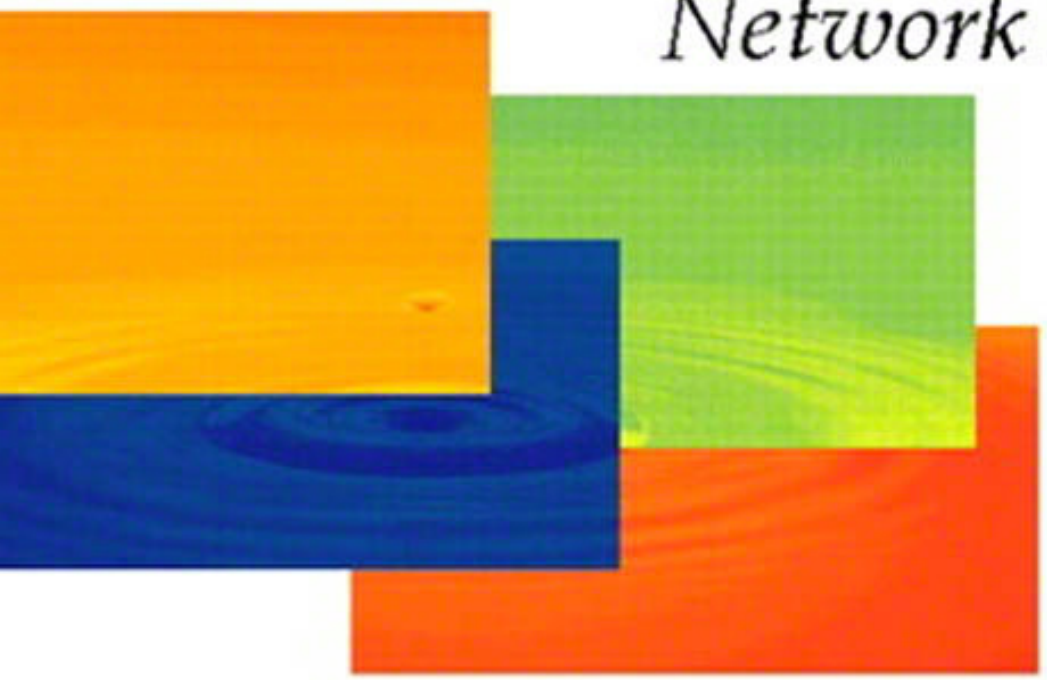
Effective TCP/IP Programming

44 Tips to Improve Your Network Programs



Jon C. Snader

Effective TCP/IP Programming

This page intentionally left blank

Effective TCP/IP Programming

44 Tips to Improve Your Network Programs

Jon C. Snader

ADDISON–WESLEY

Boston • San Francisco • New York • Toronto • Montreal
London • Munich • Paris • Madrid
Capetown • Sidney • Tokyo • Singapore • Mexico City

Pearson Education Corporate Sales Division
One Lake Street
Upper Saddle River, NJ 07458
(800) 382-3419
corpsales@pearsontechgroup.com

Visit Addison-Wesley on the Web at www.awprofessional.com

Library of Congress Cataloging-in-Publication Data
Snader, Jon C., 1944–
Effective TCP/IP programming : 44 tips to improve your network programs / Jon C. Snader.
p. cm.
Includes bibliographical references and index.
ISBN 0-201-61589-4
1. Internet programming. 2. TCP/IP (Computer network protocol) I. Title.
QA76.625 S63 2000
005.7'1376—dc21 00–026658

ISBN 0-201-61589-4
Text printed in the United States on recyled paper at Hamilton in Castleton, New York.

Eighth printing, February 2006

For Maria

This page intentionally left blank

Contents

This page intentionally left blank

Preface

Introduction

The explosive growth of the Internet, wireless communications, and networking in general has led to a corresponding growth in the number of programmers and engineers writing networking applications. TCP/IP programming can seem seductively simple. The Application Programming Interface (API) is straightforward, and even the newest beginner can take a template for a client or server and flesh it out to a working application.

Often, however, after an initial surge of productivity, neophytes begin to bog down in details and find that their applications suffer from performance or robustness problems. Network programming is a field full of dark corners and often misunderstood details. This book sheds light into those corners and helps replace misunderstanding with an appreciation for the often subtle points of TCP/IP programming.

After finishing this book, you should have a thorough understanding of many of the trouble spots in network programming. In the text we examine many areas that seem only peripherally connected to the core knowledge that a working network programmer is expected to have. We will see, however, that by gaining an understanding of these minutiae, we also gain an appreciation for how the inner workings of the network protocols can interact with our applications. Armed with this insight, application behavior that previously seemed bewildering becomes understandable, and the solutions to problems become clear.

The organization of the text is a little unusual. We examine common problems one at a time in a series of tips. During the process of studying a particular trouble spot, we usually explore some aspect of TCP/IP programming in depth. When we finish, we will not only have identified and dealt with common problems, we will also have a fairly comprehensive understanding of how the TCP/IP protocols work and interact with our applications.

The organization of the text into tips leads to a certain disjointedness. To help guide you, Chapter 1 contains a road map that explains the material covered in each chapter and how it all hangs together. The Contents, which lists each tip, will give you a sense of the text's organization. Because the title of each tip is in the form of an imperative, we can also think of the Contents as a list of network programming precepts.

On the other hand, this organization into tips makes the text more useful as a handbook. When we run into a problem in our day-to-day work, it is easy to revisit the appropriate tip to refresh our understanding of that particular problem area. You will find that many topics are visited in more than one tip, sometimes from a slightly different viewpoint. This repetition helps solidify the concepts and makes them seem more natural.

Readers

This text is written primarily for the advanced beginner or intermediate network programmer, but more experienced readers should find it useful as well. Although the reader is presumed to have a familiarity with networking and the basic sockets API, Chapter 1 contains a review of the elementary socket calls and uses them to build a primitive client and server. Tip 4 (Develop and Use Application Skeletons) revisits the various client and server models in more detail, so even a reader with minimal background should be able to understand and benefit from the text.

Almost all of the examples are written in the C language, so a reasonably good understanding of elementary C programming is necessary to get the full benefit from the programs in the text. In Tip 31 (Remember That All the World's Not C) we show some examples written in Perl, but no knowledge of Perl is assumed. Similarly, there are a few examples of small shell programs, but again, no previous experience with shell programming is necessary to understand them.

The examples and text attempt to be platform neutral. The examples are, with a few exceptions, intended to compile and run under any UNIX or Win32 system. Even those programmers not working on a UNIX or Windows system should have little trouble porting the examples to work on their platforms.

Typographical Conventions

During our explorations we will build and run many small programs designed to illustrate some aspect of the problem we are examining. When we show interactive input and output, we will use the following conventions:

- Text that we type is set in **`bold Courier`**.
- Text that the system outputs is set in `plain Courier`.
- Comments that are not part of the actual input or output are set in *italics*.

The following is an example from Tip 9:

```
bsd: $ tcprw localhost 9000
hello
received message 1                    this is printed after a 5-second delay
                                      the server is killed here
hello again
tcprw: readline failed: Connection reset by peer (54)
bsd: $
```

Notice that we include the name of the system in the shell prompt. In the previous example, we see that `tcprw` was run on the host named bsd.

When we introduce a new API function, either our own or one from the standard system calls, we enclose it in a box. The standard system calls are enclosed in a solid box, like this:

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

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

Returns: 0 on success, -1 (UNIX) or nonzero (Windows) on failure

API functions that we develop are enclosed in a dashed box, like this:

```
#include "etcp.h"

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

Returns: a listening socket (terminates on error)

> Parenthetical remarks and material that is commonly placed in footnotes are set in smaller type and are indented like this paragraph. Often, this material can be skipped on a first reading.

Lastly, we set off URLs with angle brackets like this:

<http://www.freebsd.org>

Source Code and Errata Availability

The source code for all the examples in this book are available electronically on the Web from <http://www.netcom.com/~jsnader>. Because the examples in the book were typeset directly from the program files, you can obtain and experiment with them on your own. The skeletons and library code are also available for your use.

A current list of errata is available from the same Web site.

Rich Stevens

I am a great admirer of the "look and feel" of Rich Stevens' books. When I began this project, I wrote to Rich and asked if he would mind if I stole his layout style. Rich, with his characteristic generosity, not only replied that he had no objections, but even aided and abetted the theft by sending me a copy of the Groff macros that he used for typesetting his books.

To the extent that you find the appearance and layout of this book pleasing, we have Rich to thank. If you find something in the layout that doesn't seem right, or is not pleasing, take a look at one of Rich's books to see what I was aiming for.

Colophon

I produced camera-ready copy for this book using James Clark's Groff package, and Rich Stevens' modified `ms` macros. The illustrations were prepared with `gpic` (including macros written by Rich Stevens and Gary Wright), the tables with `gtbl`, and the (limited) mathematical notation with `geqn`. The index was produced with the help of a set of `awk` scripts written by Jon Bentley and Brian Kernighan. The example code was included in the text directly from the program files with the help of Dave Hanson's `loom` program.

Acknowledgments

It is customary for authors to thank their families for their help and support during the writing of a book, and now I know why. This book literally would not have been possible were it not for my wife Maria. Without her taking on an even larger share than normal of my "50 percent," there would have been no time to work on and complete this book. These words are woefully inadequate thanks for the extra chores and lonely nights that she endured.

Another extraordinarily valuable asset to a writer are the reviewers who struggle through the early drafts. The technical reviewers for this text found numerous errors, both technical and typographical, corrected my misunderstandings, suggested fresh approaches, told me things I hadn't known before, and occasionally even offered a kind word by way of encouragement. I would like to thank Chris Cleeland, Bob Gilligan (FreeGate Corp.), Peter Haverlock (Nortel Networks), S. Lee Henry (Web Publishing, Inc.), Mukesh Kacker (Sun Microsystems, Inc.), Barry Margolin (GTE Internetworking), Mike Oliver (Sun Microsystems, Inc.), Uri Raz, and Rich Stevens for their hard work and suggestions. The text is a much better work on account of it.

Finally, I would like to thank my editor, Karen Gettman, project editor, Mary Hart, production coordinator, Tyrrell Albaugh, and copy editor, Cat Ohala. They are a joy to work with and have been incredibly helpful to this first-time author.

I welcome readers' comments, suggestions, and corrections. Feel free to send me email at the address below.

Jon C. Snader
jsnader@ix.netcom.com
http://www.netcom.com/~jsnader

Tampa, Florida
December 1999

This page intentionally left blank

1

Introduction

The purpose of this book is to help the advanced beginner or intermediate network programmer make the move to journeyman, and eventually master, status. The transition to master status is largely a matter of experience and the accretion of specific, if sometimes obscure, bits of knowledge. Nothing but time and practice will provide the experience, but this book can help with the knowledge.

Network programming is a large field, of course, and there are many contenders for the choice of networking technology when we wish to enable communication between two or more machines. The possibilities range from the simple, such as serial links, to the complex, such as IBM's System Network Architecture. Today it is increasingly clear that the TCP/IP protocol suite is the technology of choice for building networks. This is driven largely by the Internet and its most popular application: the World Wide Web.

> The Web isn't really an application, of course. It's not a protocol either, although it uses both applications (Web browsers and servers) and protocols (HTTP, for example). What we mean is that the Web is the most popular user-visible application of networking technology that runs on the Internet.

Even before the advent of the Web, however, TCP/IP was a popular method of creating networks. Because it was an open standard and could interconnect machines from different vendors, it was used increasingly to build networks and network applications. By the end of the 1990s, TCP/IP had become the dominant networking technology, and it is likely to remain so for some time. Because of this, we concentrate on TCP/IP and the networks on which it runs.

If we wish to master network programming we must first absorb some of the background knowledge needed to come to a fuller understanding and appreciation of our craft. Our plan for acquiring this knowledge is to look at several common problems that beginning network programmers face. Many of these problems are the results of misconceptions about or incomplete understanding of certain facets of the TCP/IP

protocols and the APIs used to communicate with them. All of these problems are real. They are a constant source of confusion and the frequent subjects of questions in the networking news groups.

A Few Conventions

The textual material and programs in this book are, with a few obvious exceptions, intended to be portable between UNIX (32 or 64 bit) and the Microsoft Win32 API. We have made no attempt to deal with 16-bit Windows applications. That said, almost all the material and many of the programs are appropriate in other environments as well.

This desire for portability has led to a few infelicities in the code examples. UNIX programmers will look askance at the notion of socket descriptors being defined as type `SOCKET` instead of type `int` for instance, and Windows programmers will notice that we are relentlessly committed to console applications. These conventions are described in Tip 4.

Similarly, we avoid, for the most part, using `read` and `write` on sockets because the Win32 API does not support these system calls on sockets. We will often speak of a read of or write to a socket, but we are speaking generically. By "read" we mean `recv`, `recvfrom`, or `recvmsg`; by "write" we mean `send`, `sendto`, or `sendmsg`. When we mean the read system call specifically, we will set it in Courier font as `read`, and similarly for write.

One of the hardest decisions to make was whether to include material on IPv6, the coming replacement for the current version (IPv4) of the Internet Protocol (IP). In the end, the decision was not to cover IPv6. There were many reasons for this, including

- Almost everything in this text remains true whether IPv4 or IPv6 is being used
- The differences that do appear tend to be localized in the addressing portions of the API
- This is a book based largely on the shared experiences and knowledge of journeyman network programmers and we really have no experience with IPv6 yet because implementations are just now becoming widely available

Therefore, when we speak of *IP* without qualification, we are speaking of IPv4. In those places where we do mention IPv6, we are careful to refer to it as *IPv6* explicitly.

Lastly, we refer to an 8 bit unit of data as a *byte*. It is common in the networking community to refer to such data as *octets*. The reasons for this are historical. It used to be that the size of a byte depended on the platform, and there was no agreement regarding its exact size. To avoid ambiguity, the early networking literature made the size explicit by coining the term *octet*. Today, there is universal agreement that a byte is 8 bits long [Kernighan and Pike 1999], and the use of octet seems needlessly pedantic.

> Despite this, one can still occasionally see the suggestion that bytes are 8 bits long, provoking a Usenet flame along the lines of, "Kids today! When I was a lad I worked on the Frumbaz-6, which had a byte length of 5 and a half bits. Don't tell me a byte always has 8 bits."

Road Map to the Rest of the Book

In the rest of this chapter we review the basic sockets API and client-server architecture used when writing TCP/IP applications. This is the base on which we must build as we master our craft.

Chapter 2 discusses some basic facts and misconceptions about TCP/IP and networking. We learn, for example, the difference between a connection-oriented protocol and a connectionless one. We explore IP addressing and the often confusing subjects of subnets, *classless interdomain routing* (CIDR), and *network address translation* (NAT). We see that TCP does *not*, in fact, guarantee delivery of data, that we must be prepared for misbehavior on the part of our peers or users, and that our applications are likely to behave differently on a *wide area network* (WAN), than they do on a *local area network* (LAN).

We are reminded that TCP is a *stream* protocol, and what that means for us as programmers. Similarly, we learn that TCP does not detect loss of connectivity automatically, why this is a good thing, and what we can do about it.

We see why the sockets API should almost always be preferred to the Transport Layer Interface/X/Open Transport Interface (TLI/XTI), and why we shouldn't take the Open Systems Interconnection model (OSI) too seriously. We also see that TCP is a remarkably efficient protocol with excellent performance, and that it doesn't usually make sense to duplicate its functionality using UDP.

In Chapter 2 we also develop skeleton code for several TCP/IP application models, and use it to build a library of commonly needed functions. These skeletons and the library are important because they allow us to write applications without having to worry about the routine chores such as address conversion, connection management, and so on. Because we have these skeletons available, we are less tempted to take shortcuts, such as hard-coding addresses and port numbers, or ignoring error returns.

We use these skeletons and library repeatedly throughout the text to build test cases, example code, and even stand-alone working applications. Often we can build a special-purpose application or test case merely by adding a few lines of code to one of our skeletons.

In Chapter 3 we examine several seemingly trivial subjects in depth. For example, we start with a discussion of the TCP write operation and what it does. At first blush this seems simple: We write *n* bytes and TCP sends them to our peer. As we shall see, however, this is often not the case. TCP has a complex set of rules for whether it can send data immediately upon the write, and if it can, how much. Understanding these rules and how they interact with our applications is essential if we are to write robust and efficient programs.

Similar considerations apply to reading data and to connection termination. We examine these operations and learn how to perform an orderly termination that ensures no data gets lost. We also examine the `connect` operation, how to time it out, and how to use it with UDP applications.

We study the use of the UNIX superserver, `inetd`, and see how its use can significantly reduce the effort needed to write a network-aware application. Similarly, we see how we can use `tcpmux` to relieve us of the necessity of worrying about assigning well-

known ports to our servers. We show how `tcpmux` works and build our own version to run on systems that don't already have the facility.

We look in depth at such little understood topics as the TIME-WAIT state, the Nagle algorithm, choosing buffer sizes, and the proper use of the `SO_REUSEADDR` socket option. We see how to make our applications event driven and how we can provide individual timers for each event. We examine some common mistakes that even experienced network programmers often make, and we study some techniques we can use to increase the performance of our applications.

Finally we take a look at networking and scripting languages. By using some Perl scripts, we demonstrate how it is possible to build useful network utilities and test drivers quickly and easily.

Chapter 4 addresses two areas. First we examine several tools that are essential for every network programmer. We start with the venerable `ping` utility and show how it can be used in some elementary troubleshooting. Next we examine network sniffers in general and `tcpdump` in particular. Throughout Chapter 4 we see examples of using `tcpdump` to diagnose application problems and puzzles. We study `traceroute` and use it to explore the shape of a tiny portion of the Internet.

The `ttcp` utility (co-authored by `ping` creator Mike Muuss) is a useful tool for studying network performance and the effect that certain TCP parameters have on that performance. We use it to demonstrate some diagnostic techniques. Another public domain tool, `lsof`, is invaluable for matching up network connections with the processes that have them open. Many times, `lsof` provides information that is not otherwise available without heroic effort.

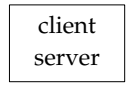
We take a long look at the `netstat` utility, and the many different types of information that it provides. We also examine system call trace facilities such as `ktrace` and `truss`.

We conclude our discussion of network diagnostic tools by building a utility to intercept and display Internet Control Messsage Protocol (ICMP) datagrams. This provides not only a useful addition to our toolbox, but also an example of the use of raw sockets.

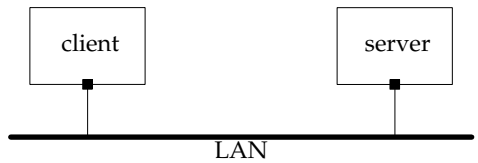
The second part of Chapter 4 discusses resources we can use to further our knowledge and understanding of TCP/IP and networking. We look at the remarkable books by Rich Stevens, sources of networking code that we can study and learn from, the request for comments (RFC) collection available from the Internet Engineering Task Force (IETF), and the Usenet news groups.

Client-Server Architecture

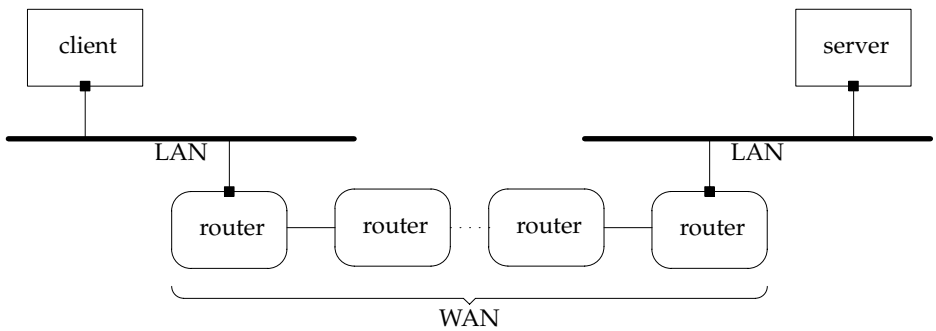
Although we speak of a *client* and a *server*, it is not always clear, in the general case, which role a particular program is playing. Often the programs are more like peers, exchanging information with neither clearly serving information to a client. With TCP/IP, though, the distinction is much clearer. The server listens for TCP connections or unsolicited UDP datagrams from a client or clients. Approaching this from the client's perspective, we can say that the client is the one who "speaks" first.



(A) Client and server on the same host



(B) Client and server on the same LAN



(C) Client and server on different LANs

Figure 1.1 Typical client-server situations

Throughout this text we consider three typical client-server situations, as shown in Figure 1.1. In the first, the client and server run on the same machine, as illustrated in Figure 1.1A. This is the simplest configuration because no physical network is involved. Output data is sent down the TCP/IP stack as usual, but instead of being placed on a network device output queue, it is looped back internally and travels back up the stack as input.

There are several advantages to this setup during development, even if the client and server eventually run on different machines. First, it is easier to judge the raw performance of the client and server applications because there is no network latency involved. Second, this method presents an idealized laboratory environment in which packets are not dropped, delayed, or delivered out of order.

> At least most of the time. As we shall see in Tip 7, it is possible to stress even this environment enough to cause the loss of UDP datagrams.

Finally, development is often easier and more convenient when we debug on the same machine.

It is also possible, of course, that the client and server will run on the same machine even in a production environment. See Tip 26 for an example of this.

Our second client-server setup, illustrated in Figure 1.1B, is for the client and server to run on different machines but on the same LAN. A real network is involved here, but this environment is still nearly ideal. Packets are rarely lost and virtually never arrive out of order. This is a very common production environment, and in many cases applications are not intended to run in any other.

A common example of this type of situation is a print server. A small LAN may have only one printer for several hosts. One of the hosts (or a TCP/IP stack built into the printer) acts as a server that takes print requests from clients on the other hosts and spools the data to the printer for printing.

The third type of client-server situation involves a client and server separated by a WAN (Figure 1.1C). The WAN could be the Internet or perhaps a corporate intranet, but the point is the two applications are not on the same LAN, and IP datagrams from one to the other must pass through one or more routers.

This environment can be significantly more hostile than the first two. As the amount of traffic on the WAN increases, the router queues used to store packets temporarily until they can be forwarded begin to fill up. When the routers run out of queue space they begin to drop packets. This leads to retransmissions, which in turn leads to duplicate and out-of-order delivery of packets. These problems are not theoretical and they're not rare, as we shall see in Tip 38.

We shall have more to say about the difference between the LAN and WAN environments in Tip 12, but for now we merely note that they can behave very differently.

Basic Sockets API Review

In this section we review the basic sockets API and use it to build rudimentary client and server applications. Although these applications are "bare-boned," they serve to illustrate the essential characteristics of a TCP client and server.

Let's start with the API calls we need for a simple client. Figure 1.2 shows the basic socket calls that are used by every client. As shown in Figure 1.2, the address of our peer is specified in a `sockaddr_in` structure that is passed to `connect`.

Generally, the first thing we must do is obtain a socket for the connection. We do this with the `socket` system call.

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

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

Returns: Socket descriptor on success, -1 (UNIX) or `INVALID_SOCKET` (Windows) on failure

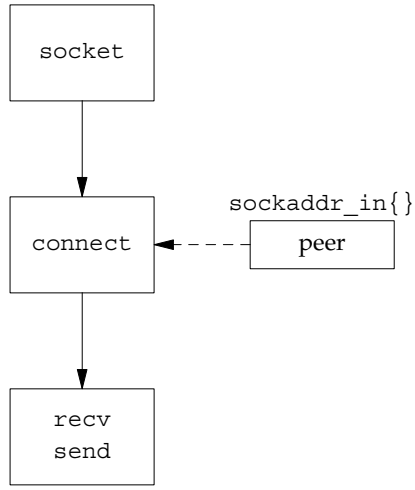


Figure 1.2 Basic socket calls for a client

The sockets API is protocol independent and can support several different *communications domains*. The *domain* parameter is a constant that represents the desired communications domain.

The two most common domains are the `AF_INET` (or Internet) domain and the `AF_LOCAL` (or `AF_UNIX`) domain. In this text we are concerned only with the `AF_INET` domain. The `AF_LOCAL` domain is used for *interprocess communication* (IPC) on the same machine.

> It's a matter of mild controversy as to whether the *domain* constants should be `AF_*` or `PF_*`. Proponents of PF_* point to their history in now-defunct versions of the `socket` call from 4.1c/2.8/2.9BSD and to the fact that PF stands for *protocol family*. Proponents for `AF_*` point out that the kernel socket code matches the *domain* parameter against the `AF_*` constants. Because the two sets of constants are defined the same—indeed, one is often defined in terms of the other—it makes no practical difference which we use.

The *type* parameter indicates the type of socket to be created. The most common values, and the ones we use in this text, are the following:

- `SOCK_STREAM`—These sockets provide a reliable, full duplex connection-oriented byte stream. In TCP/IP, this means *TCP*.
- `SOCK_DGRAM`—These sockets provide an unreliable, best-effort datagram service. In TCP/IP, this means *UDP*.
- `SOCK_RAW`—These sockets allow access to some datagrams at the IP layer. They are for special purposes such as listening for ICMP messages.

The *protocol* field indicates which protocol should be used with the socket. With TCP/IP, this is normally specified implicitly by the socket type, and the parameter is set to zero. In some cases, such as raw sockets, there are several possible protocols, and the one desired must be specified. We see an example of this in Tip 40.

For the simplest TCP client, the only other sockets API call we need to set up a conversation with our peer is `connect`, which is used to establish the connection:

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

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

Returns: 0 on success, -1 (UNIX) or nonzero (Windows) on failure

The *s* parameter is the socket descriptor returned by the `socket` call. The *peer* parameter points to an address structure that holds the address of the desired peer and some other information. For the `AF_INET` domain, this is a `sockaddr_in` structure. We look at a simple example in a moment. The *peer_len* parameter is the size of the structure pointed to by *peer*.

Once we've set up a connection, we are ready to transfer data. Under UNIX we can simply call `read` and `write` using the socket descriptor exactly as we would a file descriptor. Unfortunately, as we've already mentioned, Windows does not overload these system calls with socket semantics, so we have to use `recv` and `send` instead. These calls are just like `read` and `write` except that they have an additional parameter:

```
#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 );
```

Returns: number of bytes transferred on success, -1 on failure

The *s*, *buf*, and *len* parameters are the same as those for `read` and `write`. The values that the *flags* parameter can take are generally system dependent, but both UNIX and Windows support the following:

- `MSG_OOB`—When set, this flag causes urgent data to be sent or read.
- `MSG_PEEK`—This flag is used to peek at incoming data without removing it from the receive buffer. After the call, the data is still available for a subsequent read.
- `MSG_DONTROUTE`—This flag causes the kernel to bypass the normal routing function. It is generally used only by routing programs or for diagnostic purposes.

When dealing with TCP, these calls are generally all we need. For use with UDP, however, the `recvfrom` and `sendto` calls are useful. These calls are close cousins of `recv` and `send`, but they allow us to specify the destination address when sending a UDP datagram and to retrieve the source address when reading a UDP datagram:

```
#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 );
```

Returns: number of bytes transferred on success, -1 on failure

The first four parameters—*s*, *buf*, *len*, and *flags*—are the same as they were in the `recv` and `send` calls. The *from* parameter in the `recvfrom` call points to a socket address structure in which the kernel stores the source address of an incoming datagram. The length of this address is stored in the integer pointed to by *fromlen*. Notice that *fromlen* is a *pointer* to an integer.

Similarly, the *to* parameter in the `sendto` call points to a socket address structure that contains the address of the datagram's destination. The *tolen* parameter is the length of the address structure pointed to by *to*. Notice that *tolen* is a simple integer, not a pointer.

We're now in a position to look at a simple TCP client (Figure 1.3).

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( "socket call failed" );
19         exit( 1 );
20     }
```

```
21      rc = connect( s, ( struct sockaddr * )&peer, sizeof( peer ) );
22      if ( rc )
23      {
24          perror( "connect call failed" );
25          exit( 1 );
26      }
27      rc = send( s, "1", 1, 0 );
28      if ( rc <= 0 )
29      {
30          perror( "send call failed" );
31          exit( 1 );
32      }
33      rc = recv( s, buf, 1, 0 );
34      if ( rc <= 0 )
35          perror( "recv call failed" );
36      else
37          printf( "%c\n", buf[ 0 ] );
38      exit( 0 );
39  }
```

simplec.c

Figure 1.3 A simple TCP client

We have written Figure 1.3 as a UNIX program to delay having to deal with the complications of portable code and the Windows `WSAStartup` logic. As we'll see in Tip 4, we can hide most of this in a header file, but we need to set up some machinery first. In the meantime, we'll just use the slightly simpler UNIX model.

Set Up Our Peer's Address

12–14 We fill the `sockaddr_in` structure with the server's port number (7500) and address. The 127.0.0.1 is the loopback address. It specifies that the server is on the same host as the client.

Obtain a Socket and Connect to Our Peer

15–20 We obtain a `SOCK_STREAM` socket. As we mentioned earlier, TCP uses this type of socket because it is a stream protocol.

21–26 We establish the connection with our peer by calling `connect`. We use this call to specify our peer's address.

Send and Receive a Single Byte

27–38 We first send the single byte 1 to our peer, then immediately read a single byte from the socket. We write the byte to stdout and terminate.

Before we can exercise our client, we need a server. The socket calls for a server are a little different, as shown in Figure 1.4.

A server must listen for client connections on its well-known port. As we see in a moment, it does this with the `listen` call, but first it must bind the address of the interface and well-known port to its listening socket. This is done with the `bind` call:

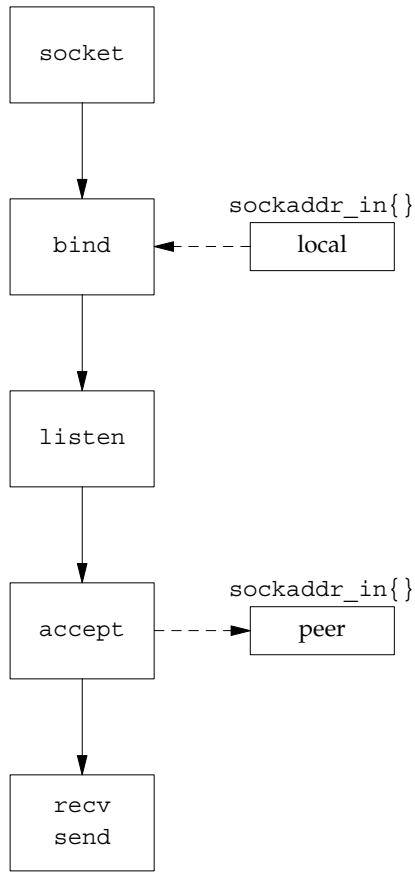


Figure 1.4 Basic socket calls in a server

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

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

Returns: 0 on success, -1 (UNIX) or SOCKET_ERROR (Windows) on error

The *s* parameter is the descriptor of the listening socket. The *name* and *namelen* parameters are used to supply the port and interface on which to listen. Usually the address is set to INADDR_ANY, indicating that a connection will be accepted on any interface. If a multihomed host wants to accept connections on only one interface, it can specify the IP address of that interface. As usual, *namelen* is the length of the sockaddr_in structure.

Once the local address is bound to the socket, we must start the socket listening for connections. This is done with the `listen` system call. This call is often misunderstood. Its only job is to mark the socket as listening. When a connection request arrives at the host, the kernel searches the list of listening sockets looking for one that matches the destination and port number in the request:

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

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

Returns: 0 on success, -1 (UNIX) or `SOCKET_ERROR` (Windows) on error

The *s* parameter is the socket descriptor of the socket we wish to mark as listening. The *backlog* parameter is the maximum number of pending connections that can be outstanding. This is *not* the maximum number of connections that can be established at the given port at one time. It is the maximum number of connections or partial connections that can be queued waiting for the application to accept them (see the `accept` call described later).

Traditionally the *backlog* parameter could not be set larger than five, but modern implementations with their need to support busy applications such as Web servers have made the maximum much larger. How large is system dependent, and we must check the system documentation to find the proper value for a given machine. If we specify a number larger than the maximum, the usual action is to reduce it silently to the maximum.

The final socket call is the `accept` system call. It is used to accept a connection from the queue of completed connections. Once accepted, the connection can be used for data transfer using, for example, the `recv` and `send` calls. On success, `accept` returns a descriptor for a new socket that can be used for data transfer. This socket has the same local port as the listening socket. The local address is the interface on which the connection came in. The foreign port and address are those of the client.

Notice that *both* sockets have the same local port. This is OK because a TCP connection is completely specified by the 4-tuple consisting of the local address, local port, foreign address, and foreign port. Because (at least) the foreign address and port of the two sockets differ, the kernel is able to tell them apart:

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

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

Returns: A connected socket if OK, -1 (UNIX) or `INVALID_SOCKET` (Windows) on failure

The *s* parameter is the socket descriptor of the *listening* socket. As shown in Figure 1.4, `accept` returns the address of the new connection's peer in the `sockaddr_in` structure pointed to by *addr*. The kernel places the length of this structure in the integer

pointed to by *addrlen*. We often don't care about our peer's address, and in this case we specify NULL for *addr* and *addrlen*.

We are now in a position to look at an elementary server (Figure 1.5). Again, this is a bare-boned program intended to show the server's structure and the basic socket calls that every server must make. As with the client code and Figure 1.2, we notice that our server follows the flow outlined in Figure 1.4 very closely.

Fill In Address and Get a Socket

12–20 We fill in the sockaddr_in structure, local, with our server's well-known address and port number. We use this for the bind call. As with the client, we obtain a SOCK_STREAM socket. This is our listening socket.

Bind Our Well-Known Port and Call `listen`

21–32 We bind the well-known port and address specified in local to our listening socket. We then call listen to mark the socket as a listening socket.

Accept a Connection

33–39 We call accept to accept new connections. The accept call blocks until a new connection is ready and then returns a new socket for that connection.

Transfer Data

39–49 We first read and print 1 byte from our client. Next, we send the single byte 2 back to our client and exit.

We can try our client and server by starting the server in one window and the client in another. Notice that it is important to start the server first or the client will fail with a "connection refused" error:

```
bsd: $ simplec
connect call failed: Connection refused
bsd: $
```

This error occurs because there was no server listening on port 7500 when the client tried to connect.

Now we start the server listening before we start the client:

```
bsd: $ simples          | bsd: $ simplec
1                       | 2
bsd: $                  | bsd: $
```

Summary

In this chapter we examined a road map for the rest of the text and briefly reviewed the basic sockets API. With this behind us, we can move forward, confident that we have the necessary background to undertake the study of the subject matter to come.

simples.c

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

 5 int main( void )
 6 {
 7     struct sockaddr_in local;
 8     int s;
 9     int s1;
10     int rc;
11     char buf[ 1 ];

12     local.sin_family = AF_INET;
13     local.sin_port = htons( 7500 );
14     local.sin_addr.s_addr = htonl( INADDR_ANY );
15     s = socket( AF_INET, SOCK_STREAM, 0 );
16     if ( s < 0 )
17     {
18         perror( "socket call failed" );
19         exit( 1 );
20     }
21     rc = bind( s, ( struct sockaddr * )&local, sizeof( local ) );
22     if ( rc < 0 )
23     {
24         perror( "bind call failure" );
25         exit( 1 );
26     }
27     rc = listen( s, 5 );
28     if ( rc )
29     {
30         perror( "listen call failed" );
31         exit( 1 );
32     }
33     s1 = accept( s, NULL, NULL );
34     if ( s1 < 0 )
35     {
36         perror( "accept call failed" );
37         exit( 1 );
38     }
39     rc = recv( s1, buf, 1, 0 );
40     if ( rc <= 0 )
41     {
42         perror( "recv call failed" );
43         exit( 1 );
44     }
45     printf( "%c\n", buf[ 0 ] );
46     rc = send( s1, "2", 1, 0 );
47     if ( rc <= 0 )
48         perror( "send call failed" );
49     exit( 0 );
50 }
```

simples.c

Figure 1.5 A simple TCP server

2

Basics

Tip 1: Understand the Difference between Connected and Connectionless Protocols

One of the most fundamental notions in network programming is that of *connection-oriented* versus *connectionless* protocols. Although there is nothing intrinsically difficult about the distinction, it is a frequent cause of confusion among those new to network programming. Part of the problem is a matter of context: Obviously two computers must be "connected" in some sense if they are to communicate, so what does "connectionless communication" mean?

The answer is that *connection-oriented* and *connectionless* refer to *protocols*. That is, the terms are applied to how we transfer data over a physical medium, not to the physical medium itself. Connection-oriented and connectionless protocols can, and routinely do, share the same physical medium simultaneously.

If the distinction has nothing to do with the physical medium carrying the data, what *does* it have to do with? The essential difference is that with connectionless protocols each packet is handled independently from any other, whereas with connection-oriented protocols, state information is maintained about successive packets by the protocol implementation.

With a connectionless protocol each packet, called a *datagram*, is addressed individually and sent by the application (but see Tip 30). From the point of view of the protocol, each datagram is an independent entity that has nothing to do with any other datagram that may have been sent between the same two peers.

> This is not to say that datagrams are independent from the point of view of the application. If the application implements anything more complicated than a simple request/reply protocol, in which a client sends a single request to a server and receives a single reply, then it will