автоматически отвергается системой. Возвращает 0 при успешном завершении или SOCKET_ERROR при ошибке. Применима к потоковым сокетам, принимающим запросы на соединения (см. ниже).

setsockopt() – настройка сокета путем присваивания или переопределения отдельных параметров (опций). Опции характеризуются уровнем, именем и значением. *Уровень* опции показывает, к какой функциональной группе они относятся (например опции потоковых сокетов). *Имя* (индекс) опции выбирает конкретную опцию. Сама опция передается в виде указателя на переменную (область памяти), где содержится значение опции или куда оно записывается в результате выполнения функции. В примерах (см. приложения) показано ис-

пользование setsockopt() для настройки сокета на работу с широковещательными адресами.

getsockopt() – получение значений отдельных опций сокета.

ioctlsocket() – низкоуровневое управление сокетом. Правила и формат вызова соответствуют принятым для API драйверов. Необходимость ее использования возникает достаточно редко.

4.4.2. Функции проверки состояния и информационные

select() – проверка состояния сокетов. В частности, позволяет избежать блокирования операций приема. Более подробно будет рассмотрена ниже.

gethostname() – получение сетевого имени хоста, на котором выполняется программа, в виде текстовой ASCIIZ строки.

getsockname() – получение собственного локального адреса, с которым связан сокет, в виде структуры семейства sockaddr_*x*.

getpeername() – получение адреса партнера по соединению в виде структуры семейства sockaddr_*x*.

gethostbyname() и gethostbyaddr() – получение адреса и некоторой другой информации о хосте по его сетевому имени либо адресу соответственно. Функции заполняют поля структуры HOSTENT и возвращает указатель на нее. Этот экземпляр структуры модифицировать нельзя, а необходимую для дальнейшего использования информацию из него надо немедленно скопировать. Выполнение функции бывает длительным, так как оно может требовать обращений к другим хостам сети (например, gethostbyname() фактически осуществляет обращение к системе *DNS* для разрешения имени).

4.4.3. Функции управления сеансом

accept() – прием из очереди очередного запроса на соединение, установление виртуального канала и подтверждение соединения. Аргумент – «слушающий» сокет. В случае успеха возвращает новый созданный и «привязанный» сокет для обслуживания этого соединения, в случае ошибки – значение INVALID_SOCKET. Функция блокирующая – если очередь пуста, ожидает поступления запроса. Используется на стороне сервера для потоковых сокетов.

connect() – запрос на соединение. Аргументы – заранее созданный сокет и полный адрес (структура sockaddr и ее реальный размер) сервера, с которым нужно установить соединение. При успехе создается соединение с сервером (виртуальный канал), с которым будет связан этот сокет, и возвращает 0; при ошибке возвращает значение SOCKET_ERROR. Если сокет не был «привязан» заранее, выполняется также привязка его на выбираемый системой локальный адрес. Функция блокирующая: если запрос был поставлен в очередь сервера, она будет ожидать завершения его обслуживания. Используется программами-клиентами, как правило, для потоковых сокетов, но применима и для датаграммных, в этом случае выполняется только неявная привязка без создания соединения (сеанса).

shutdown() – выборочное прекращение функционирования сокета без его разрушения: на прием, на передачу или в обоих направлениях, при этом освобождаются очереди данных соответствующего направления. В отличие от closesocket() результат скажется на всех пользователях сокета. Считается, что shutdown() закрывает соединение более корректно, полезна она также, например, для соединений, работающих только на передачу – при попытке чтения из такого сокета будет распознана ошибка, и блокировка не произойдет. С другой стороны, в некоторых реализациях WinSockets вместо такого избирательного закрытия соединение может разрушиться полностью.

4.4.4. Функции обмена данными

send() – посылка данных через потоковый сокет при установленном соединении.

sendto() – посылка данных с явным указанием адресата в структуре sockaddr_*x*. Функция обычно применяется для передачи без установления соединения. Если сокет потоковый и соединение уже установлено, то данные отсылаются через него, а адрес, указанный при вызове функции, игнорируется.

recv() – прием данных из сокета. Обычно используется для потоковых сокетов с установленным соединением. Для пакетных сокетов функция также работоспособна, но адрес отправителя останется неизвестным.

recvfrom() – прием данных из сокета вместе с адресом отправителя (в структуре sockaddr_*x*). Обычно используется для датаграммных сокетов.

4.4.5. Функции преобразования значений

htons(), htonl(), ntohs(), ntohl() – преобразование между «локальным» (***h***ost) платформо-зависимым и стандартным «сетевым» (***n***etwork) форматами для «коротких» (***s***hort) 16-разрядных и «длинных» (***l***ong) 32-разрядных целых чисел соответственно. Напомним, что стандартный для сетей порядок байт – «*big-endian*» («старший первый»), а локальный для Intel x86 – «*little-endian*» («младший первый», или «младший байт по младшему адресу»).

inet_ntoa() – получение текстовой строки с «точечной» пооктетной записью IP-адреса из его естественной 32-битной формы.

inet_aton() – получение IP-адреса из его строковой «точечной» записи. К сожалению, эта и ряд других стандартных для работы с сокетами функций преобразования в Winsockets не реализованы.

inet_addr() – аналогичное преобразование, но IP-адрес получается в формате 32-битного числа. В случае ошибки функция возвращает значение INADDR_NONE, численно равное -1 и не отличимое от INADDR_BROADCAST – результата преобразования широковещательного IP-адреса 255.255.255.255.

4.4.6. Расширения WinSockets

WSAStartup() – инициализация WinSockets. В качестве аргументов передаются требуемый номер версии WinSockets (согласно MSDN, старший байт – младший номер, младший байт – старший номер) и адрес структуры WSADATA, которая будет заполнена этой функцией. До успешного выполнения этой функции подсистема сокетов в Win 32 не действует.

WSACleanup() – деинициализация WinSockets, должна вызываться перед завершением приложения.

WSAGetLastError() – код последней ошибки, произошедшей при обращениях к Winsockets. Получаемое значение корректно только для синхронных операций.

Прочие WSA-функции, как аналоги перечисленных выше межплатформенных вызовов, так и имеющие расширенную функциональность, здесь не рассматриваются. Большинство из них реализовано в Winsockets 2.2.

4.5. Взаимодействие без установления соединения

Этот вид взаимодействия предусматривает обмен датаграммами, характеризуется минимальным уровнем сервиса со стороны системы и минимальными служебными затратами. Целостность передаваемых данных системой не гарантируются, при необходимости о ней должны заботиться сами взаимодействующие программы. Используемый тип сокета – SOCK_DGRAM, которому в IP-сетях соответствует протокол UDP. Порядок взаимодействия схематично показан на рис. 4.3.

Полезной особенностью большинства протоколов датаграммной передачи, в том числе и UDP, является *широковещание* – доставка соответствующим образом адресованного сообщения всем доступным узлам сети (подсети). Например, посредством широковещательного запроса можно обнаружить сервер, адрес которого не известен (очевидно, порт программы-сервера тем не менее знать необходимо), для чего ряд служб, ориентированных на использование TCP-соединений, поддерживают также и интерфейс UDP. (Отметим, что широковещание действует не всегда: оно может не поддерживаться, быть запрещено административно или отключаться при отсутствии реального соединения с сетью, когда реально функционирует только интерфейс *localhost*.)

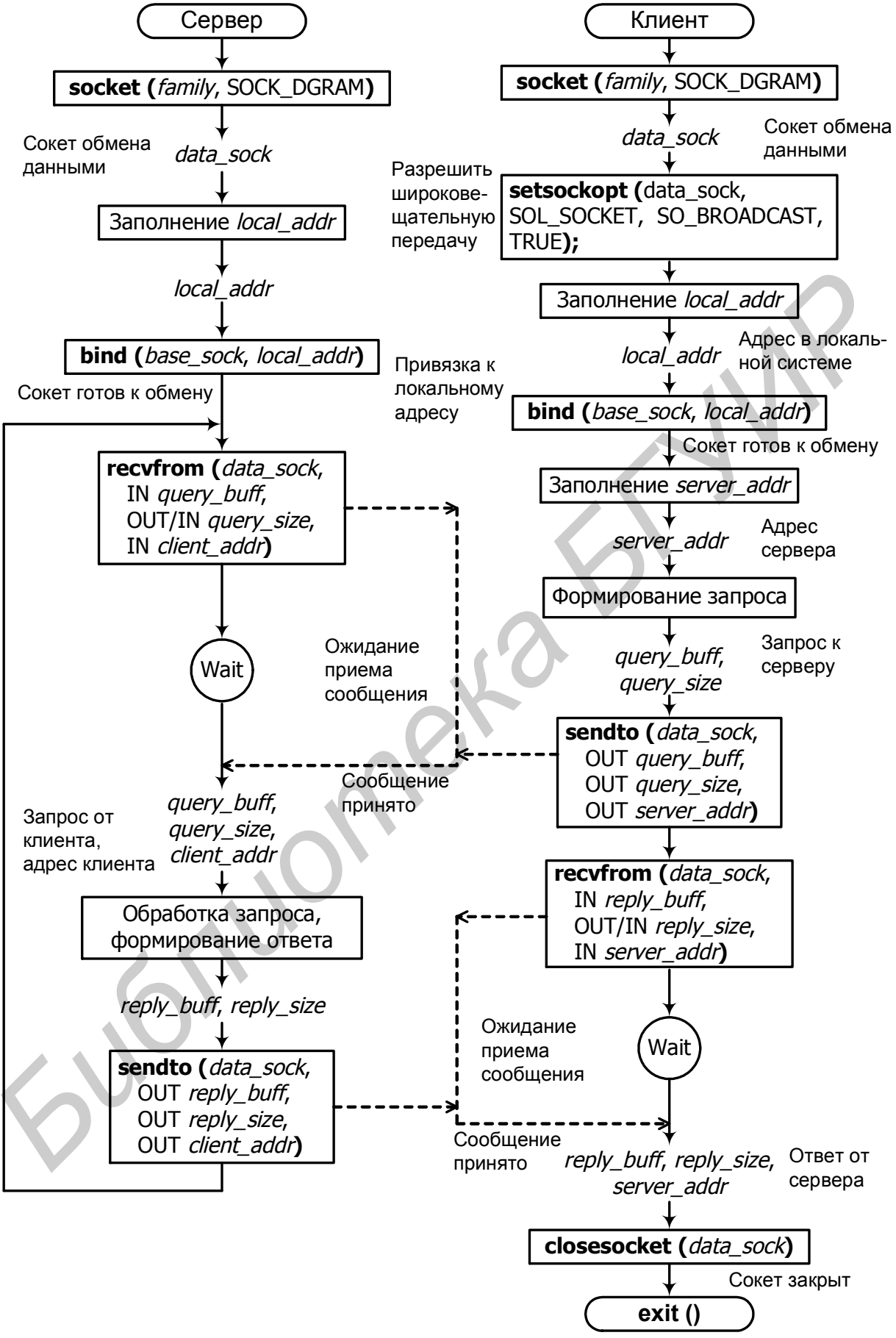

Рис. 4.3. Взаимодействие без установления соединения

Примечание. На схеме имена системных функций сохранены, но формат их вызова упрощен для повышения наглядности. Аргументы, описывающие передаваемые и принимаемые сообщения, снабжены пометками, являются ли соответствующие объекты параметрами вызова или создаются (заполняются) в результате его выполнения.

В показанном примере действия клиента и сервера в общем схожи, и с точки зрения программирования сокетов отличия между ними сугубо номинальные. Каждый из них использует всего один сокет, служащий и для приема, и для передачи сообщений. Ради упрощения считаем, что клиент отсылает серверу единственной запрос (*query*) и ожидает ответ на него (*reply*), а сервер в цикле принимает запросы, обрабатывает их и отсылает ответы, используя «обратные адреса» запросов. Цикл сервера показан бесконечным; на практике для управления серверами служат команды, предусмотренные в протоколе обмена, или какие-либо дополнительные средства.

Характерным примером сетевой службы с подобным порядком взаимодействия служит *echo*: «эхо»-сервер принимает все запросы от всех клиентов и немедленно возвращает их отправителям. Простейшая реализация «эхо»-сервера и обращающегося к нему клиента приведена в прил. 1. Обе программы консольные, необходимые параметры передаются посредством командной строки. Сервер позволяет указывать порт для приема сообщений (по умолчанию – 7 согласно *RFC 1700*). Клиент обязательно требует указания адреса или сетевого имени сервера и порта, прочие аргументы рассматриваются как сообщения, каждое из которых клиент передает серверу и затем ожидает от него ответа. Порт локального адреса клиента выбирается автоматически.

Важно отметить, что некоторые действия в программах являются блокирующими, то есть приостанавливают выполнение на неопределенный срок. В первую очередь, это ожидание поступления сообщения (функция recvfrom()). Подобная ситуация очень характерна для сетевых приложений, причем она усугубляется неопределенностью состояния партнера и каналов связи. В результате, например, при отсутствии сервера может произойти как распознавание ошибки приема, так и переход клиента в состояние бесконечного ожидания (в ходе экспериментов этот эффект наблюдался только при запросах на широковещательный адрес), соответствие пар запрос-ответ может нарушаться, если возвращается более одного ответа, и т. д. Для реальных приложений такие дефекты обычно недопустимы. Более подробно эта проблема и методы ее решения рассматриваются в разд. 4.7.

4.6. Взаимодействие с установлением соединения

Наличие постоянного соединения (виртуального канала) усложняет порядок взаимодействия, но зато обеспечивает более развитый сервис, в первую очередь контроль целостности сеанса и передаваемых данных. Используемый тип сокета – SOCK_STREAM, которому в IP-сетях соответствует протокол TCP. Порядок взаимодействия схематично показан на рис. 4.4.

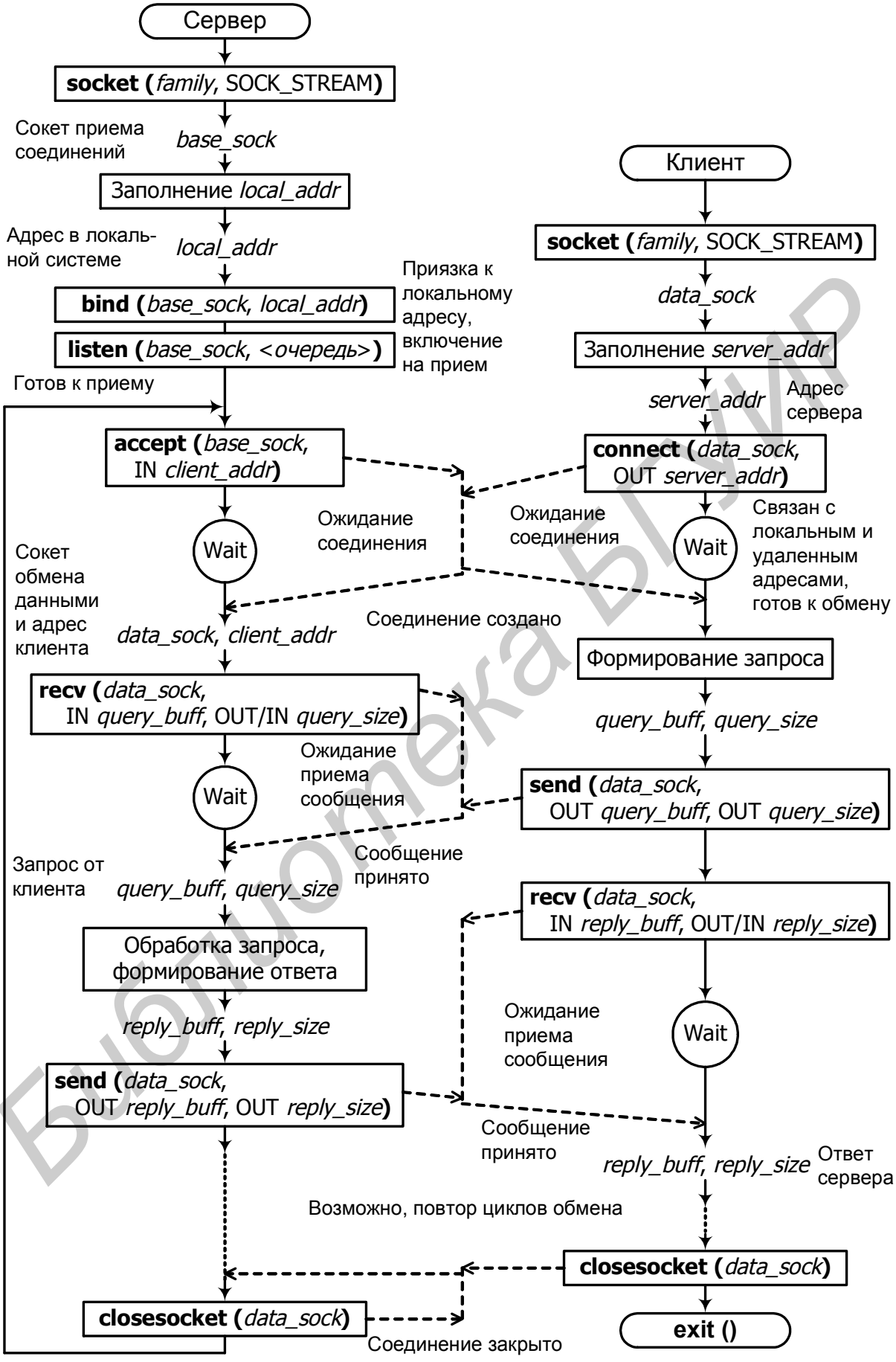

Рис. 4.4. Взаимодействие с установлением соединения

В начале работы сервер создает и связывает с локальным адресом единственный сокет (назовем его *базовый сокет*), на котором он будет производить прием запросов на соединение. Именно порт, с которым он связан, указывается в качестве порта, занятого данным сервером. Далее базовый сокет переводится в режим «прослушивания», и, начиная с этого момента, в системе организуется очередь, принимающая запросы на соединение. Основной цикл сервера состоит в приеме этих запросов вызовом accept(). При поступлении такого запроса системой создается новое соединение, и вызов accept() на стороне сервера успешно завершается, возвращая новый сокет, связанный с этим соединением – *сокет данных*, уже готовый к использованию. Адрес для сокета данных выбирается системой автоматически, и он может совпадать с адресом базового сокета.

После этого сервер начинает вложенный цикл обмена данными (сообщениями) через новое соединение, чаще всего это ожидание входящего запроса, его обработка и отсылка ответа клиенту. В отличие от UDP при передаче данных адрес назначения указывать уже не нужно, поэтому используется более простой вызов send(). Признаком завершения в простейшем случае может служить прием одной или нескольких порций данных нулевой длины. В протоколе прикладного уровня может быть также предусмотрены извещения от клиента о прекращении обмена или возможность разрыва соединения по инициативе сервера. После окончания вложенного цикла сервер может вернуться к приему следующего соединения.

Со своей стороны клиент создает сокет и передает ее в вызов connect(), указывая также и адрес сервера, с которым нужно установить соединение – IP-адрес хоста, где находится сервер, и закрепленный за ним порт. Делать предварительную привязку этого сокета к локальному адресу необязательно – в этом случае система выполнит ее автоматически. После успешного завершения вызова connect() сокет будет связан с созданным соединением и готов к работе, клиент может начинать цикл обмена данными с сервером.

Блокирующими вызовами здесь помимо recv() являются accept() и connect(). Вызов accept() фактически принимает запрос на соединение из очереди, и если очередь пуста, то он остается в состоянии ожидания. Вызов connect() блокируется, если сгенерированный им запрос на соединение не обслужен сервером, но поставлен в очередь. Если в очереди сервера свободных мест нет, то запрос отвергается немедленно и соответствующий connect() клиента завершается как неуспешный. Кроме того, во время обслуживания клиентского соединения сервер с описанным алгоритмом приостанавливает прием новых соединений (отметим, что по этой причине здесь не имеет смысла резервировать большую очередь для запросов). Размер очереди задается при «включении» базового сокета вызовом listen().

Определенной проблемой является контроль целостности TCP-соединения. Алгоритмы TCP таковы, что прекращение связи обнаруживается только при обмене сегментами (система сама также может посылать специальные сегменты для проверки «живости» соединения). Корректный разрыв со-

единения также должен сопровождаться специальными флагами в заголовке сегмента (см. подразд. 3.3). Поэтому полная потеря связи, аварийное (без соблюдения процедуры разрыва) завершение или просто неактивность программы на другом конце соединения без специальных мер обычно не отличимы друг от друга. Вариантом решения этой проблемы может быть введение в прикладной протокол обмена (поверх TCP) обязательных периодических «пустых» посылок или специальных «зондирующих» сообщений в сочетании с использованием тайм-аутов.

Более подробно с распознаванием нестандартных ситуаций и другими аспектами TCP/IP программирования можно ознакомиться в литературе, например [11, 12].

В качестве примера в прил. 2 приводится TCP-вариант «эхо»-сервера (служба *echo* предусматривает поддержку обоих протоколов: TCP и UDP) и простейший демонстрационный TCP-клиент. Оба приложения близки примеру. Сервер создает базовый сокет и в цикле выполняет на нем прием соединений (функция accept()). Для каждого возвращенного ею сокета данных, соответствующего вновь созданному соединению, выполняется вложенный цикл, состоящий в приеме (recv()) порции данных из этого соединения и немедленной отсылке (send()) ее обратно. Признаком завершения цикла здесь служит получение сообщения нулевой длины (прием пустого сегмента), после чего сокет данных закрывается, и повторяется итерация внешнего цикла (accept()). Клиент устанавливает соединение с сервером (функция connect()), передает (send()) через это соединение каждую переданную в качестве аргумента строку, всякий раз дожидаясь приема ответа (recv()), после чего разрывает соединение (shutdown(), closesocket()).

Напомним, что в качестве универсального TCP-клиента может служить программа *telnet*, различные варианты либо аналоги которой присутствуют в большинстве современных систем.

Оба примера (прил. 1, 2) работоспособны, но максимально упрощены для сокращения объема и повышения наглядности исходного текста. Реальные приложения должны выполняться в более «правильном» стиле: структурированными, с выделением подпрограмм, развернутой обработкой ошибок и т. д. Но главное – на практике, как правило, необходимо учитывать вероятные длительные блокировки программы в состоянии ожидания и обеспечивать возможность работы сервера более чем с одним клиентом.

4.7. Элементы параллелизма в сетевых приложениях

4.7.1. Постановка проблемы

В рассмотренных выше примерах серверы способны взаимодействовать с несколькими клиентами, но только поочередно, то есть новое соединение не создается ранее закрытия текущего, новый запрос не принимается ранее окончания обработки текущего. Такой тип сервера называется *последовательным*, для него характерны простота и минимальные затраты ресурсов на служебные

функции, однако невозможность работы более чем с одним клиентом одновременно остается принципиальным ограничением. Следовательно, последовательный сервер применим в случаях, когда заведомо не требуется обслуживать более чем одного клиента или интенсивность запросов очень мала, или запросы можно выполнять очень быстро, причем соединение (если используется TCP) затем немедленно разрывается самим сервером (примером последнего может служить служба *daytime*, см. *RFC 867*).

Однако большинство серверов должно иметь дело со многими клиентами, присылающими, возможно, очень большой поток запросов. При этом помимо эффективного распределения ресурсов сервера (что является темой отдельного обсуждения), необходимо решить проблему совмещения во времени ряда операций:

– создание новых соединений;
– прием данных (запросов) от каждого из клиентов;
– обработка каждого из принятых запросов;
– отсылка ответов на запросы;
– интерфейсные функции программы-сервера: прием и обработка сообщений, ввод с консоли и т. п.;
– другие, не связанные с сетью обращения к системе.

Таким образом, выполнение одновременно нескольких действий может понадобиться даже при обслуживании сервером единственного клиента. Большинство этих действий так или иначе связано с вводом-выводом и, следовательно, сопровождаются переходом в состояние ожидания, которое в случае простого последовательного сервера блокирует прочую активность. В частности, при обращении к сокетам, в первую очередь за получением некоторых данных, причинами неопределенно длительного ожидания могут стать:

– задержки передачи через сеть, особенно глобальную;
– особенности функционирования модулей транспортной системы;
– асинхронность и случайный характер происходящих событий;
– непредсказуемые ошибки;
– неопределенность состояния партнера.

В программе все они проявятся как ожидание завершения некоторых функций, основными из которых будут:

– accept() – установление соединений, «внешний» цикл TCP-сервера;
– connect() – установление соединения на стороне TCP-клиента;
– recv() и recvfrom() – прием данных, как TCP, так и UDP;
– send() и sendto() – передача данных (как источник блокировки эти функции рассматриваются гораздо реже).

Легко обнаружить, что именно ожидание будет составлять значительную долю расходуемого сервером времени, а благодаря асинхронности событий, можно совместить ожидание, связанное с одними клиентами, с обслуживанием активности других. Следовательно, сервер сможет поддерживать общение более чем с одним клиентом так, что время отклика для каждого из них будет ос-

таваться в приемлемых пределах – конечно, если сервер не перегружен запросами. Такой сервер будем называть *многопользовательским* (нужно помнить, что для большинства сетевых служб естественен именно такой режим).

Ниже рассмотрены основные варианты многопользовательских серверов.

Примечание. Большая часть материала относится к случаю TCP-сервера как наиболее характерному (обычно необходимо поддерживать несколько клиентских соединений). Однако актуальность его сохраняется и для UDP-сервера (выполнение длительных запросов на фоне приема следующих) и даже для клиентских программ (интерфейс с сетью и одновременно с пользователем).

4.7.2. Использование системной многозадачности

Первым и наиболее очевидным решением является вынесение цепочек логически связанных потенциально блокирующих операций в отдельные ветви алгоритма, выполняющиеся независимо от остальных. Реально такими цепочками могут стать: цикл приема соединений; циклы обслуживания соединений; выполнение отдельных запросов; при необходимости – интерфейсные функции. Упрощенная схема такого сервера показана на рис. 4.5.

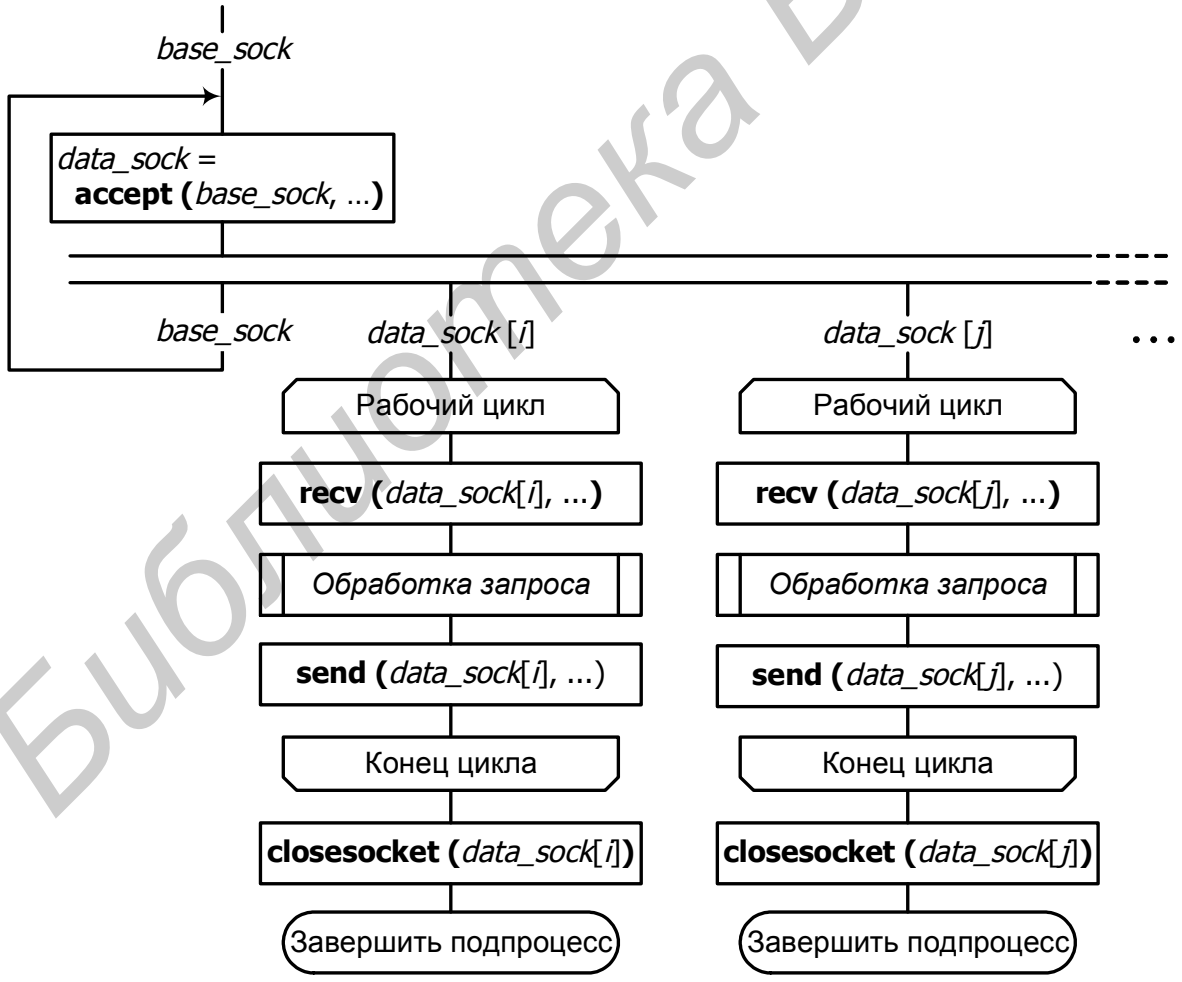

Рис. 4.5. Сервер с использованием системной многозадачности

Здесь «главный» сервер, как и ранее, принимает запросы на соединение, после чего порождает новый экземпляр «исполнительного» сервера, который будет заниматься обслуживанием этого соединения, а сам возвращается к ожиданию следующего соединения. Все порожденные экземпляры функционируют и завершаются независимо друг от друга. Таким образом, за распараллеливание операций отвечают системные средства обеспечения многозадачности, присутствующие в большинстве современных ОС.

На рис. 4.5 эти экземпляры названы «подпроцессами», на практике их роль в зависимости от особенностей конкретной системы могут играть как собственно *процессы* (*process*), так и *потоки* (*thread*). Первое более «традиционно», особенно в Unix-системах, но под Windows чаще используют именно потоки, отчасти из-за их меньшей ресурсоемкости, отчасти потому, что в Windows нет такого простого и эффективного способа создания нового процесса, как «клонирующий» вызов fork(). В Win 32 для создания процессов и потоков служат вызовы CreateProcess() и CreateThread() соответственно.

К достоинствам такого подхода можно отнести относительную простоту и «прозрачность» распараллеливания, хорошую масштабируемость при выполнении на параллельной (многопроцессорной) ЭВМ. Недостатками же являются значительные затраты на поддержание процессов (потоков) и сложности при их взаимодействии между собой. При достаточно большом количестве (обычно несколько сот) одновременно выполняющихся процессов (потоков) управляющие ими механизмы ОС перегружаются: начинается «пробуксовка» планировщика, возрастают затраты времени на переключение и его потери из-за конфликтов. Задача синхронизации в программах с элементами параллелизма и особенно отладка таких программ также очень сложны и к тому же не всегда имеют однозначное эффективное решение.

Основное различие между потоком и процессом заключается в том, что все потоки одного процесса выполняются в общем адресном пространстве и могут использовать общие глобальные переменные, что упрощает взаимодействие между ними (но не снимает проблему синхронизации!). Потоки потенциально менее ресурсоемки, хотя реализация их в различных конкретных системах может сильно различаться. Например, в классических Unix-системах потоки отсутствовали, а сейчас в некоторых из них потоки являются фактически особой разновидностью процессов. ОС семейства Win 32 изначально многопоточные, более того, планировщик в них *всегда* работает именно с потоками, а не с процессами, поэтому использование потоков в Win 32 оказывается более эффективным и используется чаще. С другой стороны, процессы менее зависимы друг от друга, и зависание или аварийное завершение одного из них (не главного) скорее всего не повлияет на остальных, поэтому многопроцессный вариант в целом более надежен и устойчив. С точки зрения программирования, отличие будет также и в том, что поток порождается внутри процесса из одной из его подпрограмм, а процесс – из внешнего исполняемого файла либо путем клонирования исходного родительского процесса.

(Более подробно мультипрограммирование, возникающие при этом проблемы и методы и средства их решения рассматриваются в дисциплинах «Системное программирование», «Операционные системы и среды».)

Так как особенно много ресурсов расходуется на само порождение процесса (потока), то естественно желание избавиться хотя бы от них. В результате получается разновидность сервера с *предварительно порожденными* (*pre-forked*, *pre-created*) процессами (потоками), выполняющими функции «исполнительных» серверов. Изначально все они находятся в приостановленном (ждущем) состоянии. При появлении запроса главный цикл активирует одного из них для обслуживания клиента, а после завершения обработки «исполнительный» сервер снова становится неактивным. Их количество выбирается заранее с таким расчетом, чтобы иметь возможность удовлетворять достаточно много запросов, но и не перегружать систему. Переключение «исполнительных» серверов может выполняться и иначе: вместо главного цикла все одинаковые исполнительные серверы сами обращаются за получением запроса, переходя в состояние ожидания, и активизация одного из них выполняется уже непосредственно системой. С той же целью можно (и обычно более предпочтительно) использовать семафоры или другие *объекты ожидания*.

В целом многозадачный вариант для сетевых приложений наиболее эффективен в случаях, когда число одновременно обслуживаемых клиентов ограничено, интенсивность установления новых соединений и поступления запросов невелики, сами запросы достаточно сложны и требуют длительной обработки, а взаимодействие между «исполнительными» сервера минимально (то есть синхронизация требуется как можно реже).

В качестве примера в прил. 3 приведена программа – UDP-клиент, отдаленно напоминающий *telnet*: ввод текстовых сообщений, отсылка их на указанный сервер и отображение ответов. Программа двухпоточная: в главном потоке выполняется ввод текстовых строк и их отсылка, а прием ответов (блокирующий, с ожиданием) и их отображение вынесены в отдельный поток. Этим устраняются недостатки программы, описанной в подразд. 4.5, а типичные проблемы многопоточных приложений при данном алгоритме работы проявиться практически не могут (хотя в параметрах проекта следует все-таки выбирать многопоточную версию стандартных подключаемых библиотек).

4.7.3. Мультиплексирование запросов

Другим достаточно естественным способом избежать блокировки является предварительная проверка состояния объектов перед обращением к ним с потенциально блокирующими вызовами. Вызов выполняется только тогда, когда он сможет завершиться без ожидания, в этом случае все проверки и рабочие вызовы могут выполняться последовательно, не задействуя «настоящую» многозадачность, – принято говорить о *мультиплексировании* запросов в едином процессе или потоке (не путать с мультиплексированием сообщений протоколов в стеке модулей их поддержки!).

Упрощенная схема TCP-сервера, использующего этот подход, изображена на рис. 4.6.

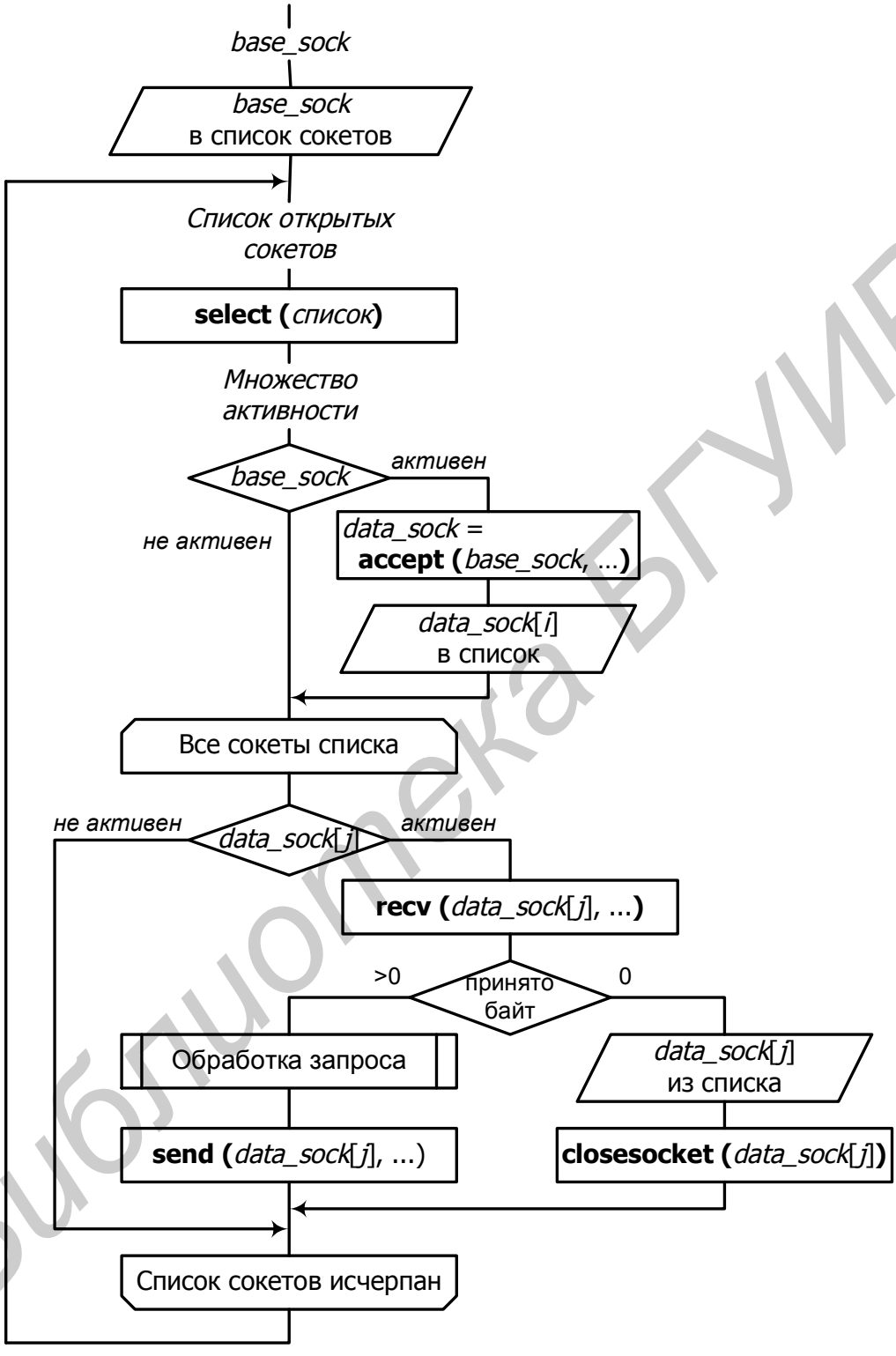

Рис. 4.6. Сервер с мультиплексированием запросов

Использование такого подхода требует возможности надежно и эффективно анализировать состояние сокетов любого типа, которую предоставляет специальный вызов select(). Его применение достаточно сложно, и на нем следует остановиться подробнее.

Формат вызова следующий:

```
int select (
  int fd_count,
  struct fd_set * reading_set,
  struct fd_set * writing_set,
  struct fd_set * except_set,
  const struct timeval * timeout
);
```

Передаваемые в него по указателю три структуры fd_set представляют собой «множества» файловых описателей (***f***ile ***d***escriptor), которые будут проверяться соответственно на готовность к чтению, готовность к записи и на наличие особых состояний. Вызов изменяет содержимое множеств – в них остаются только те описатели, для которых проверка показала требуемое состояние. Если какая-либо из проверок интереса не представляет, вместо одной из структур передается пустой указатель NULL. Напомним, что в Unix-системах, где появился вызов select(), сокеты приравниваются к файловым описателям, однако в Win-Sockets этот вызов предназначен только для сокетов.

Первый параметр *fd_count* ограничивает число проверяемых описателей. В Windows он игнорируется.

Параметр *timeout* задает максимальное время, в течение которого вызов может ожидать появления хотя бы одного описателя, удовлетворяющего условиям проверок, а затем он обязан завершиться, возвратив пустые множества, что позволяет чередовать проверку с другими действиями. При обнаружении «подходящих» описателей вызов завершается немедленно, и некоторых реализациях в этой же переменной возвращается время, оставшееся до завершения назначенного тайм-аута. Сама структура timeval определена как

```
struct timeval {
  long  tv_sec;  //счетчик секунд
  long  tv_usec; //счетчик микросекунд
};
```

Реализация структуры fd_set считается зависящей от системы, в Win-Sockets она представляет собой массив типа SOCKET и счетчик реально занятых в нем ячеек, размер массива ограничен глобальной константой FD_SETSIZE. По соображениям платформо-независимости работать со структурой fd_set напрямую не рекомендуется, для этого определены следующие макросы:

FD_ZERO – полная очистка множества;
FD_SET – добавление описателя в множество;
FD_CLR – удаление описателя из множества;
FD_ISSET – проверка наличия описателя в множестве.

Таким образом, типичное применение вызова select() включает шаги:

1) подготовка множества: очистка (если нужно) и заполнение;

2) обращение к select();

3) проверка интересующих сокетов на присутствие их в соответствующем множестве и обработка этой ситуации.

Полный список открытых сокетов обычно хранится отдельно – в «эталонной» копии множества или в какой-либо специальной структуре данных.

Вызов select() возвращает количество сокетов, найденных удовлетворяющими условиям проверок, 0 – если истекло время ожидания или SOCKET_ERROR – при ошибке. Проверка результативности важна, так как в последнем случае в множествах могут остаться сокеты, не прошедшие проверку, и обращение к ним может оказаться блокирующим. К такой ошибке приводит в том числе и наличие в множестве сокета с недопустимым значением, например INVALID_SOCKET.

Часто говорят, что приложение, реализующее алгоритм мультиплексирования запросов с помощью select(), использует модель *конечного автомата* (*finite state machine – FSM*). В действительности автоматное представление оказывается даже шире, чем просто описание реакций на состояния сокетов: для ограничения времени реакции сервера опрос сокетов должен выполняться достаточно часто, из-за чего длительные сложные алгоритмы обслуживания отдельных запросов приходится разбивать на стадии, которые также включаются в FSM как отдельные состояния. При этом состояния всех сеансов (запросов) накладываются друг на друга, образуя своего рода многомерное «суперсостояние» всего автомата в целом, и фактически реализуется некоторый аналог корпоративной многозадачности в рамках одного приложения. Проектирование такого автомата бывает очень трудоемкой задачей, часто нетривиальной и требующей высокой квалификации разработчика.

Важнейшим достоинством описанного подхода остается возможность ограничиться единственным процессом (потоком), что позволяет избежать затрат на синхронизацию и неконтролируемого увеличения нагрузки на системные механизмы многозадачности, или даже вообще использовать систему без явной многозадачности. Однако это же ограничивает масштабируемость приложения – без специальных мер оно не сможет эффективно использовать возможности многопроцессорных ЭВМ либо современных многопоточных или многоядерных процессоров. При очень большом количестве открытых сокетов анализ их состояния и манипуляции с массивами начинают требовать достаточно много времени – возможно, больше, чем при использовании событийных механизмов. Но главную проблему составляет само построение адекватного и надежного конечного автомата, реализующего множество состояний множества отдельных сеансов, в каждом из которых ведется обработка различных запросов.

Описанный подход в целом эффективен для приложений с не слишком сложными алгоритмами, ограниченным количеством одновременно поддерживаемых соединений, но при любой интенсивности обращений – запросы, которые приложение не успевает обслужить, просто остаются во входных очередях, что автоматически блокирует поступление следующих. Кроме того, в Unix-системах наряду с сокетами и однотипно с ними можно обрабатывать также и другие устройства ввода-вывода, например файлы или консоль, что существенно упрощает наращивание функциональности приложения.

В качестве примера в прил. 4 приведен усовершенствованный вариант сервера *echo* с поддержкой TCP и UDP, причем число TCP-соединений ограничивается размерностью массивов в структурах fd_set: там размещаются сокет UDP и базовый сокет TCP, а остальные ячейки заполняются сокетами данных TCP по мере их создания. Отметим, что сокеты проверяются только на готовность к чтению. Любой «исправный» сокет, который не был явно закрыт для передачи, при проверке был бы немедленно найден готовым к записи, поэтому такую проверку имеет смысл делать только перед записью в него.

4.7.4. Комбинированный сервер

Иногда алгоритм работы сервера не позволяет обойтись одним мультиплексированием запросов с помощью вызова select() или необходимый для этого конечный автомат оказывается слишком сложным, но при этом чистая многопроцессная или многопоточная схема также неэффективна. В этих случаях может быть построен сервер смешанного типа, состоящего из нескольких (обычно немногих) процессов или потоков, часть из которых реализуют мультиплексирование. Еще одним фактором выбора такой схемы может быть наличие не устранимых с помощью select() блокировок вызовами, которые не используют сокеты явно (например gethostbyname()) или вообще не относятся к подсистеме сокетов (в Unix-системах select() действует на любые файловые дескрипторы, в Windows – нет). При таком подходе удается сочетать главные преимущества обоих подходов и избежать наиболее критичных их недостатков.

К сожалению, это все же не решает радикально проблему ограничения количества активных клиентских соединений, одновременно поддерживаемых сервером, или количества одновременно исполняемых им запросов. На практике скорее приходится оценивать «коэффициент полезного действия» сервера, то есть насколько эффективно он может использовать мощности конкретной вычислительной системы для выполнения конкретных задач.

В качестве самостоятельного упражнения предлагается добавить в сервер из последнего примера (см. прил. 4) возможность интерактивного управления из консоли подобно распараллеливанию ввода сообщений и ожидания ответов на сокетах в двухпоточном UDP-клиенте (см. прил. 3).

4.7.5. Использование асинхронного ввода-вывода

В расширенной версии библиотеки WinSock 2 была добавлена возможность *асинхронной* работы с сокетами. Суть асинхронного ввода-вывода состоит в том, что прикладная программа лишь инициирует операцию неблокирующим вызовом, а затем контролирует ее завершение, сохраняя возможность выполнять другие действия, исполняет же её один из системных (внутренних) потоков, более быстрых и менее ресурсоемких, чем «прикладные».

Этот вариант выходит за рамки стандартных сокетов и в настоящем пособии не рассматривается.

5. ДРУГИЕ ИНТЕРФЕЙСЫ СЕТЕВЫХ ФУНКЦИЙ

Несмотря на распространенность стека протоколов TCP/IP и связанных с ним функций API, на практике применяются также и другие программные интерфейсы. В некоторых случаях они оказываются более простыми в использовании, в других – дают наиболее прямой доступ к тем или иным возможностям. Здесь рассмотрены только интерфейсы Mailslots и NetBIOS.

5.1. Интерфейс Windows Mailslot

Mailslot – один из предусмотренных в Win 32 механизмов межпроцессного взаимодействия (*IPC*). С его помощью можно организовывать передачу данных в пределах одного приложения, между приложениями в пределах одного компьютера и между различными компьютерами сети; обеспечивается также простая синхронизация участников обмена. Mailslot-ами пользуются некоторые службы Windows, например служба пользовательских сообщений. Для прикладных программ они представляют интерес вследствие того, что сочетают простоту использования и ряд свойств, характерных для всех средств сетевого взаимодействия, поэтому на их примере удобно отрабатывать основы построения сетевых приложений.

Собственно Maislot («почтовый ящик») представляет собой именованный системный объект, который может быть создан либо открыт по его имени, после чего становится доступным посредством его *описателя* (*handle*).

Mailslot создается специальным вызовом:

```
HANDLE CreateMailslot (
  lpszName, //имя Mailslot-а
  dwMaxMsgSize, //лимит размера сообщения в Mailslot (0 – лимита нет)
  dwReadTimeout, //лимит время ожидания появления данных в миллисекундах,
       либо 0 – нет ожидания, либо MAILSLOT_WAIT_FOREVER – нет лимита
  lpSecurityAttribute //параметры безопасности, может быть NULL
)
```

Напомним, что открытые файлы в Windows также идентифицируются описателями, которым соответствует тип HANDLE. Описатели объектов Mailslot и File не только совпадают синтаксически – они функционально подобны, то есть к них применимы одни и те же операции чтения и записи. Отличие состоит лишь в том, что Mailslot всегда несимметричен и, следовательно, обеспечивает только однонаправленный обмен: его создатель (владелец) всегда получает права на управление объектом и его чтение (но не на запись!), а все остальные – только на запись.

Имена задаются в виде

\\.\mailslot\[path]name.

(Напомним, в программах на языке С символ ’\’ необходимо экранировать как имеющий специальный смысл.)

Для открытия уже существующего Mailslot используется универсальный вызов CreateFile():

```
HANDLE CreateFile (
  lpszMSlotName, //имя, в том числе необязательно в локальной файловой системе
  dwDesiredAccess, //запрашиваемые права доступа; здесь – GENERIC_WRITE
  dwShareMode, //права совместного доступа
  lpSecurityAttributes, //параметры безопасности, может быть NULL
  dwCreationDisposition, //режим открытия; здесь – OPEN_EXISTING
  dwFlagsAndAttributes, //флаги и атрибуты; здесь можно оставить 0
  hTemplateFile //файл-шаблон; здесь не используется, поэтому NULL
)
```

Вместо ’.’ в строке имени Mailslot-а, обозначающего локальную систему, можно использовать имя другого (удаленного) компьютера (системы), имя домена или символ ’*’, соответствующий всем системам домена. В этом случае все записи, адресованные полученному в результате объекту, будут направлены в одноименный Mailslot соответствующих систем.

Для освобождения объекта Mailslot используется универсальный вызов CloseHandle().

Работа Mailslot-ов обеспечивается сервисами NetBIOS, использующими в свою очередь транспорт TCP/UDP/IP с номерами портов 137 и 138 (UDP) и 139 (TCP) (не следует считать их предназначенными только для обслуживания интерфейса Mailslot – в действительности ими пользуются многие системные сервисы, см. *RFC 1700*). При этом сообщения длиной менее 425 байт передаются в виде датаграммы UDP, а более длинные, но не более 64 Кбайт – через сеанс *SMB* (и соединение TCP). Побочным эффектом является то, что системы Win NT не позволяют передавать через Mailslot сообщения длиной 425 и 426 байт, а Win 9x по имеющимся данным – 425 и более байт.

По сравнению с интерфейсом сокетов Mailslot – средство более высокого уровня и может быть проще и удобнее в использовании, например, благодаря сходству с обычным файлом. Как механизм, встроенный в операционную систему, Mailslot поддерживается независимо от реального набора транспортных протоколов, поэтому и использующая его прикладная программа также не будет зависеть от особенностей сетевых настроек. Интерфейс Mailslot обычно не затрагивается ограничениями и запретами при администрировании сети, хотя отдельные службы могут быть запрещены. Но поддержка Mailslot ограничена только операционными системами Microsoft, а сокеты стали фактическим стандартом сетевого взаимодействия в большинстве современных систем и поэтому пригодны для межплатформенного взаимодействия и платформо-независимых приложений. Кроме того, сокеты, будучи низкоуровневым средством, позволяют более гибко управлять соединением и обменом данными.

Основные приемы работы с объектами Mailslot иллюстрируются на примере двух программ, приведенных в прил. 5. Первая из них («сервер») создает Mailslot и в цикле ожидает появления в нем данных, которые затем выводятся на консоль как текстовые строки. Вторая («клиент») открывает существующий

Mailslot, затем вводит текстовые строки с консоли и передает в него; признаком завершения служит закрытие потока ввода. Имя Mailslot, включая имя узла, может быть задано как параметр командной строки.

5.2. Интерфейс NetBIOS

Еще один простой протокол, реально используемый рядом служб и в некоторых случаях полезный для прикладных программ, – протокол *NetBIOS*.

Протокол NetBIOS охватывает три уровня модели OSI: сетевой, транспортный и сеансовый. Первоначально он разрабатывался для локальных сетей IBM, имеет ряд расширений, например *NetBEUI* (*NetBIOS Enhanced User Interface*), и в настоящее время поддерживается сетевым программным обеспечением многих производителей: IBM, Nowell, Microsoft и так далее, однако реализация его может различаться. В целом NetBIOS представляет собой простое и достаточно универсальное средство взаимодействия в небольших локальных сетях (подсетях), обеспечивая следующие основные возможности:

– идентификация абонента по его адресу (идентификатору) или имени;

– индивидуальная, групповая и широковещательная адресация;

– установление и разрыв соединения;

– передача данных посредством датаграмм или виртуального канала (режим виртуального канала считается основным).

После подключения к интерфейсу (регистрации, см. ниже) пользователь (прикладная программа) получает в свое распоряжение его функционально независимый экземпляр.

Программный интерфейс NetBIOS состоит в наличии системного вызова, которому передается в качестве аргумента адрес сформированного прикладной программой так называемого управляющего блока *NCB – NetBIOS Control Block*. NCB содержит ряд полей, служащих для идентификации абонентов, спецификации выполняемых действий, рабочих областей, дополнительных опций, возврата информации о результате и т.д. В зависимости от предстоящей операции заполняются только необходимые для нее поля, неиспользуемые инициализируются нулями; в ходе выполнения операции драйвер также заполняет некоторые поля NCB и обращается к рабочим областям. При использовании асинхронных команд (см. ниже) на одном экземпляре интерфейса (подключении) можно выполнять несколько операций, не противоречащих друг другу. До завершения операции прикладная программа не должна изменять поля NCB, соответствующего этой операции, можно лишь анализировать текущий код результата (см. ниже).

Помимо NCB вызов манипулирует вспомогательными структурами данных, служащими в основном для представления дополнительных сведений о состоянии интерфейса.

Структура NCB и форматы полей едины для всех средств поддержки интерфейса, но могут различаться в зависимости от версии реализации. Детали синтаксиса деклараций, а также, возможно, имена зависят от языка, системы

программирования и библиотек; здесь в качестве основы принимается синтаксис C/C++ и нотация MS Visual C++.

Windows предусматривает функцию API Netbios() с единственным аргументом – указателем на структуру NCB, соответствующую управляющему блоку NetBIOS. Кроме того, используются вспомогательные структуры ADAPTER_STATUS, ACTION_HEADER, NAME_BUFFER, FIND_NAME_HEADER, FIND_NAME_BUFFER, LANA_ENUM, передаваемые посредством NCB.

Возвращаемое функцией значение соответствует коду возврата (для асинхронных – предварительному, см. ниже).

Вызовы NetBIOS реализуются системным модулем NETAPI32.DLL. Для доступа к ним в среде MS Visual C++ используется библиотека NETAPI32.LIB, необходимые декларации находятся в заголовочном файле NB30.H. В Delphi предусмотрен библиотечный модуль NB30.DCU.

Для идентификации абонентов NetBIOS используются *имена* длиной 16 байт. Имя может относиться к одному из трех типов:

1) *постоянное* – уникально для каждой сетевой платы (если не используются средства эмуляции аппаратных адресов), состоит из 10 лидирующих нулевых байт (символьное представление '\0') и 6 байт аппаратного (*MAC*, см. подразд. 1.7) áдреса платы (именно числовых значений, а не соответствующих им символов – цифр или букв);

2) *локальное* – уникально для каждого подключения к интерфейсу;

3) *групповое* – разделяется несколькими абонентами (группой).

Все имена считаются равноправными с точностью до естественной специфики их использования (например, постоянное имя нельзя удалить, нельзя регистрировать несколько одинаковых локальных имен, и т. д.). Использование групповой или индивидуальной адресации в конкретной операции зависит не от характера имени абонента, а от кода команды.

Зарезервированными считаются имена, начинающиеся с символов:

– '\0' – считается признаком постоянного имени;

– '*' – идентифицирует (локально) собственный экземпляр интерфейса, а также в некоторых случаях любое имя (гарантированно справедливо для имени из единственного символа '*', дополненного 15 пробелами).

В сетях Windows для узла обычно определено также общее для всех NetBIOS-подключений локальное имя, совпадающее с его сетевым именем.

После регистрации имени ему (и созданному подключению) присваивается однобайтовый уникальный *номер* (идентификатор). Зарезервированным считаются значения:

– 0 – идентифицирует собственное подключение;

– FFh – идентифицирует все локальные подключения (при передаче) или любого абонента (при приеме).

Ограничений на количество идентификаторов или имен не предусмотрено, за исключением общего допустимого количества идентификаторов. Программа может иметь несколько подключений и соответствующих им имен.

Действие локального имени и соответствующего ему номера прекращается после удаления имени. Групповое имя существует, пока его использует хотя бы один абонент. Постоянное имя существует всегда и удалению не подлежит.

Операции, выполняемые интерфейсом, специфицируются однобайтовым *кодом команды*. Признак результативности операции также однобайтный, значение 0 соответствует нормальному успешному завершению, прочие сигнализируют об ошибках или особых состояниях. В связи с наличием команд асинхронного типа предусмотрены два вида кода возврата – *предварительный* и *окончательный* (для синхронных команд они совпадают).

Большинство команд NetBIOS имеют два варианта: синхронный и асинхронный. Коды синхронных и асинхронных команд совпадают, за исключением старшего бита: 0 – синхронная, 1 – асинхронная (определена маска ASYNCH, численно равная 80h). Кроме того, для асинхронной команды может быть определена так называемая *post-процедура*, обращение к которой произойдет по завершении операции (фактически процедура обратного вызова). Post-процедура описывается как тип NCB_POST и имеет формат:

```
typedef void CALLBACK (*)(NCB*) NCB_POST_PROC; //MS Visual C++)
TNCBPostProc = procedure(p:TNCB^); //Borland Delphi
```

При выполнении синхронной команды управление вызвавшей функцию программе возвращается только после завершения операции с выставлением признака ncb_retcode, работа приложения (или его потока) на это время блокируется. Вызов функции с асинхронной командой завершается немедленно с выставлением признака результата в ncb_retcode (успех или немедленно выявляемые ошибки), а в поле ncb_cmd_cplt заносится значение NCR_PENDING (FFh) как признак выполнения асинхронной команды, и операция выполняется драйвером независимо от вызвавшей программы; после завершения операции выставляется признак окончательного результата в ncb_cmd_cplt и вызывается post-процедура, если только ее адрес не равен NULL. В последнем случае для контроля состояния команды нужно использовать текущее значение ncb_cmd_cplt, например:

```
while (ncb.ncb_cmd_cplt == NCR_PENDING) //пока FFh
  MyMessagesCycle(); //обработка очередных сообщений
```

что обеспечивает ожидание окончания с сохранением способности приложения реагировать на новые поступающие события.

Применение асинхронных операций значительно повышает эффективность программ, в том числе благодаря параллельному выполнению нескольких операций на одном подключении, а также их надежность при работе с нестабильной сетевой средой, особенно в однозадачных ОС. В многопоточных ОС семейства Win 32 альтернативой асинхронным операциям может быть вынесение обычных синхронных (блокирующих) в отдельный поток.

Все команды NetBIOS можно разделить на следующие группы:

– общее управление интерфейсом и получение информации о нем;

– управление именами (добавление, удаление, поиск);
– управление сеансами (создание и закрытие виртуальных каналов);
– обмен данными посредством виртуального канала;
– обмен данными посредством датаграмм;
– вспомогательные, информационные и прочие команды.

Функциональность виртуальных каналов и датаграмм, реализуемых NetBIOS, в целом соответствует общепринятой.

Виртуальный канал создается при открытии *сеанса* (*session*) NetBIOS. Сеанс идентифицируется его *номером*, определенным локально (для данного подключения), а также, возможно, именем (фигурирует в блоке информации о состоянии сеанса). Сеанс устанавливается только с определенным именем путем одновременного (точнее, перекрывающегося во времени) выполнения обоими участниками пары команд – запрос на соединение и готовность к соединению, разрыв сеанса осуществляется специальной командой. Передача данных требует также одновременного выполнения команд посылки и приема, размер блока, передаваемого одной «обычной» командой, не более 64 Кбайт, но существуют команды «расширенной» передачи «цепочки» из двух блоков суммарной длиной до 128 К. Стандартно передача сопровождается *квитированием* – операция посылки завершается по получении подтверждения о приеме, но существуют команды без ожидания подтверждений, что позволяет повысить производительность канала.

Датаграммы представляют собой блоки данных, передаваемых независимо друг от друга, квитирование не предусматривается. Размер датаграммы ограничен 512 байтами (без учета служебной информации). Датаграммы четко подразделяются на направленные (адресованные определенному адресу, индивидуальному или групповому) и широковещательные на уровне используемых команд: индивидуальные и групповые датаграммы не принимаются командами широковещательного приема и наоборот.

Можно заметить, что функциональность NetBIOS во многом сходна с протоколами TCP и UDP, включая режимы установления связи и передачи данных. Наиболее серьезным ограничением NetBIOS является его адресация, практически пригодная только для локальной сети. Для преодоления этого используется шлюз с NetBIOS на IP и TCP или UDP, однако потребность в глобальной связи средствами NetBIOS возникает редко.

Более подробные сведения о NetBIOS, в том числе полный перечень команд, кодов возврата и т.д. см. в литературе либо электронных справочных системах, например *MSDN*.

В качестве примера в прил. 6 приведена программа, определяющая MAC-адрес компьютера с помощью NetBIOS. Для этого она инициализирует интерфейс, выполняет команду получения его статуса и анализирует поле аппаратного адреса adapter_address в заполненной вызовом структуре. Это один из наиболее простых способов получения MAC-адреса.

ЛИТЕРАТУРА

1. Артамонов Г.Т., Тюрин В.Д. Топология сетей ЭВМ и многопроцессорных систем. – М.: Радио и связь, 1991.

2. Бремнер Л.М. и др. Библиотека программиста Intranet: Основы новейших технологий Intranet: Пер. с англ. / Л.М. Бремнер, Э.Ф. Изи, О. Сервати. – Мн.: Попурри, 1998.

3. Веттинг Д. Novell NetWare: Пер. с нем. – К.: BHV, 1994.

4. Ги К. Введение в локальные вычислительные сети: Пер. с англ. – М.: Радио и связь, 1986.

5. Гук М. Локальные сети Novell. Карманная энциклопедия. – СПб.: Питер, 1996.

6. Ларионов А.М. и др. Вычислительные комплексы, системы и сети: Учебник для вузов. / А.М. Ларионов, С.А. Майоров, Г.И. Новиков. – Л.: Энергоатомиздат, 1987.

7. Ногл М. TCP/IP. Иллюстрированный учебник. – М.: ДМК Пресс, 2001. – 480 с.: ил. / Matthew Naugle. Illustrated TCP/IP. A Graphic Guide to the Protocol Suite.

8. Олифер В.Г., Олифер Н.А. Компьютерные сети. Принципы, технологии, протоколы. – СПб.: Питер, 2001. – 672 с.: ил.

9. Поспелов Д.А. Введение в теорию вычислительных систем. – М.: Сов. радио, 1972.

10. Семенов Ю.А. Протоколы и ресурсы Internet. – М.: Радио и связь, 1996.

11. Снейдер Й. Эффективное программирование TCP/IP. Библиотека программиста. – СПб.: Питер, 2001. – 320 с.: ил.

12. Стивенс У.Р. Протоколы TCP/IP. Практическое руководство. – BHV (серия «В подлиннике»), 2003. – 672 с.: ил.

13. Microsoft TCP/IP: Учеб. курс: [Для самост. подготовки]: Пер. с англ. О.О. Михальского и др.: Под общ. ред. О.О. Михальского. – М.: Русская ред., 1998.

Приложение 1

ПРИМЕР ИСПОЛЬЗОВАНИЯ СОКЕТОВ: ДАТАГРАММЫ UDP

```
/* -                                                                    ---
- UDP_ECHO.C - демонстрационный "сервер" UDP (служба echo):           -
- возвращает все получаемые сообщения отправителям                    -
- Вызов: udp_echo [<порт>]; завершение: <Ctrl-C>                      -
- Порт по умолчанию - 7                                               -
- В проект должен быть включен файл wsock32.lib                       -
---                                                                   - */

#include <windows.h>
#include <stdio.h>
#include <winsock.h>

#define   DEFAULT_ECHO_PORT       7

char DataBuffer [1024];

int main (int argc, char** argv)
{
    struct sockaddr_in SockAddrLocal, SockAddrRemote;
    SOCKET SockLocal = INVALID_SOCKET;
    unsigned short nPort = DEFAULT_ECHO_PORT;
    int nAddrSize, nCnt;
    WSADATA WSAData;
    WORD wWSAVer;
//разбор командной строки: номер порта
    if (argc > 1)
        if (sscanf (argv[1], "%u", &nPort) < 1)
            fprintf (stderr, "Ошибочный порт: %s, use default", nPort);
//инициализация подсистемы сокетов
    wWSAVer = MAKEWORD (1, 1);
    if (WSAStartup (wWSAVer, &WSAData) != 0) {
        puts ("Ошибка инициализации подсистемы WinSocket");
        return -1;
    }
//создание локального сокета
    SockLocal = socket (PF_INET, SOCK_DGRAM, 0);
    if (SockLocal==INVALID_SOCKET) {
        fputs ("Ошибка создания сокета\n", stderr);
        return -1;
    }
//привязка сокета к локальному адресу
    memset (&SockAddrLocal, 0, sizeof(SockAddrLocal));
    SockAddrLocal.sin_family = AF_INET;
    SockAddrLocal.sin_addr.S_un.S_addr = INADDR_ANY;
    SockAddrLocal.sin_port = htons (nPort); //(<номер_порта_сервера>);
    if (bind (SockLocal,
        (struct sockaddr*) &SockAddrLocal, sizeof(SockAddrLocal)
```

```
    ) != 0)
    {
        fprintf (stdout, "Ошибка привязки сокета, порт %u\n",
            ntohs(SockAddrLocal.sin_port)
        );
        return -1;
    }
    fprintf (stderr, "Сервер запущен, порт %u\n",
        ntohs(SockAddrLocal.sin_port)
    );
//основной рабочий цикл
    while (1) { //для сервера цикл обычно бесконечен
        nAddrSize = sizeof (SockAddrRemote);
    //принять входящее сообщение ("эхо-запрос")
        nCnt = recvfrom (SockLocal,
            DataBuffer, sizeof(DataBuffer)-1, 0,
            (struct sockaddr*) &SockAddrRemote, &nAddrSize
        );
        if (nCnt < 0) { //ошибка приема запроса
            fputs ("Ошибка приема сообщения\n", stderr);
            continue;
        }
    //вернуть сообщение как эхо-отклик
        sendto (SockLocal,
            DataBuffer, nCnt, 0,
            (struct sockaddr*) &SockAddrRemote, sizeof (SockAddrRemote)
        );
    }
//завершение - здесь никогда не достигается!
    shutdown (SockLocal, 2);
    Sleep (100);
    closesocket (SockLocal); SockLocal = INVALID_SOCKET;
    WSACleanup ();
    return 0;
}


/* -                                                          ---
- UDP_SEND.C - демонстрационный "клиент" UDP:                  -
- отсылает на "сервер" одно или несколько сообщений,          -
- ждет и отображает полученные ответы                         -
- Вызов: udp_send <адрес/имя> <порт> <сообщение1> ...         -
- Завершение: исчерпание списка сообщений или <Ctrl-C>        -
- В проект должен быть включен файл wsock32.lib               -
---                                                         - */

#include <windows.h>
#include <stdio.h>
#include <winsock.h>

char DataBuffer [1024];
```

```
int main (int argc, char** argv)
{
    struct sockaddr_in SockAddrLocal, SockAddrSend, SockAddrRecv;
    SOCKET SockLocal = INVALID_SOCKET;
    struct hostent* pHostEnt;
    int nSockOptBC, nAddrSize, nPortRemote, nMsgLen, i;
    WSADATA WSAData;
    WORD wWSAVer;
//командная строка
    if (argc < 3) {
        puts ("Недостаточно аргументов\n");
        puts ("Вызов: UDP_SEND <addr/name> <port> <msg1> ...\n");
        return -1;
    }
//инициализация подсистемы сокетов
    wWSAVer = MAKEWORD (1, 1);
    if (WSAStartup (wWSAVer, &WSAData) != 0) {
        puts ("Ошибка инициализации WinSockets");
        return -1;
    }
//создание локального сокета
    SockLocal = socket (PF_INET, SOCK_DGRAM, 0);
    if (SockLocal==INVALID_SOCKET) {
        fputs ("Ошибка создания сокета\n", stderr);
        return -1;
    }
//настройка сокета: разрешить отсылку на "широковещательный" адрес
    nSockOptBC = 1;
    setsockopt (SockLocal,
        SOL_SOCKET, SO_BROADCAST,
        (char*) (&nSockOptBC), sizeof(nSockOptBC)
    );
//привязка сокета к "локальному" адресу
    memset (&SockAddrLocal, 0, sizeof(SockAddrLocal));
    SockAddrLocal.sin_family = AF_INET;
    SockAddrLocal.sin_addr.S_un.S_addr = INADDR_ANY; //все интерфейсы
    SockAddrLocal.sin_port = 0; //выбирать порт автоматически
    if (bind(SockLocal,(struct sockaddr*)&SockAddrLocal,sizeof(SockAddrLocal)) != 0)
    {
        fputs ("Ошибка привязки к локальному адресу\n", stderr);
        return -1;
    }
//подготовка адреса сервера
    memset (&SockAddrSend, 0, sizeof (SockAddrSend));
    SockAddrSend.sin_family = AF_INET;
    if (strcmp (argv[1], "255.255.255.255") == 0) //адрес broadcast
        SockAddrSend.sin_addr.S_un.S_addr = INADDR_BROADCAST;
    else {
        SockAddrSend.sin_addr.S_un.S_addr = inet_addr (argv[1]);
        if (SockAddrSend.sin_addr.S_un.S_addr == INADDR_NONE) {
            if ((pHostEnt = gethostbyname (argv[1])) == NULL) {
                fprintf (stderr, "Хост не опознан: %s\n", argv[1]);
```

```
            return -1;
        }
        SockAddrSend.sin_addr = *(struct in_addr*)(pHostEnt->h_addr_list[0]);
    }
}
if (sscanf (argv[2], "%u", &nPortRemote) < 1) {
    fprintf (stderr, "Ошибочный номер порта: %s\n", argv[2]);
    return -1;
}
SockAddrSend.sin_port = htons ((unsigned short)nPortRemote);
//рабочий цикл
for (i=3; i<argc; ++i) { //остальные аргументы командной строки
//отослать сообщение
    fprintf (stdout, "Отсылка на %s:%u: \"%s\" \n",
        inet_ntoa(SockAddrSend.sin_addr),
        ntohs(SockAddrSend.sin_port),
        argv[i]
    );
    nMsgLen = strlen (argv[i]) + 1;
    if (sendto (SockLocal, argv[i], nMsgLen, 0,
            (struct sockaddr*) &SockAddrSend, sizeof (SockAddrSend)) < nMsgLen)
    {
        fprintf (stderr, "Ошибка отсылки: \"%s\"\n", argv[i]);
        continue;
    }
//принять и отобразить ответ
    nAddrSize = sizeof (SockAddrRecv);
    nMsgLen = recvfrom (SockLocal, DataBuffer, sizeof(DataBuffer)-1, 0,
        (struct sockaddr*) &SockAddrRecv, &nAddrSize
    );
    if (nMsgLen <= 0) { //ошибка приема запроса
        fputs ("Ошибка приема ответа\n", stderr);
        continue;
    }
    DataBuffer [nMsgLen] = '\0';
    fprintf (stdout, "Ответ от %s:%u: \"%s\" \n",
        inet_ntoa (SockAddrRecv.sin_addr),
        ntohs (SockAddrRecv.sin_port),
        DataBuffer
    );
}
//завершение
shutdown (SockLocal, 2);
Sleep (100);
closesocket (SockLocal); SockLocal = INVALID_SOCKET;
WSACleanup ();
return 0;
}
```

Приложение 2

ПРИМЕР ИСПОЛЬЗОВАНИЯ СОКЕТОВ: СОЕДИНЕНИЯ TCP

```
/* -                                                              ---
- TCP_ECHO.C - демонстрационный сервер TCP (служба echo):        -
- принимает соединения от клиентов,                              -
- принимает сообщения и возвращает их клиентам.                  -
- Одновременно поддерживает только одно соединение.              -
- Вызов: tcp_echo [<порт>]; завершение: <Ctrl-C>                 -
- Порт по умолчанию - 7                                          -
- В проект должен быть включен файл wsock32.lib                  -
---                                                            - */

#include <windows.h>
#include <stdio.h>
#include <winsock.h>

#define   DEFAULT_ECHO_PORT       7

char DataBuffer [1024];

int main (int argc, char** argv)
{
    struct sockaddr_in SockAddrBase, SockAddrPeer;
    SOCKET SockBase = INVALID_SOCKET, SockData = INVALID_SOCKET;
    unsigned short nPort = DEFAULT_ECHO_PORT;
    int nAddrSize, nCnt;
    WSADATA WSAData;
    WORD wWSAVer;
//разбор командной строки: номер порта
    if (argc > 1)
        if (sscanf (argv[1], "%u", &nPort) < 1)
            fprintf (stderr, "Ошибочный порт: %s, use default", nPort);
//инициализация подсистемы сокетов
    wWSAVer = MAKEWORD (1, 1);
    if (WSAStartup (wWSAVer, &WSAData) != 0) {
        puts ("Ошибка инициализации подсистемы WinSocket");
        return -1;
    }
//создание локального сокета
    SockBase = socket (PF_INET, SOCK_STREAM, 0);
    if (SockBase == INVALID_SOCKET) {
        fputs ("Ошибка создания сокета\n", stderr);
        return -1;
    }
//привязка базового сокета к локальному адресу
    memset (&SockAddrBase, 0, sizeof(SockAddrBase));
    SockAddrBase.sin_family = AF_INET;
    SockAddrBase.sin_addr.S_un.S_addr = INADDR_ANY;
    SockAddrBase.sin_port = htons (nPort); //(<номер_порта_сервера>);
    if (bind (SockBase,
```

```
        (struct sockaddr*) &SockAddrBase, sizeof(SockAddrBase)
    ) != 0)
    {
        fprintf (stderr, "Ошибка привязки к локальному порту: %u\n",
            ntohs(SockAddrBase.sin_port)
        );
        return -1;
    }
//включение режима "прослушивания"
    if (listen (SockBase, 2) != 0) { //очередь на 2 места
        closesocket (SockBase);
        fputs ("Ошибка включения режима прослушивания\n", stderr);;
        return -1;
    }
    fprintf (stderr,
        "Сервер запущен, порт %u\n",
        ntohs(SockAddrBase.sin_port)
    );
//основной рабочий цикл - прием и обслуживание соединений
    while (1) { //для сервера цикл обычно бесконечен
        nAddrSize = sizeof (SockAddrPeer);
        SockData = accept (SockBase,
            (struct sockaddr*)&SockAddrPeer, &nAddrSize
        );
        if (SockData == INVALID_SOCKET) {
            fputs ("Ошибка приема соединения\n", stderr);
            continue;
        }
    //цикл обслуживания одного соединения
        while (1) {
            nCnt = recv (SockData, DataBuffer, sizeof(DataBuffer)-1, 0);
            if (nCnt <= 0)
                break;
            send (SockData, DataBuffer, nCnt, 0); //возврат "эха"
        }
        shutdown (SockData, 2);
        closesocket (SockData); SockData = INVALID_SOCKET;
    }
//завершение - здесь никогда не достигается!
    shutdown (SockBase, 2);
    Sleep (100);
    closesocket (SockBase); SockBase = INVALID_SOCKET;
    WSACleanup ();
    return 0;
}
```

```
/* -                                                              ---
- TCP_SEND.C - демонстрационный клиент TCP:                       -
- отсылает на сервер одно или несколько сообщений,               -
- после каждого ждет получения ответа и отображает его           -
- Вызов: tcp_send <адрес/имя> <порт> <сообщение1> ...             -
- Завершение: исчерпание списка сообщений или <Ctrl-C>           -
- В проект должен быть включен файл wsock32.lib                  -
---                                                            - */

#include <windows.h>
#include <stdio.h>
#include <winsock.h>

char DataBuffer [1024];

int main (int argc, char** argv)
{
    struct sockaddr_in SockAddrServer;
    SOCKET SockData = INVALID_SOCKET;
    struct hostent* pHostEnt;
    int nPortServer, nMsgLen, i;
    WSADATA WSAData;
    WORD wWSAVer;
//командная строка
    if (argc < 3) {
        puts ("Недостаточно аргументов\n");
        puts ("Вызов: TCP_SEND <addr/name> <port> <msg1> ...\n");
        return -1;
    }
//инициализация подсистемы сокетов
    wWSAVer = MAKEWORD (1, 1);
    if (WSAStartup (wWSAVer, &WSAData) != 0) {
        puts ("Ошибка инициализации WinSocket");
        return -1;
    }
//подготовка адреса сервера
    memset (&SockAddrServer, 0, sizeof (SockAddrServer));
    SockAddrServer.sin_family = AF_INET;
    if (strcmp (argv[1], "255.255.255.255") == 0) //адрес broadcast
        SockAddrServer.sin_addr.S_un.S_addr = INADDR_BROADCAST;
    else {
        SockAddrServer.sin_addr.S_un.S_addr = inet_addr (argv[1]);
        if (SockAddrServer.sin_addr.S_un.S_addr == INADDR_NONE) {
            if ((pHostEnt = gethostbyname (argv[1])) == NULL) {
                fprintf (stderr, "Хост не опознан: %s\n", argv[1]);
                return -1;
            }
            SockAddrServer.sin_addr = *(struct in_addr*)(pHostEnt->h_addr_list[0]);
        }
    }
    if (sscanf (argv[2], "%u", &nPortServer) < 1) {
        fprintf (stderr, "Ошибочный номер порта: %s\n", argv[2]);
```

```
        return -1;
    }
    SockAddrServer.sin_port = htons ((unsigned short)nPortServer);
//создание локального сокета
    SockData = socket (PF_INET, SOCK_STREAM, 0);
    if (SockData == INVALID_SOCKET) {
        fputs ("Ошибка создания сокета\n", stderr);
        return -1;
    }
//запрос на установление соединения
    if (connect (SockData,
        (const struct sockaddr*)&SockAddrServer, sizeof(SockAddrServer)) != 0)
    {
        fprintf (stderr, "Ошибка соединения с %s:%u\n",
            inet_ntoa (SockAddrServer.sin_addr),
            ntohs (SockAddrServer.sin_port)
        );
        closesocket (SockData);
        return -1;
    }
    fprintf (stdout, "Установлено соединение с сервером: %s:%u\n",
        inet_ntoa (SockAddrServer.sin_addr),
        ntohs (SockAddrServer.sin_port)
    );
//рабочий цикл
    for (i=3; i<argc; ++i) { //остальные аргументы командной строки
    //отослать сообщение
        fprintf (stdout, "Отсылка: \"%s\" \n", argv[i]);
        nMsgLen = strlen (argv[i]) + 1;
        if (send (SockData, argv[i], nMsgLen, 0) < nMsgLen) {
            fprintf (stderr, "Ошибка отсылки: \"%s\"\n", argv[i]);
            continue;
        }
    //принять и отобразить ответ
        fprintf (stdout, "Прием...");
        nMsgLen = recv (SockData, DataBuffer, sizeof(DataBuffer)-1, 0);
        if (nMsgLen <= 0) { //ошибка приема ответа
            fputs ("Ошибка приема\n", stderr);
            continue;
        }
        DataBuffer [nMsgLen] = '\0';
        fprintf (stdout, "\b\b\b: \"%s\" \n", DataBuffer);   }
//завершение
    shutdown (SockData, 2);
    Sleep (100);
    closesocket (SockData); SockData = INVALID_SOCKET;
    WSACleanup ();
    return 0;
}
```

Приложение 3

ПРИМЕР ИСПОЛЬЗОВАНИЯ СОКЕТОВ: КЛИЕНТ UDP С ИСПОЛЬЗОВАНИЕМ МНОГОПОТОЧНОСТИ

```
/* -                                                                  ---
- TNET_UDP.C - демонстрационный "клиент" UDP                          -
- (упрощенный аналог telnet):                                        -
- считывает сообщения из стандартного потока ввода, отсылает их      -
- на заданный "сервер" и отображает полученные ответы.               -
- Ожидания ввода и ответов реализованы в параллельных потоках.       -
- Вызов: tnet_udp <адрес/имя> <порт>                                 -
- Завершение: закрытие stdin (<Ctrl-Z> в DOS/Windows), <Ctrl-C>      -
- В проект должна быть включена библиотека wsock32.lib,              -
- стандартные библиотеки должны быть multithread                     -
---                                                                 - */

#include <windows.h>
#include <stdio.h>
#include <winsock.h>

void Sender (SOCKET SockSend, const struct sockaddr_in *pSockAddrSend);
DWORD WINAPI Receiver (void* pArg);

bool FillInetAddr (struct sockaddr_in* pSockAddr,
    const char* pszIP, u_long nDefIP,
    const char* pszPort, u_short nDefPort);

int main (int argc, char** argv)
{
    struct sockaddr_in SockAddrLocal, SockAddrRemote;
    SOCKET SockLocal = INVALID_SOCKET;
    HANDLE hReceiver = NULL;
    DWORD idRecv; //реально не используется, но необходим в Win 9x
    int nSockOptBC;
    WSADATA WSAData;
    WORD wWSAVer;
//командная строка
    if (argc < 3) {
        fputs ("Недостаточно аргументов\n", stderr);
        fputs ("Формат вызова: TNET_UDP <адрес/имя> <порт>\n", stderr);
        return -1;
    }
//инициализация подсистемы сокетов
    wWSAVer = MAKEWORD (1, 1);
    if (WSAStartup (wWSAVer, &WSAData) != 0) {
        puts ("Ошибка инициализации подсистемы Winsocket");
        return -1;
    }
//создание локального сокета
    SockLocal = socket (PF_INET, SOCK_DGRAM, 0);
    if (SockLocal==INVALID_SOCKET) {
```

```
        fputs ("Ошибка создания локального сокета\n", stderr);
        return -1;
    }
//настройка сокета: разрешить работу с широковещательными данными
    nSockOptBC = 1;
    setsockopt (SockLocal, SOL_SOCKET, SO_BROADCAST,
        (char*) (&nSockOptBC), sizeof(nSockOptBC)
    );
//привязка сокета к локальному адресу
    FillInetAddr (&SockAddrLocal, NULL, INADDR_ANY, NULL, 0);
    if (bind (SockLocal, (struct sockaddr*) &SockAddrLocal, sizeof(SockAddrLocal)
    ) != 0)
    {
        fputs ("Ошибка привязки локального сокета\n", stderr);
        closesocket (SockLocal); SockLocal = INVALID_SOCKET;
        WSACleanup ();
        return -1;
    }
//подготовка адреса сервера
    if (! FillInetAddr (&SockAddrRemote, argv[1], INADDR_BROADCAST, argv[2], 0)) {
        closesocket (SockLocal);
        WSACleanup ();
        return -1;
    }
//запустить поток приема ответов
    hReceiver =
        CreateThread (NULL, 0, Receiver, (void*)SockLocal, 0, &idRecv);
    if (hReceiver == NULL) {
        fputs ("Ошибка создания потока\n", stderr);
        closesocket (SockLocal);
        WSACleanup ();
        return -1;
    }
//запустить отсылку сообщений в главном потоке
    Sender (SockLocal, &SockAddrRemote);
//завершение
    TerminateThread (hReceiver, 0);
    shutdown (SockLocal, 2);
    Sleep (100);
    closesocket (SockLocal); SockLocal = INVALID_SOCKET;
    WSACleanup ();
    return 0;
}

/* -- Функция ввода и отсылки в сокет сообщений.
Выполняется в общем (главном) потоке.
*/
void Sender (SOCKET SockSend, const struct sockaddr_in *pSAddrSend)
{
    char DataSendBuff [1024];
    int nSended, i;
//
```

```
    if ((SockSend == INVALID_SOCKET) && (pSAddrSend == NULL))
        return;
    printf ("Отсылка сообщений на хост: %s:%u\n",
        inet_ntoa (pSAddrSend->sin_addr), ntohs (pSAddrSend->sin_port)
    );
//рабочий цикл: прочитать и отослать сообщение
    while (1) {
        printf ("> ");
        if (fgets (DataSendBuff, sizeof(DataSendBuff)-1, stdin) == NULL)
            break;
        if ((i = strlen (DataSendBuff)) > 0) {
            for (--i; (DataSendBuff[i]=='\n') || (DataSendBuff[i]=='\r'); --i); ++ i;
            DataSendBuff[i] = '\0';
        }
        nSended = sendto (SockSend, DataSendBuff, i+1, 0,
                (const struct sockaddr*)pSAddrSend, sizeof(*pSAddrSend));
        if (nSended < (i+1))
            fputs ("Ошибка отсылки сообщения\n", stderr);
        Sleep (20); //пауза - дать отработать приемнику
    }
}

/* -- Функция приема и отображения сообщений, поступающих в сокет.
Выполняется в отдельном потоке.
*/
DWORD WINAPI Receiver (void* pArg)
{
    char DataRecvBuffer [1024];
    struct sockaddr_in SockAddrRecv;
    int nAddrSize, nReceived;
    SOCKET SockRecv = (SOCKET)pArg;
//рабочий цикл (бесконечный): принять и отобразить сообщение
    while (1) {
    //принять
        nAddrSize = sizeof (SockAddrRecv);
        nReceived = recvfrom (SockRecv,
            DataRecvBuffer, sizeof(DataRecvBuffer)-1, 0,
            (struct sockaddr*) &SockAddrRecv, &nAddrSize
        );
        if (nReceived <= 0) {
            fputs ("Ошибка приема ответа\n", stderr);
            continue;
        }
    //отобразить
        DataRecvBuffer [nReceived] = '\0';
        printf ("От %s:%u: \"%s\" \n",
            inet_ntoa(SockAddrRecv.sin_addr), ntohs(SockAddrRecv.sin_port),
            DataRecvBuffer
        );
    }
    return 0;
}
```

```
/* -- Заполнение структуры адреса (семейство INet)
Вход:
    IP-адрес и порт в виде строк (вместо адреса может быть сетевое имя);
    их значения по умолчанию (используются если строки пустые или NULL)
Выход:
    TRUE если преобразование успешно, FALSE если ошибка
*/
bool FillInetAddr (struct sockaddr_in* pSockAddr,
    const char* pszIP, u_long nDefIP,
    const char* pszPort, u_short nDefPort)
{
    struct hostent* pHostEnt;
    u_long nIP, nPort;
//проверка и инициализация
    if (pSockAddr == NULL)
        return false;
    memset (pSockAddr, 0, sizeof(struct sockaddr_in));
    pSockAddr->sin_addr.s_addr = htonl (nDefIP);
    pSockAddr->sin_port = htons (nDefPort);
//семейство адреса
    pSockAddr->sin_family = AF_INET;
//ip-адрес
    if ((pszIP != NULL) && (pszIP[0] != '\0')) { //IP задан явно
        if (strcmp (pszIP, "255.255.255.255") == 0)
            nIP = INADDR_BROADCAST;
        else {
            if ((nIP = inet_addr (pszIP)) == INADDR_NONE) {
                if ((pHostEnt = gethostbyname (pszIP)) == NULL) {
                    fprintf (stderr, "Хост не опознан: %s\n", pszIP);
                    return false;
                }
                nIP = ((struct in_addr*)(pHostEnt->h_addr_list[0]))->s_addr;
            }
        }
        pSockAddr->sin_addr.s_addr = nIP;
    }
//номер порта
    if ((pszPort != NULL) && (pszPort[0] != '\0')) { //порт задан явно
        if (sscanf (pszPort, "%lu", &nPort) < 1) {
            fprintf (stderr, "Ошибочный номер порта: %s\n", pszPort);
            return false;
        }
        pSockAddr->sin_port = htons ((u_short)nPort);
    }
//завершение
    return true;
}
```

Приложение 4

ПРИМЕР ИСПОЛЬЗОВАНИЯ СОКЕТОВ: МНОГОПОЛЬЗОВАТЕЛЬСКИЙ ОДНОПОТОЧНЫЙ СЕРВЕР TCP/UDP

```
/* -                                                              ---
- ECHO_MUL.C - демонстрационный сервер echo (RFC-862)            -
- Поддержка TCP (множественные соединения) и UDP.                -
- Однопоточная реализация, используется вызов select().          -
- Вызов: echo_mul [<порт>]; завершение: <Ctrl-C>                 -
- Порт по умолчанию - 7 (TCP и UDP)                              -
- Рассчитан на Winsocket версии 1, требует wsock32.lib           -
---                                                            - */

#include <windows.h>
#include <winsock.h> //winsock2.h нужно было бы включать до windows.h
#include <stdio.h>

/* --- Глобальные константы --- */

#define  DEF_ECHO_PORT   7   //по умолчанию - стандартный ECHO (RFC-1700)
#define  MAX_TCP_CONNECTIONS     (FD_SETSIZE - 2) //лимит соединений TCP
#define  MAX_MSG_LENGTH    1024  //лимит длины сообщений

#ifndef SD_BOTH
#define SD_BOTH 2   //этой константы нет в winsocket.h
#endif

/* --- Глобальные переменные --- */

SOCKET BaseSockTCP = INVALID_SOCKET; //базовый сокет TCP
SOCKET DataSocksTCP [MAX_TCP_CONNECTIONS]; //сокетов TCP-соединений
SOCKET SockUDP = INVALID_SOCKET; //сокет UDP для приема/передачи данных

struct fd_set ReadSet; //множество сокетов для вызова select()

/* --- Прототипы подпрограмм --- */

bool UDP_MakeServer (const struct sockaddr_in* pLocalAddr);
void UDP_DataProc (void);
bool TCP_MakeServer (const struct sockaddr_in* pLocalAddr);
void TCP_AcceptProc (void);
void TCP_DataProc (void);
int PrepareSockSets (void);
bool FillInetAddr (struct sockaddr_in* pSockAddr,
       const char* pszIP, u_long nDefIP,
       const char* pszPort, u_short nDefPort);
void CloseSocket (SOCKET* pSockToClose);

/* --- Головной модуль программы - основной алгоритм --- */
```

```
int main (int argc, char* argv[])
{
   WSADATA WSADat;
   struct sockaddr_in LocalAddr;
   struct timeval WaitLimit;
   int i, nSockCnt;
   bool fExit; //флаг выхода
//инициализация
   if (WSAStartup (0x0202, &WSADat) != 0) {
      fputs ("Ошибка инициализации подсистемы WinSocket\n", stderr);
      return -1;
   }
   if (! FillInetAddr (&LocalAddr,
      NULL, INADDR_ANY, ((argc>1)?argv[1]:NULL), DEF_ECHO_PORT))
   {
      WSACleanup ();
      return -1;
   }
   for(i=0; i<MAX_TCP_CONNECTIONS; ++i)
      DataSocksTCP[i] = INVALID_SOCKET;
//подготовка серверов UDP и TCP
   UDP_MakeServer (&LocalAddr);
   TCP_MakeServer (&LocalAddr);
//основной цикл сервера (флаг зарезервирован для "мягкого прерывания)
   for (fExit=false; !fExit; ++i) {
//заполнение множества проверяемых сокетов
      if ((nSockCnt = PrepareSockSets ()) < 1)
         break; //ни одного сокета, цикл не имеет смысла
//запрос состояния сокетов
      WaitLimit.tv_sec = 1; WaitLimit.tv_usec = 0; //ожидание – 1.0 сек.
      if (select (nSockCnt, &ReadSet, NULL, NULL, &WaitLimit) <= 0)
         continue; //ошибка или тайм-аут
//обслуживание запросов
      TCP_AcceptProc ();
      UDP_DataProc ();
      TCP_DataProc ();
   }
//завершение - сейчас никогда не достигается!
   for(i=0; i<MAX_TCP_CONNECTIONS; ++i)
      CloseSocket (&(DataSocksTCP[i]));
   CloseSocket (&BaseSockTCP); CloseSocket (&SockUDP);
   WSACleanup ();
   return 0;
}

/* --- Подпрограммы сервера UDP --- */

/* -- Создание сокета сервера UDP
Вход:
   Локальный адрес (для привязки сокета)
Выход:
   TRUE если сокет создан и подготовлен, иначе FALSE
```

```
Глобальные переменные:
   SockUDP - сокет UDP (для приема и передачи данных)
*/
bool UDP_MakeServer (const struct sockaddr_in* pLocalAddr)
{
   if (pLocalAddr == NULL)
      return false;
   if ((SockUDP = socket (PF_INET, SOCK_DGRAM, 0)) == INVALID_SOCKET) {
      fputs ("Ошибка создания сокета UDP\n", stderr);
      return false;
   }
   if (bind (SockUDP, (struct sockaddr*)pLocalAddr, sizeof(*pLocalAddr)) != 0)
   {
      fprintf (stderr, "Ошибка привязки сокета UDP на порт %u\n",
         pLocalAddr->sin_port);
      CloseSocket (&SockUDP);
      return false;
   }
   printf ("UDP-часть запущена, порт %u\n",
      ntohs (pLocalAddr->sin_port));
   return true;
}

/* -- Обработка запросов UDP
Глобальные переменные:
   SockUDP - сокет приема/передачи данных UDP
   ReadSet - множество сокетов после проверки готовности
*/
void UDP_DataProc (void)
{
   char DataBuf [MAX_MSG_LENGTH];
   struct sockaddr_in RemoteAddr;
   int nReceived, nSended;
   int nAddrLen = sizeof(RemoteAddr);
//
   if ((SockUDP == INVALID_SOCKET) || (! FD_ISSET (SockUDP, &ReadSet)))
      return;
//прием данных и адреса, с которого они поступили
   nReceived = recvfrom (SockUDP, DataBuf, sizeof(DataBuf), 0,
      (struct sockaddr*)&RemoteAddr, &nAddrLen
   );
   switch (nReceived) {
      case 0:
         break;
      case SOCKET_ERROR:
         fputs ("Error receiving data\n", stderr);
         break;
      default: //nReceived>0
//отчет о запросе
         printf ("UDP: запрос от %s:%u\n",
            inet_ntoa (RemoteAddr.sin_addr), ntohs (RemoteAddr.sin_port));
//ответ отправителю данных
```

```
            nSended = sendto (SockUDP, DataBuf, nReceived, 0,
                (const struct sockaddr*)&RemoteAddr, sizeof(RemoteAddr)
            );
            if (nSended == SOCKET_ERROR)
                fputs ("Error sending data\n", stderr);
    }
}

/* --- Подпрограммы сервера TCP --- */

/* -- Создание базового сокета сервера TCP
Вход:
   Локальный адрес (для привязки сокета)
Выход:
   TRUE если базовый сокет создан и подготовлен, иначе FALSE
Глобальные переменные:
   BaseSockTCP - базовый сокет TCP (сокет приема соединений)
*/
bool TCP_MakeServer (const struct sockaddr_in* pLocalAddr)
{
    if (pLocalAddr == NULL)
        return false;
//создание базовго сокета
    BaseSockTCP = socket (PF_INET, SOCK_STREAM, 0);
    if (BaseSockTCP == INVALID_SOCKET) {
        fputs ("Ошибка создания базового сокета TCP\n", stderr);
        return false;
    }
//привязка к локальному адресу
    if (bind (BaseSockTCP, (struct sockaddr*)pLocalAddr, sizeof(*pLocalAddr)) != 0)
    {
        fprintf (stderr, "Ошибка привязки базового сокета TCP на порт %u\n",
            pLocalAddr->sin_port);
        CloseSocket (&BaseSockTCP);
        return false;
    }
//включение режима прослушивания
    if (listen (BaseSockTCP, 4) != 0) { //очередь на 4 места
        fputs ("Ошибка включения базового сокета TCP", stderr);
        CloseSocket (&BaseSockTCP);
        return false;
    }
//завершение
    printf ("TCP-часть запущена, порт %u\n",
        ntohs (pLocalAddr->sin_port));
    return true;
}

/* -- Прием соединения (обработка запроса на соединение)
Выход:
   Новый сокет данных или INVALID_SOCKET если ошибка или нет TCP-запросов
Глобальные переменные:
```

```
   BaseSock - базовый сокет
   ReadSet - множество сокетов данных после проверки готовности
*/
void TCP_AcceptProc (void)
{
   SOCKET DataSock;
   struct sockaddr_in ClientAddr;
   int i;
   int nAddrLen = sizeof(ClientAddr);
//проверка состояния
   if ((BaseSockTCP==INVALID_SOCKET) || (! FD_ISSET (BaseSockTCP, &ReadSet)))
      return;
//прием соединения, создание сокета данных
   DataSock = accept (BaseSockTCP, (struct sockaddr*)(&ClientAddr), &nAddrLen);
   if (DataSock == INVALID_SOCKET) {
      fputs ("Ошибка приема соединения\n", stderr);
      return;
   }
//сохранение нового сокета данных в списке
   for (i=0; (i<MAX_TCP_CONNECTIONS); ++i)
      if (DataSocksTCP[i] == INVALID_SOCKET) {
         DataSocksTCP[i] = DataSock; //новое соединение в список
//отчет о новом соединении
         printf ("TCP: соединение (#%d) с %s:%u\n",
            i, inet_ntoa (ClientAddr.sin_addr), ntohs (ClientAddr.sin_port));
         return;
      }
   fputs ("Ошибка - превышен лимит TCP-соединений сервера\n", stderr);
   CloseSocket (&DataSock);
}

/* -- Обработка запросов соединений TCP
Глобальные переменные:
   DataSocksTCP - список (массив) сокетов данных
   ReadSet - множество сокетов после проверки готовности
*/
void TCP_DataProc (void)
{
   char DataBuf [MAX_MSG_LENGTH];
   int i, nReceived, nSended;
//проверка всех сокетов данных в списке
   for (i=0; i<MAX_TCP_CONNECTIONS; ++i) {
      if ((DataSocksTCP[i] != INVALID_SOCKET)
         && (FD_ISSET (DataSocksTCP[i], &ReadSet)))
      {
//прием данных, отсылка данных
         nReceived = recv (DataSocksTCP[i], DataBuf, sizeof(DataBuf), 0);
         switch (nReceived) {
            case 0: //принят пустой сегмент, разрыв соединения
               CloseSocket (&(DataSocksTCP[i]));
               printf ("TCP: разрыв соединения (#%d)\n", i); //отчет
               break;
```

```
                case SOCKET_ERROR: //ошибка приема
                    fputs ("Ошибка приема данных\n", stderr);
                    CloseSocket (&(DataSocksTCP[i]));
                    break;
                default: //принят сегмент данных
                    nSended = send (DataSocksTCP[i], DataBuf, nReceived, 0);
                    if (nSended == SOCKET_ERROR)
                        fputs ("Ошибка отсылки данных\n", stderr);
                    break;
            }
        }
    }
}

/* --- Вспомогательные подпрограммы --- */

/* -- Заполнение множеств (struct fd_set) сокетов для проверки
Выход:
   Количество сокетов, включенных в множество
Использует глобальные переменные:
   BaseSockTCP, DataSocksTCP[], SockUDP - сокеты, подлежащие проверке
   ReadSet - заполняемое множество
*/
int PrepareSockSets (void)
{
   int nCnt = 0;
//
   FD_ZERO (&ReadSet);
   if (BaseSockTCP != INVALID_SOCKET) { //базовый сокет TCP
       FD_SET (BaseSockTCP, &ReadSet);
       ++ nCnt;
   }
   if (SockUDP != INVALID_SOCKET) { //сокет UDP
       FD_SET (SockUDP, &ReadSet);
       ++ nCnt;
   }
   for (int i=0; i<MAX_TCP_CONNECTIONS; ++i) //сокеты данных TCP
       if (DataSocksTCP[i] != INVALID_SOCKET) {
           FD_SET (DataSocksTCP[i], &ReadSet);
           ++ nCnt;
       }
   return nCnt;
}

/* -- Заполнение структуры адреса (семейство INet)
Вход:
   IP-адрес и порт в виде строк (вместо адреса может быть сетевое имя);
   их значения по умолчанию (используются если строки пустые или NULL)
Выход:
   TRUE если преобразование успешно, FALSE если ошибка
*/
bool FillInetAddr (struct sockaddr_in* pSockAddr,
```

```
    const char* pszIP, u_long nDefIP,
    const char* pszPort, u_short nDefPort)
{
    struct hostent* pHostEnt;
    u_long nIP, nPort;
//проверка и инициализация
    if (pSockAddr == NULL)
        return false;
    memset (pSockAddr, 0, sizeof(struct sockaddr_in));
    pSockAddr->sin_addr.s_addr = htonl (nDefIP);
    pSockAddr->sin_port = htons (nDefPort);
//семейство адреса
    pSockAddr->sin_family = AF_INET;
//ip-адрес
    if ((pszIP != NULL) && (pszIP[0] != '\0')) { //IP задан явно
        if (strcmp (pszIP, "255.255.255.255") == 0)
            nIP = INADDR_BROADCAST;
        else {
            if ((nIP = inet_addr (pszIP)) == INADDR_NONE) {
                if ((pHostEnt = gethostbyname (pszIP)) == NULL) {
                    fprintf (stderr, "Хост не опознан: %s\n", pszIP);
                    return false;
                }
                nIP = ((struct in_addr*)(pHostEnt->h_addr_list[0]))->s_addr;
            }
        }
        pSockAddr->sin_addr.s_addr = nIP;
    }
//номер порта
    if ((pszPort != NULL) && (pszPort[0] != '\0')) { //порт задан явно
        if (sscanf (pszPort, "%lu", &nPort) < 1) {
            fprintf (stderr, "Ошибочный номер порта: %s\n", pszPort);
            return false;
        }
        pSockAddr->sin_port = htons ((u_short)nPort);
    }
//завершение
    return true;
}

/* -- Закрыть одиночный сокет (со сбросом значения на невалидное)
Вход:
    Адрес переменной "сокет" для закрытия и сброса значения
*/
void CloseSocket (SOCKET* pSockToClose)
{
    if ((pSockToClose != NULL) && (*pSockToClose != INVALID_SOCKET)) {
        shutdown (*pSockToClose, SD_BOTH);
        closesocket (*pSockToClose);
        *pSockToClose = INVALID_SOCKET;
    }
}
```

Приложение 5

ПРИМЕР ИСПОЛЬЗОВАНИЯ ИНТЕРФЕЙСА WINDOWS MAILSLOTS

```
/* -                                                                    ---
- MSLOTSRV.C - демонстрационный "сервер" с использованием Mailslots   -
- Создает Mailslot с заданным именем,                                 -
- в цикле принимает поступающие в него сообщения и отображает их.     -
- Вызов: mslotsrv [<имя_mailslot>]                                    -
- Прерывание консольного приложения - <Ctrl-C>                        -
---                                                                  - */

#include <stdio.h>
#include <windows.h>

#define SLOT_TIMEOUT  1000

const char DEF_SLOT_NAME[] = "ExampleMailSlot";

char szFullSlotName [FILENAME_MAX];
char cBuf [1024];

int main (int argc, char** argv)
{
    HANDLE hMslot;
    DWORD nCnt;
    const char* pszSlotName = (argc > 1) ? argv[1] : DEF_SLOT_NAME;
//создание Mailslot
    sprintf (szFullSlotName, "\\\\.\\mailslot\\%s", pszSlotName);
    hMslot =
        CreateMailslot (szFullSlotName, //имя
            sizeof(cBuf)-1, SLOT_TIMEOUT, //параметры (лимиты)
            NULL //атрибуты безопасности - не используются
        );
    if (hMslot == INVALID_HANDLE_VALUE) { //сравнение не с NULL!
        printf ("Ошибка создания Mailslot: %s\n", pszSlotName);
        return -1;
    }
    printf ("Создан Mailslot: %s\n", pszSlotName);
//основной цикл сервера
    while (1) { //"вечный" цикл
        while (! ReadFile (hMslot, cBuf, sizeof(cBuf)-1, &nCnt, NULL));
        cBuf [nCnt] = '\0';
        printf ("Принято сообщение: %s\n", cBuf);
        Sleep (10); //кратковременно отдать управление
    }
//завершение - никогда не достигается
    CloseHandle (hMslot);
    return 0;
}
```

```
/* -                                                                    ---
- MSLOTCLN.C - демонстрационный "клиент" с Mailslots                    -
- Подключается к Mailslot с заданныем именем,                          -
- отсылает в него вводимые с консоли строки.                           -
- Вызов: mslotcln [<имя_mailslot>]                                     -
- (если имя не задано, используется имя по умолчанию в локальной системе)  -
- Прерывание цикла ввода строк - <Ctrl-Z>                              -
---                                                                  - */

#include <stdio.h>
#include <windows.h>

const char DEF_SLOT_NAME[] = "ExampleMailSlot";

char szFullSlotName [FILENAME_MAX];
char cBuf [1024];

int main (int argc, char** argv)
{
    HANDLE hMslot;
    DWORD nCnt;
//разбор командной строки
    if (argc>1) {
        if (strncmp (argv[1], "\\\\", 2) == 0) //наверно, полный путь
            strcpy (szFullSlotName, argv[1]);
        else //тогда в локальной системе
            sprintf (szFullSlotName, "\\\\.\\mailslot\\%s", argv[1]);
    }
    else //имя по умолчанию, в локальной системе
        sprintf (szFullSlotName, "\\\\.\\mailslot\\%s", DEF_SLOT_NAME);
//подключение к существующему Mailslot
    hMslot = CreateFile (szFullSlotName, //имя
        GENERIC_WRITE, FILE_SHARE_READ | FILE_SHARE_WRITE, //доступ
        NULL, //атрибуты безопасности - не используются
        OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, //параметры открытия
        NULL   //файл-шаблон - не используется
    );
    if (hMslot == INVALID_HANDLE_VALUE) { //сравнение не с NULL!
        printf ("Ошибка подключения к Mailslot: %s\n", szFullSlotName);
        return -1;
    }
//основной цикл клиента
    while (fgets (cBuf, sizeof(cBuf)-1, stdin) != NULL) {
        if (! WriteFile (hMslot, cBuf, strlen (cBuf), &nCnt, NULL))
            printf ("Ошибка записи в Mailslot: %s\n", szFullSlotName);
    }
//завершение - достигается после закрытия потока ввода (<Ctrl-Z>+<Enter>)
    CloseHandle (hMslot);
    return 0;
}
```

Приложение 6

ПРИМЕР ИСПОЛЬЗОВАНИЯ ИНТЕРФЕЙСА NETBIOS: ПОЛУЧЕНИЕ MAC-АДРЕСА

```
/* -                                                                    ---
- NBGETMAC.C - получение MAC-адреса сетевого адаптера вызовом NetBIOS  -
---                                                                     - */

#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <nb30.h>

typedef struct tagAStatusBuffer { // тип: буфер запроса состояния адаптера
      ADAPTER_STATUS AStatus;
      NAME_BUFFER Names [1]; //только одно имя
} TAStatusBuffer;

int main (void)
{
      TAStatusBuffer AStatusBuff; //буфер данных
      BYTE AAddr [6]; //массив MAC-адреса (48 бит)
      NCB Ncb; //NetBIOS control block
// начальный сброс адаптера
      memset (&Ncb, 0, sizeof(Ncb));
      Ncb.ncb_command = NCBRESET; //команда - Reset
      Ncb.ncb_lana_num = 0; //номер адаптера - по умолчанию 0
      Netbios (&Ncb); //вызов NetBIOS
      if (Ncb.ncb_retcode != NRC_GOODRET) { //проверка ошибки
            printf ("Ошибка инициализации: %02Xh\n", Ncb.ncb_retcode);
            return -1;
      }
// запрос состояния адаптера, в т.ч. его MAC-адреса
      memset (&Ncb, 0, sizeof(Ncb));
      Ncb.ncb_command = NCBASTAT; //команда - GetAdapterStatus
      Ncb.ncb_lana_num = 0; //номер адаптера - по умолчанию 0
      memset (&(Ncb.ncb_callname), ' ', 16); //заранее дополнить пробелами
      Ncb.ncb_callname[0] = '*'; //будет локальное имя
      Ncb.ncb_buffer = (BYTE*)&AStatusBuff; //буфер результата вызова
      Ncb.ncb_length = sizeof(AStatusBuff); //размер буфера
      Netbios (&Ncb); //вызов NetBIOS
      if (Ncb.ncb_retcode != NRC_GOODRET) { //проверка ошибки
            printf ("Ошибка запроса состояния: %02Xh\n", Ncb.ncb_retcode);
            return -1;
      }
// вывод результата
      memcpy (AAddr, AStatusBuff.AStatus.adapter_address, 6); //сохранить
      printf("MAC-адрес адаптера: %02X-%02X-%02X-%02X-%02X-%02X",
            AAddr[0], AAddr[1], AAddr[2], AAddr[3], AAddr[4], AAddr[5]);
      return 0;
}
```

Св. план 2005, поз. 51

Учебное издание

Сиротко Сергей Иванович,
Волосевич Алексей Александрович

КОМПЬЮТЕРНЫЕ СЕТИ

Учебное пособие
для студентов специальности I-31 03 04 «Информатика»
всех форм обучения

Редактор Н.В. Гриневич
Корректор Е.Н. Батурчик

Подписано в печать 02.02.2006. Формат 60x84 1/16. Бумага офсетная.
Гарнитура «Таймс». Печать ризографическая Усл. печ. л. 5,7.
Уч.-изд. л. 5,0. Тираж 150 экз. Заказ 457.

Издатель и полиграфическое исполнение: Учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники».
Лицензия на осуществление издательской деятельности № 02330/0056964 от 01.04.2004.
Лицензия на осуществление полиграфической деятельности № 0233/0131518 от 30.04.2004.
220013, Минск, П.Бровки, 6