Содержание

Глава 3. Создание эффективных и устойчивых сетевых программ 131

Посвящается Марии

Предисловие

В результате взрывного развития Internet, беспроводных видов связи и сетей соразмерно увеличилось число программистов и инженеров, занимающихся разработкой сетевых приложений. Программирование TCP/IP может показаться обманчиво простым. Интерфейс прикладного программирования (API) несложен. Даже новичок может взять шаблон клиента или сервера и создать на его основе работающее приложение.

К сожалению, нередко после весьма продуктивного начала неофиты начинают понимать, что все не так очевидно, а созданная ими программа оказывается и медленной, и нестабильной. В сетевом программировании есть множество «темных уголков» и трудно понимаемых деталей. Цель этой книги – ответить на возникающие вопросы и помочь разобраться с тонкостями программирования TCP/IP.

Прочитав данную книгу, вы научитесь преодолевать трудности сетевого программирования. Здесь будут рассмотрены многие вопросы, на первый взгляд, лишь отдаленно связанные с теми знаниями, которыми должен обладать программист сетевых приложений. Но без понимания таких деталей не разобраться в том, как сетевые протоколы взаимодействуют с приложением. Ранее казавшееся загадочным «поведение» приложения при ближайшем рассмотрении становится совершенно понятным, решение проблемы лежит на поверхности.

Книга построена несколько необычно. Типичные проблемы представлены в виде серии советов. Разбираясь с конкретным вопросом, вы будете переходить к изучению того или иного аспекта TCP/IP. К концу главы вы не только решите частную задачу, но и углубите понимание того, как работают и взаимодействуют с приложением протоколы TCP/IP.

Разбивка книги на отдельные советы в какой-то мере лишает текст логической последовательности. Чтобы помочь вам сориентироваться, в главу 1 помещен путеводитель, описывающий расположение материала. Получить общее представление об организации книги поможет также оглавление, в котором перечислены все советы. Поскольку каждый совет дан в повелительном наклонении, оглавление можно считать списком рецептов.

С другой стороны, такая организация материала позволяет использовать книгу в качестве справочника. Столкнувшись в повседневной работе с какой-либо проблемой, вы можете обратиться к соответствующему совету, чтобы вспомнить

ее решение. Многие темы затрагиваются в нескольких советах, иногда под разным углом зрения. Такое повторение помогает усвоению сложных концепций, проясняя мотивы тех или иных решений.

Аудитория

Данная книга предназначена, главным образом, начинающим и программистам среднего уровня, но даже опытные специалисты найдут в ней много полезного для себя. Хотя и предполагается, что читатель знаком с сетями и основами API на базе сокетов, в главе 1 приводится обзор элементарных вызовов API и их использования для создания примитивного клиента и сервера. В совете 4 более детально рассмотрены модели клиента и сервера, поэтому даже читатель с минимальной подготовкой сможет извлечь из представленного материала практическую пользу.

Почти все примеры написаны на языке C, безусловно, необходимы базовые навыки программирования на этом языке для понимания приведенных в книге программ. В совете 31 представлены некоторые примеры на языке Perl. Но, впрочем, предварительное знание Perl необязательно. Здесь встречаются и небольшие примеры на языках командных интерпретаторов (shell), но и для их понимания знакомства с shell-программированием не нужно.

Материал для изучения подан по возможности максимально независимо от платформы. За немногими исключениями, приводимые в примерах программы должны компилироваться и работать на любой платформе UNIX или Win32. Но программисты, которые используют системы, отличные от UNIX и Windows, тоже могут без особых трудностей применять примеры на своей платформе.

Принятые в книге соглашения

По ходу изложения приведены небольшие программы для иллюстрации различных аспектов исследуемой проблемы. Здесь будут использоваться следующие соглашения:

- текст, который набирает пользователь, печатается **полужирным моноширинным шрифтом**;
- текст, который выводят системы, печатается обычным моноширинным шрифтом;
- комментарии, не являющиеся частью ввода или вывода, печатаются *курсивным моноширинным шрифтом*.

Пример из совета 9:

```
bsd: $ tcprw localhost 9000
hello
получено сообщение 1     печатается после пятисекундной задержки
                         здесь сервер остановили
hello again
tcprw: ошибка вызова readline: Connection reset by peer (54)
bsd: $
```

Обратите внимание на приглашение командного интерпретатора, содержащее имя системы. Предыдущий пример исполнялся на машине с именем bsd.

В рамке дается описание вводимой в рассмотрение новой функции API – собственной или системного вызова. Стандартные системные вызовы обводятся сплошной рамкой:

```
#include <sys/socket.h>   /* UNIX */
#include <winsock2.h>     /* Windows */

int connect( SOCKET s, const struct sockaddr *peer, int peer_len );
```

Возвращаемое значение: 0 – нормально, –1 (UNIX) или не 0 (Windows) – ошибка.

Разработанные автором функции обведены пунктирной рамкой:

```
#include "etcp.h"

SOCKET tcp_server( char *host, char *port );
```

Возвращаемое значение: сокет в режиме прослушивания (в случае ошибки завершает программу).

Примечание *Таким образом отмечены попутные замечания и дополнительный материал. При первом чтении подобный материал можно пропускать.*

Наконец, URL подчеркивается: http://www.freebsd.org.

Исходные тексты и список исправлений

Исходные тексты всех встречающихся в книге примеров представлены на сайте издательства «ДМК-Пресс» http://www.dmk.ru. Вы можете загрузить их на свой компьютер и поэкспериментировать. На этом сайте находятся каркасы программ и код библиотечных функций.

Оформление

Мне очень нравится оформление книг Ричарда Стивенса*. Приступая к работе над этой книгой, я попросил у Рича разрешения скопировать его стиль. Рич со свойственным ему великодушием не только не возражал, но даже посодействовал этой «краже», прислав мне макросы для форматера GROFF, которыми он пользовался при наборе своих книг.

Если вам понравится это издание, то благодарить следует Рича. В противном случае загляните в любую из его книг, чтобы понять, к чему я стремился.

* Речь идет о трехтомном издании «TCP\IP Illustrated» и двухтомном «UNIX Network Programming». – *Прим. перев.*

Благодарности

Традиционно авторы книг благодарят свои семьи за поддержку во время работы над книгой, и теперь я знаю, почему. Эта работа не была бы закончена, если бы не помощь моей жены Марии. И эти слова – лишь жалкая попытка воздать ей долж-ное за дополнительные хлопоты и одинокие вечера.

Также ценнейшую поддержку оказали рецензенты. В ранних вариантах рукописи они нашли многочисленные ошибки, как технические, так и типографские, разъяснили то, что я неправильно понимал или не знал, предложили свежий подход. Хочется выразить благодарность Крису Клиланду (Chris Cleeland), Бобу Джиллигену (Bob Gilligan, FreeGate Corp.), Питеру Хэверлоку (Peter Haverlock, Nortel Networks), С. Ли Генри (S. Lee Henry, Web Publishing, Inc.), Мукешу Кэкеру (Mukesh Kacker, Sun Microsystems, Inc.), Барри Марголину (Barry Margolin, GTE Internetworking), Майку Оливеру (Mike Oliver, Sun Microsystems, Inc.), Юри Рацу (Uri Raz) и Ричу Стивенсу (Rich Stevens).

Необходимо поблагодарить редактора Карен Геттман (Karen Gettman), ведущего редактора Мэри Гарт (Mary Hart), координатора проекта Тиреллу Элбо (Tyrrell Albaugh) и корректора Кэт Охала (Cat Ohala). Мне было приятно с ними работать, они очень помогли начинающему автору.

Я приветствую любые замечания и предложения читателей. Пишите мне по электронному адресу, указанному ниже.

Йон Снейдер

Тампа, Флорида *jsnader@ix.netcom.com*

Декабрь, 1999 *http://www.netcom.com/~jsnader*

Глава 1. Введение

Цель этой книги – помочь программистам разных уровней – от начального до среднего – повысить свою квалификацию. Для получения статуса мастера требуется практический опыт и накопление знаний в конкретной области. Конечно, опыт приходит только со временем и практикой, но данная книга существенно пополнит багаж ваших знаний.

Сетевое программирование – это обширная область с большим выбором различных технологий для желающих установить связь между несколькими машинами. Среди них такие простые, как последовательная линия связи, и такие сложные, как системная сетевая архитектура (SNA) компании IBM. Но сегодня протоколы TCP/IP – наиболее перспективная технология построения сетей. Это обусловлено развитием Internet и самого распространенного приложения – Всемирной паутины (World Wide Web).

Примечание *Вообще-то, Web – не приложение. Но это и не протокол, хотя в ней используются и приложения (Web-браузеры и серверы), и протоколы (например, HTTP). Web – это самое популярное среди пользователей Internet применение сетевых технологий.*

Однако и до появления Web TCP/IP был распространенным методом создания сетей. Это открытый стандарт, и на его основе можно объединять машины разных производителей. К концу 90-х годов TCP/IP завоевал лидирующее положение среди сетевых технологий, видимо, оно сохранится и в дальнейшем. По этой причине в книге рассматриваются TCP/IP и сети, в которых он работает.

При желании совершенствоваться в сетевом программировании необходимо сначала овладеть некоторыми основами, чтобы в полной мере оценить, чем же вам предстоит заниматься. Рассмотрим несколько типичных проблем, с которыми сталкиваются начинающие. Многие из этих проблем – результат частичного или полного непонимания некоторых аспектов протоколов TCP/IP и тех API, с помощью которых программа использует эти протоколы. Такие проблемы возникают в реальной жизни и порождают многочисленные вопросы в сетевых конференциях.

Некоторые термины

За немногими исключениями, весь материал этой книги, в том числе примеры программ, предложен для работы в системах UNIX (32 и 64-разрядных) и системах, использующих API Microsoft Windows (Win32 API). Я не экспериментировал с 16-разрядными приложениями Windows. Но и для других платформ почти все остается применимым.

Желание сохранить переносимость привело к некоторым несообразностям в примерах программ. Так, программисты, работающие на платформе UNIX, неодобрительно отнесутся к тому, что для дескрипторов сокетов применяется тип `SOCKET` вместо привычного `int`. А программисты Windows заметят, что я ограничился только консольными приложениями. Все принятые соглашения описаны в совете 4.

По той же причине я обычно избегаю системных вызовов `read` и `write` для сокетов, так как Win32 API их не поддерживает. Для чтения из сокета или записи в него применяются системные вызовы `recv`, `recvfrom` или `recvmsg` для чтения и `send`, `sendto` или `sendmsg` для записи.

Одним из самых трудных был вопрос о том, следует ли включать в книгу материал по протоколу IPv6, который в скором времени должен заменить современную версию протокола IP (IPv4). В конце концов, было решено не делать этого. Тому есть много причин, в том числе:

- почти все изложенное в книге справедливо как для IPv4, так и для IPv6;
- различия, которые все-таки имеются, по большей части сосредоточены в тех частях API, которые связаны с адресацией;
- книга представляет собой квинтэссенцию опыта и знаний современных сетевых программистов, а реального опыта работы с протоколом IPv6 еще не накоплено.

Поэтому, если речь идет просто об IP, то подразумевается IPv4. Там, где упоминается об IPv6, об этом написано.

И, наконец, я называю восемь бит информации *байтом*. В сетевом сообществе принято называть такую единицу *октетом* – по историческим причинам. Когда-то размер байта зависел от платформы, и не было единого мнения о его точной длине. Чтобы избежать неоднозначности, в ранней литературе по сетям и был придуман термин *октет*. Но сегодня все согласны с тем, что длина байта равна восьми битам [Kernighan and Pike 1999], так что употребление этого термина можно считать излишним педантизмом.

Примечание *Однако утверждения о том, что длина байта равна восьми битам, время от времени все же вызывают споры в конференциях Usenet: «Ох уж эта нынешняя молодежь! Я в свое время работал на машине Баста-6, в которой байт был равен пяти с половиной битам. Так что не рассказывайте мне, что в байте всегда восемь бит».*

Путеводитель по книге

Ниже будут рассмотрены основы API сокетов и архитектура клиент-сервер, свойственная приложениям, в которых используется TCP/IP. Это тот фундамент, на котором вы станете возводить здание своего мастерства.

В главе 2 обсуждаются некоторые заблуждения по поводу TCP/IP и сетей вообще. В частности, вы узнаете, в чем разница между протоколами, требующими логического соединения, и протоколами, не нуждающимися в нем. Здесь будет рассказано об IP-адресации и подсетях (эта концепция часто вызывает недоумение), о бесклассовой междоменной маршрутизации (Classless Interdomain Routing – CIDR)

и преобразовании сетевых адресов (Network Address Translation – NAT). Вы увидите, что TCP в действительности не гарантирует доставку данных. И нужно быть готовым к некорректным действиям как пользователя, так и программы на другом конце соединения. Кроме того, приложения будут по-разному работать в глобальной (WAN) и локальной (LAN) сетях.

Следует напомнить, что TCP – это *потоковый* протокол, и разъяснить его значение для программистов. Вы также узнаете, что TCP автоматически не обнаруживает потерю связи, почему это хорошо и что делать в этой ситуации.

Вам будет понятно, почему API сокетов всегда следует предпочитать API на основе интерфейса транспортного уровня (Transport Layer Interface – TLI) и транспортному интерфейсу X/Open (X/Open Transport Interface – XTI). Кроме того, я объясню, почему не стоит слишком уж серьезно воспринимать модель открытого взаимодействия систем (Open Systems Interconnection – OSI). TCP – очень эффективный протокол с отличной производительностью, так что обычно не нужно дублировать его функциональность с помощью протокола UDP.

В главе 2 разработаны каркасы для нескольких видов приложений TCP/IP и на их основе построена библиотека часто используемых функций. Каркасы и библиотека позволяют писать приложения, не заботясь о преобразовании адресов, управлении соединением и т.п. Если каркас готов, то вряд ли следует срезать себе путь, например, «зашив» в код адреса и номера портов или опустив проверку ошибок.

Каркасы и библиотека используются в книге для построения тестов, небольших примеров и автономных приложений. Часто требуется всего лишь добавить пару строк в один из каркасов, чтобы создать специализированную программу или тестовый пример.

В главе 3 подробно рассмотрены некоторые вопросы, на первый взгляд кажущиеся тривиальными. Например, что делает операция записи в контексте TCP. Вроде бы все очевидно: записывается в сокет n байт, а TCP отсылает их на другой конец соединения. Но вы увидите, что иногда это происходит не так. В протоколе TCP есть сложный свод правил, определяющих, можно ли посылать данные немедленно и, если да, то сколько. Чтобы создавать устойчивые и эффективные программы, необходимо усвоить эти правила и их влияние на приложения.

То же относится к чтению данных и завершению соединения. Вы изучите эти операции и разберетесь, как нужно правильно завершать соединение, чтобы не потерять информацию. Здесь будет рассмотрена и операция установления соединения `connect`: когда при ее выполнении возникает тайм-аут и как она используется в протоколе UDP.

Будет рассказано об имеющимся в системе UNIX суперсервере `inetd`, упрощающим написание сетевых приложений. Вы научитесь пользоваться программой `tcpmux`, которая избавляет от необходимости назначать серверам хорошо известные порты. Узнаете, как работает `tcpmux`, и сможете создать собственную версию для систем, где это средство отсутствует.

Кроме того, здесь подробно обсуждаются такие вопросы, как состояние TIME-WAIT, алгоритм Нейгла, выбор размеров буферов и правильное применение опции `SO_REUSEADDR`. Вы поймете, как сделать свои приложения событийно-управляемыми и создать отдельный таймер для каждого события. Будут описаны некоторые

типичные ошибки, которые допускают даже опытные программисты, и приемы повышения производительности.

Наконец, вы познакомитесь с некоторыми языками сценариев, используемыми при программировании сетей. С их помощью можно легко и быстро создавать полезные сетевые утилиты.

Глава 4 посвящена двум темам. Сначала будет рассмотрено несколько инструментальных средств, необходимых каждому сетевому программисту. Показано, как использовать утилиту `ping` для диагностики простейших неисправностей. Затем рассказывается о сетевых анализаторах пакетов (sniffer) вообще и программе `tcpdump` в частности. В этой главе дано несколько примеров применения `tcpdump` для диагностики сетевых проблем. С помощью программы `traceroute` исследуется маленькая часть Internet.

Утилита `ttcp`, в создании которой принимал участие Майк Муусс (Mike Muuss) – автор программы `ping`, является полезным инструментом для изучения производительности сети и влияния на нее тех или иных параметров TCP. Будут продемонстрированы некоторые методы диагностики. Еще одна бесплатная инструментальная программа `lsof` необходима в ситуации, когда нужно сопоставить сетевые соединения с процессами, которые их открыли. Очень часто `lsof` предоставляет информацию, получение которой иным способом потребовало бы поистине героических усилий.

Много внимания уделено утилите `netstat` и той разнообразной информации, которую можно получить с ее помощью, а также программам трассировки системных вызовов, таким как `ktrace` и `truss`.

Обсуждение инструментов диагностики сетей завершается построением утилиты для перехвата и отображения датаграмм протокола ICMP (протокол контроля сообщений в сети Internet). Она не только вносит полезный вклад в ваш набор инструментальных средств, но и иллюстрирует использование простых сокетов (raw sockets).

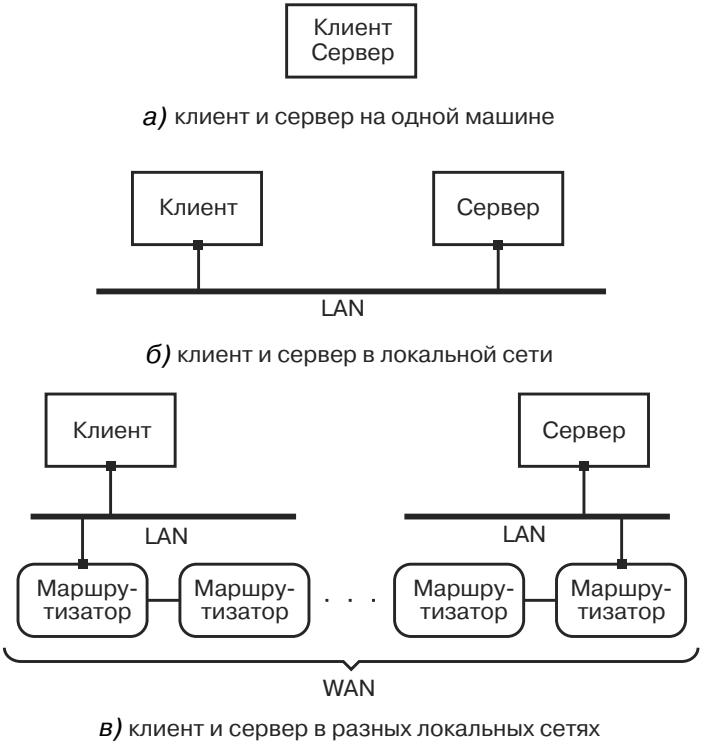
Во второй части главы 4 описаны дополнительные ресурсы для пополнения знаний о TCP/IP и сетях. Я познакомлю вас с замечательными книгами Ричарда Стивенса, источниками исходных текстов, и собранием документов RFC (предложений для обсуждения), размещенных на сервере проблемной группы проектирования Internet (Internet Engineering Task Force – IETF) и в конференциях Usenet.

Архитектура клиент-сервер

Хотя постоянно говорится о *клиентах* и *серверах*, не всегда очевидно, какую роль играет конкретная программа. Иногда программы являются равноправными участниками обмена информацией, нельзя однозначно утверждать, что одна программа обслуживает другую. Однако в случае с TCP/IP различие более четкое. Сервер прослушивает порт, чтобы обнаружить входящие TCP-соединения или UDP-датаграммы от одного или нескольких клиентов. С другой стороны, можно сказать, что клиент – это тот, кто начинает диалог первым.

В книге рассмотрены три типичных случая архитектуры клиент-сервер, показанные на рис. 1.1. В первом случае клиент и сервер работают на одной машине

(рис. 1.1а). Это самая простая конфигурация, поскольку нет физической сети. Посылаемые данные, передаются стеку TCP/IP, но не помещаются в выходную очередь сетевого устройства, а закольцовываются системой и возвращаются обратно в стек, но уже в качестве принятых данных.



а) клиент и сервер на одной машине

б) клиент и сервер в локальной сети

в) клиент и сервер в разных локальных сетях

Рис. 1.1
Типичные примеры архитектуры клиент-сервер

На этапе разработки такое размещение клиента и сервера дает определенные преимущества, даже если в реальности они будут работать на разных машинах. Во-первых, проще оценить производительность обеих программ, так как сетевые задержки исключаются. Во-вторых, этот метод создает идеальную лабораторную среду, в которой пакеты не пропадают, не задерживаются и всегда приходят в правильном порядке.

Примечание *По крайней мере, почти всегда. Как вы увидите в совете 7, даже в этой среде можно создать такую нагрузку, что UDP-датаграммы будут пропадать.*

И, наконец, разработку вести проще и удобнее, когда можно все отлаживать на одной машине.

Разумеется, даже в условиях промышленной эксплуатации вполне возможно, что клиент и сервер будут работать на одном компьютере. В совете 26 описана такая ситуация.

Во втором примере конфигурации (рис. 1.1б) клиент и сервер работают на разных машинах, но в пределах одной локальной сети. Здесь имеет место реальная

сеть, но условия все же близки к идеальным. Пакеты редко теряются и практически всегда приходят в правильном порядке. Такая ситуация очень часто встречается на практике. Причем некоторые приложения предназначены для работы только в такой среде.

Типичный пример – сервер печати. В небольшой локальной сети может быть только один такой сервер, обслуживающий несколько машин. Одна машина (или сетевое программное обеспечение на базе TCP/IP, встроенное в принтер) выступает в роли сервера, который принимает запросы на печать от клиентов на других машинах и ставит их в очередь к принтеру.

В третьем примере (рис. 1.1в) клиент и сервер работают на разных компьютерах, связанных глобальной сетью. Этой сетью может быть Internet или корпоративная Intranet, но главное – приложения уже не находятся внутри одной локальной сети, так что на пути IP-датаграмм есть, по крайней мере, один маршрутизатор.

Такое окружение может быть более «враждебным», чем в первых двух случаях. По мере роста трафика в глобальной сети начинают переполняться очереди, в которых маршрутизатор временно хранит поступающие пакеты, пока не отправит их адресату. А когда в очереди больше нет места, маршрутизатор отбрасывает пакеты. В результате клиент должен передавать пакеты повторно, что приводит к появлению дубликатов и доставке пакетов в неправильном порядке. Эти проблемы возникают довольно часто, как вы увидите в совете 38.

О различиях между локальными и глобальными сетями будет рассказано в совете 12.

Элементы API сокетов

В этом разделе кратко рассмотрены основы API сокетов и построены простейшие клиентское и серверное приложения. Хотя эти приложения очень схематичны, на их примере проиллюстрированы важнейшие характеристики клиента и сервера TCP.

Начнем с вызовов API, необходимых для простого клиента. На рис. 1.2 показаны функции, применяемые в любом клиенте. Адрес удаленного хоста задается с помощью структуры `sockaddr_in`, которая передается функции `connect`.

Первое, что вы должны сделать, – это получить сокет для логического соединения. Для этого предназначен системный вызов `socket`.

```
#include <sys/socket.h>  /* UNIX */
#include <winsock2.h>    /* Windows */

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

Возвращаемое значение: дескриптор сокета в случае успеха; –1 (UNIX) или `INVALID_SOCKET` (Windows) – ошибка.

API сокетов не зависит от протокола и может поддерживать разные *адресные домены*. Параметр `domain` – это константа, указывающая, какой домен нужен сокету.

Чаще используются домены AF_INET (то есть Internet) и AF_LOCAL (или AF_UNIX). В книге рассматривается только домен AF_INET. Домен AF_LOCAL применяется для межпроцессного взаимодействия (IPC) на одной и той же машине.

Примечание *Существуют разногласия по поводу того, следует ли обозначать константы доменов* AF_* *или* PF_*. *Сторонники* PF_* *указывают на их происхождение от уже устаревших вариантов вызова* socket *в системах 4.1c/2.8/2.9BSD. И, кроме того, они считают, что PF означает protocol family (семейство протоколов). Сторонники же* AF_* *говорят, что в коде ядра, относящемся к реализации сокетов, параметр* domain *сравнивается именно с константами* AF_*. *Но, поскольку оба набора констант определены одинаково – в действительности одни константы просто выражаются через другие, – на практике можно употреблять оба варианта.*

С помощью параметра *type* задается тип создаваемого сокета. Чаще встречаются следующие значения (а в этой книге только такие) сокетов:

- SOCK_STREAM – обеспечивают надежный дуплексный протокол на основе установления логического соединения. Если говорится о семействе протоколов TCP/IP, то это TCP;
- SOCK_DGRAM – обеспечивают ненадежный сервис доставки датаграмм. В рамках TCP/IP это будет протокол UDP;
- SOCK_RAW –предоставляют доступ к некоторым датаграммам на уровне протокола IP. Они используются в особых случаях, например для просмотра всех ICMP-сообщений.

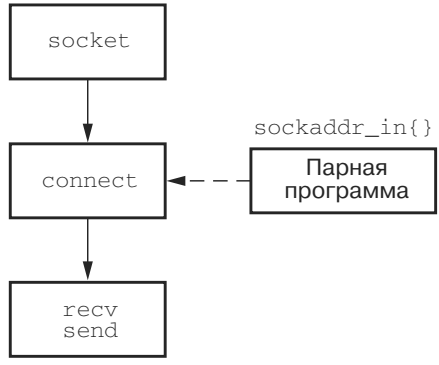


Рис. 1.2. Основные вызовы API сокетов для клиентов

Параметр *protocol* показывает, какой протокол следует использовать с данным сокетом. В контексте TCP/IP он обычно неявно определяется типом сокета, поэтому в качестве значения задают 0. Иногда, например в случае простых (raw) сокетов, имеется несколько возможных протоколов, так что нужный необходимо задавать явно. Об этом будет рассказано в совете 40.

Для самого простого TCP-клиента потребуется еще один вызов API сокетов, обеспечивающий установление соединения:

```
#include <sys/socket.h>  /* UNIX */
#include <winsock2.h>    /* Windows */

int connect( SOCKET s, const struct sockaddr *peer, int peer_len );
```

Возвращаемое значение: 0 – нормально, –1 (UNIX) или не 0 (Windows) – ошибка.

Параметр *s* – это дескриптор сокета, который вернул системный вызов socket. Параметр *peer* указывает на структуру, в которой хранится адрес удаленного хоста и некоторая дополнительная информация. Для домена AF_INET – это структура типа sockaddr_in. Ниже вы увидите, как она заполняется. Параметр *peer_len* содержит размер структуры в байтах, на которую указывает *peer*.

После установления соединения можно передавать данные. В ОС UNIX вы должны обратиться к системным вызовам read и write и передать им дескриптор сокета точно так же, как передали бы дескриптор открытого файла. Увы, как уже говорилось, в Windows эти системные вызовы не поддерживают семантику сокетов, поэтому приходится пользоваться вызовами recv и send. Они отличаются от read и write только наличием дополнительного параметра.

```
#include <sys/socket.h>  /* UNIX */
#include <winsock2.h>    /* Windows */

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

int send( SOCKET s, const void *buf, size_t len, int flags );
```

Возвращаемое значение: число принятых или переданных байтов в случае успеха или –1 в случае ошибки.

Параметры *s*, *buf* и *len* означают то же, что и для вызовов read и write. Значение параметра *flags* в основном зависит от системы, но и UNIX, и Windows поддерживают следующие флаги:

- MSG_OOB – следует послать или принять срочные данные;
- MSG_PEEK – используется для просмотра поступивших данных без их удаления из приемного буфера. После возврата из системного вызова данные еще могут быть получены при последующем вызове read или recv;
- MSG_DONTROUTE – сообщает ядру, что не надо выполнять обычный алгоритм маршрутизации. Как правило, используется программами маршрутизации или для диагностических целей.

При работе с протоколом TCP вам ничего больше не понадобится. Но при работе с UDP нужны еще системные вызовы recvfrom и sendto. Они очень похожи на recv и send, но позволяют при отправке датаграммы задать адрес назначения, а при приеме – получить адрес источника.

```
#include <sys/socket.h>  /* UNIX */
#include <winsock2.h>    /* Windows */

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

int sendto( SOCKET s, const void *buf, size_t len, int flags,
      const struct sockaddr *to, int tolen);
```

Возвращаемое значение: число принятых или переданных байтов в случае успеха или –1 при ошибке.

Первые четыре параметра – *s*, *buf*, *len* и *flags* – такие же, как в вызовах recv и send. Параметр *from* в вызове recvfrom указывает на структуру, в которую ядро помещает адрес источника пришедшей датаграммы. Длина этого адреса хранится в целом числе, на которое указывает параметр *fromlen*. Обратите внимание, что *fromlen* – это *указатель* на целое.

Аналогично параметр *to* в вызове sendto указывает на адрес структуры, содержащей адреса назначения датаграммы, а параметр *tolen* – длина этого адреса. Заметьте, что *to* – это целое, а не указатель.

В листинге 1.1 приведен пример простого TCP-клиента.

Листинг 1.1. Простейший TCP-клиент

simplec.c

```
1 #include <sys/types.h>
2 #include <sys/socket.h>
3 #include <netinet/in.h>
4 #include <arpa/inet.h>
5 #include <stdio.h>

6 int main( void )
7 {
8     struct sockaddr_in peer;
9     int s;
10    int rc;
11    char buf[ 1 ];

12    peer.sin_family = AF_INET;
13    peer.sin_port = htons( 7500 );
14    peer.sin_addr.s_addr = inet_addr( "127.0.0.1" );

15    s = socket( AF_INET, SOCK_STREAM, 0 );
16    if ( s < 0 )
17    {
18       perror( "ошибка вызова socket" );
19       exit( 1 );
20    }

21    rc = connect( s, ( struct sockaddr * )&peer, sizeof( peer ) );
22    if ( rc )
23    {
24       perror( "ошибка вызова connect" );
25       exit( 1 );
26    }

27    rc = send( s, "1", 1, 0 );
28    if ( rc <= 0 )
29    {
30       perror( "ошибка вызова send" );
31       exit( 1 );
32    }

33    rc = recv( s, buf, 1, 0 );
34    if ( rc <= 0 )
```

```
35       perror( "ошибка вызова recv" );
36    else
37       printf( "%c\n", buf[ 0 ] );
38    exit( 0 );
39 }
```
simplec.c

Клиент в листинге 1.1 написан как UNIX-программа, чтобы не было сложностей, связанных с переносимостью и Windows-функцией `WSAStartup`. В совете 4 сказано, что в основном эти сложности можно скрыть в заголовочном файле, но сначала надо подготовить некоторые механизмы. Пока ограничимся более простой моделью UNIX.

Подготовка адреса сервера

12-14 Заполняем структуру `sockaddr_in`, записывая в ее поля номер порта (7500) и адрес. 127.0.0.1 – это возвратный адрес, который означает, что сервер находится на той же машине, что и клиент.

Получение сокета и соединение с сервером

15-20 Получаем сокет типа `SOCK_STREAM`. Как было отмечено выше, протокол TCP, будучи потоковым, требует именно такого сокета.

21-26 Устанавливаем соединение с сервером, обращаясь к системному вызову `connect`. Этот вызов нужен, чтобы сообщить ядру адрес сервера.

Отправка и получение одного байта

27-38 Сначала посылаем один байт серверу, затем читаем из сокета один байт и записываем полученный байт в стандартный вывод и завершаем сеанс.

Прежде чем тестировать клиента, необходим сервер. Вызовы API сокетов для сервера немного иные, чем для клиента. Они показаны на рис. 1.3.

Сервер должен быть готов к установлению соединений с клиентами. Для этого он обязан прослушивать известный ему порт с помощью системного вызова `listen`. Но предварительно необходимо привязать адрес интерфейса и номер порта к прослушивающему сокету. Для этого предназначен вызов `bind`:

```
#include <sys/socket.h>  /* UNIX */
#include <winsock2.h>    /* Windows */

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

Возвращаемое значение: 0 – нормально, –1 (UNIX) или `SOCKET_ERROR` (Windows) – ошибка.

Параметр *s* – это дескриптор прослушивающего сокета. С помощью параметров *name* и *namelen* передаются порт и сетевой интерфейс, которые нужно прослушивать. Обычно в качестве адреса задается константа `INADDR_ANY`. Это означает, что будет принято соединение, запрашиваемое по любому интерфейсу. Если хосту с несколькими сетевыми адресами нужно принимать соединения только по