UNIX Network Programming

The Sockets Networking API

Volume 1 • Third Edition

W. Richard Stevens
Bill Fenner
Andrew M. Rudoff

Addison-Wesley

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

GOOGLE EXHIBIT 1017

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and Addison-Wesley was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers discounts on this book when ordered in quantity for bulk purchases and special sales. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside of the U.S., please contact:

International Sales
(317) 581-3793
international@pearsontechgroup.com

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

Library of Congress Cataloging-in-Publication Data

A CIP catalog record for this book can be obtained from the Library of Congress.

Copyright © 2004 by Pearson Education, Inc.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher. Printed in the United States of America. Published simultaneously in Canada.

For information on obtaining permission for use of material from this work, please submit a written request to:

Pearson Education, Inc.
Rights and Contracts Department
75 Arlington Street, Suite 300
Boston, MA 02116
Fax: (617) 848-7047

ISBN: 0-13-141155-1

Text printed on recycled paper

First printing

Contents

Foreword xvii

Preface xix

Part 1. Introduction and TCP/IP 1

Chapter 1. Introduction 3

1.1 Introduction 3
1.2 A Simple Daytime Client 6
1.3 Protocol Independence 10
1.4 Error Handling: Wrapper Functions 11
1.5 A Simple Daytime Server 13
1.6 Roadmap to Client/Server Examples in the Text 16
1.7 OSI Model 18
1.8 BSD Networking History 20
1.9 Test Networks and Hosts 22
1.10 Unix Standards 25
1.11 64-Bit Architectures 28
1.12 Summary 29

Chapter 2. The Transport Layer: TCP, UDP, and SCTP 31

2.1 Introduction 31
2.2 The Big Picture 32
2.3 User Datagram Protocol (UDP) 34

2.4 Transmission Control Protocol (TCP) 35
2.5 Stream Control Transmission Protocol (SCTP) 36
2.6 TCP Connection Establishment and Termination 37
2.7 TIME_WAIT State 43
2.8 SCTP Association Establishment and Termination 44
2.9 Port Numbers 50
2.10 TCP Port Numbers and Concurrent Servers 52
2.11 Buffer Sizes and Limitations 55
2.12 Standard Internet Services 61
2.13 Protocol Usage by Common Internet Applications 62
2.14 Summary 63

Part 2. Elementary Sockets 65

Chapter 3. Sockets Introduction 67

3.1 Introduction 67
3.2 Socket Address Structures 67
3.3 Value-Result Arguments 74
3.4 Byte Ordering Functions 77
3.5 Byte Manipulation Functions 80
3.6 `inet_aton`, `inet_addr`, and `inet_ntoa` Functions 82
3.7 `inet_pton` and `inet_ntop` Functions 83
3.8 `sock_ntop` and Related Functions 86
3.9 `readn`, `writen`, and `readline` Functions 88
3.10 Summary 92

Chapter 4. Elementary TCP Sockets 95

4.1 Introduction 95
4.2 `socket` Function 95
4.3 `connect` Function 99
4.4 `bind` Function 101
4.5 `listen` Function 104
4.6 `accept` Function 109
4.7 `fork` and `exec` Functions 111
4.8 Concurrent Servers 114
4.9 `close` Function 117
4.10 `getsockname` and `getpeername` Functions 117
4.11 Summary 120

Chapter 5. TCP Client/Server Example 121

5.1 Introduction 121
5.2 TCP Echo Server: `main` Function 122
5.3 TCP Echo Server: `str_echo` Function 123
5.4 TCP Echo Client: `main` Function 124
5.5 TCP Echo Client: `str_cli` Function 125
5.6 Normal Startup 126
5.7 Normal Termination 128

5.8 POSIX Signal Handling 129
5.9 Handling SIGCHLD Signals 132
5.10 wait and waitpid Functions 135
5.11 Connection Abort before accept Returns 139
5.12 Termination of Server Process 141
5.13 SIGPIPE Signal 142
5.14 Crashing of Server Host 144
5.15 Crashing and Rebooting of Server Host 144
5.16 Shutdown of Server Host 145
5.17 Summary of TCP Example 146
5.18 Data Format 147
5.19 Summary 151

Chapter 6. I/O Multiplexing: The select and poll Functions 153

6.1 Introduction 153
6.2 I/O Models 154
6.3 select Function 160
6.4 str_cli Function (Revisited) 167
6.5 Batch Input and Buffering 169
6.6 shutdown Function 172
6.7 str_cli Function (Revisited Again) 173
6.8 TCP Echo Server (Revisited) 175
6.9 pselect Function 181
6.10 poll Function 182
6.11 TCP Echo Server (Revisited Again) 185
6.12 Summary 188

Chapter 7. Socket Options 191

7.1 Introduction 191
7.2 getsockopt and setsockopt Functions 192
7.3 Checking if an Option Is Supported and Obtaining the Default 194
7.4 Socket States 198
7.5 Generic Socket Options 198
7.6 IPv4 Socket Options 214
7.7 ICMPv6 Socket Option 216
7.8 IPv6 Socket Options 216
7.9 TCP Socket Options 219
7.10 SCTP Socket Options 222
7.11 fcntl Function 233
7.12 Summary 236

Chapter 8. Elementary UDP Sockets 239

8.1 Introduction 239
8.2 recvfrom and sendto Functions 240
8.3 UDP Echo Server: main Function 241
8.4 UDP Echo Server: dg_echo Function 242
8.5 UDP Echo Client: main Function 244
8.6 UDP Echo Client: dg_cli Function 245

IPv4 Socket Address Structure

An IPv4 socket address structure, commonly called an "Internet socket address structure," is named `sockaddr_in` and is defined by including the `<netinet/in.h>` header. Figure 3.1 shows the POSIX definition.

```
struct in_addr {
  in_addr_t   s_addr;          /* 32-bit IPv4 address */
                               /* network byte ordered */
};

struct sockaddr_in {
  uint8_t         sin_len;     /* length of structure (16) */
  sa_family_t     sin_family;  /* AF_INET */
  in_port_t       sin_port;    /* 16-bit TCP or UDP port number */
                               /* network byte ordered */
  struct in_addr  sin_addr;    /* 32-bit IPv4 address */
                               /* network byte ordered */
  char            sin_zero[8]; /* unused */
};
```

Figure 3.1 The Internet (IPv4) socket address structure: `sockaddr_in`.

There are several points we need to make about socket address structures in general using this example:

- The length member, `sin_len`, was added with 4.3BSD-Reno, when support for the OSI protocols was added (Figure 1.15). Before this release, the first member was `sin_family`, which was historically an `unsigned short`. Not all vendors support a length field for socket address structures and the POSIX specification does not require this member. The datatype that we show, `uint8_t`, is typical, and POSIX-compliant systems provide datatypes of this form (Figure 3.2).

  Having a length field simplifies the handling of variable-length socket address structures.

- Even if the length field is present, we need never set it and need never examine it, unless we are dealing with routing sockets (Chapter 18). It is used within the kernel by the routines that deal with socket address structures from various protocol families (e.g., the routing table code).

  The four socket functions that pass a socket address structure from the process to the kernel, `bind`, `connect`, `sendto`, and `sendmsg`, all go through the `sockargs` function in a Berkeley-derived implementation (p. 452 of TCPv2). This function copies the socket address structure from the process and explicitly sets its `sin_len` member to the size of the structure that was passed as an argument to these four functions. The five socket functions that pass a socket address structure from the kernel to the process, `accept`, `recvfrom`, `recvmsg`, `getpeername`, and `getsockname`, all set the `sin_len` member before returning to the process.

  Unfortunately, there is normally no simple compile-time test to determine whether an implementation defines a length field for its socket address structures. In our code, we test our own `HAVE_SOCKADDR_SA_LEN` constant (Figure D.2), but whether to define this constant or not

requires trying to compile a simple test program that uses this optional structure member and seeing if the compilation succeeds or not. We will see in Figure 3.4 that IPv6 implementations are required to define `SIN6_LEN` if socket address structures have a length field. Some IPv4 implementations provide the length field of the socket address structure to the application based on a compile-time option (e.g., `_SOCKADDR_LEN`). This feature provides compatibility for older programs.

- The POSIX specification requires only three members in the structure: `sin_family`, `sin_addr`, and `sin_port`. It is acceptable for a POSIX-compliant implementation to define additional structure members, and this is normal for an Internet socket address structure. Almost all implementations add the `sin_zero` member so that all socket address structures are at least 16 bytes in size.

- We show the POSIX datatypes for the `s_addr`, `sin_family`, and `sin_port` members. The `in_addr_t` datatype must be an unsigned integer type of at least 32 bits, `in_port_t` must be an unsigned integer type of at least 16 bits, and `sa_family_t` can be any unsigned integer type. The latter is normally an 8-bit unsigned integer if the implementation supports the length field, or an unsigned 16-bit integer if the length field is not supported. Figure 3.2 lists these three POSIX-defined datatypes, along with some other POSIX datatypes that we will encounter.

Datatype	Description	Header
`int8_t`	Signed 8-bit integer	`<sys/types.h>`
`uint8_t`	Unsigned 8-bit integer	`<sys/types.h>`
`int16_t`	Signed 16-bit integer	`<sys/types.h>`
`uint16_t`	Unsigned 16-bit integer	`<sys/types.h>`
`int32_t`	Signed 32-bit integer	`<sys/types.h>`
`uint32_t`	Unsigned 32-bit integer	`<sys/types.h>`
`sa_family_t`	Address family of socket address structure	`<sys/socket.h>`
`socklen_t`	Length of socket address structure, normally `uint32_t`	`<sys/socket.h>`
`in_addr_t`	IPv4 address, normally `uint32_t`	`<netinet/in.h>`
`in_port_t`	TCP or UDP port, normally `uint16_t`	`<netinet/in.h>`

Figure 3.2 Datatypes required by the POSIX specification.

- You will also encounter the datatypes `u_char`, `u_short`, `u_int`, and `u_long`, which are all unsigned. The POSIX specification defines these with a note that they are obsolete. They are provided for backward compatibility.

- Both the IPv4 address and the TCP or UDP port number are always stored in the structure in network byte order. We must be cognizant of this when using these members. We will say more about the difference between host byte order and network byte order in Section 3.4.
- The 32-bit IPv4 address can be accessed in two different ways. For example, if `serv` is defined as an Internet socket address structure, then `serv.sin_addr` references

the 32-bit IPv4 address as an `in_addr` structure, while `serv.sin_addr.s_addr` references the same 32-bit IPv4 address as an `in_addr_t` (typically an unsigned 32-bit integer). We must be certain that we are referencing the IPv4 address correctly, especially when it is used as an argument to a function, because compilers often pass structures differently from integers.

> The reason the `sin_addr` member is a structure, and not just an `in_addr_t`, is historical. Earlier releases (4.2BSD) defined the `in_addr` structure as a `union` of various structures, to allow access to each of the 4 bytes and to both of the 16-bit values contained within the 32-bit IPv4 address. This was used with class A, B, and C addresses to fetch the appropriate bytes of the address. But with the advent of subnetting and then the disappearance of the various address classes with classless addressing (Section A.4), the need for the `union` disappeared. Most systems today have done away with the `union` and just define `in_addr` as a structure with a single `in_addr_t` member.

- The `sin_zero` member is unused, but we *always* set it to 0 when filling in one of these structures. By convention, we always set the entire structure to 0 before filling it in, not just the `sin_zero` member.

  > Although most uses of the structure do not require that this member be 0, when binding a nonwildcard IPv4 address, this member must be 0 (pp. 731–732 of TCPv2).

- Socket address structures are used only on a given host: The structure itself is not communicated between different hosts, although certain fields (e.g., the IP address and port) are used for communication.

Generic Socket Address Structure

A socket address structures is *always* passed by reference when passed as an argument to any socket functions. But any socket function that takes one of these pointers as an argument must deal with socket address structures from *any* of the supported protocol families.

A problem arises in how to declare the type of pointer that is passed. With ANSI C, the solution is simple: `void *` is the generic pointer type. But, the socket functions predate ANSI C and the solution chosen in 1982 was to define a *generic* socket address structure in the `<sys/socket.h>` header, which we show in Figure 3.3.

```
struct sockaddr {
  uint8_t       sa_len;
  sa_family_t   sa_family;    /* address family: AF_xxx value */
  char          sa_data[14];  /* protocol-specific address */
};
```

Figure 3.3 The generic socket address structure: `sockaddr`.

The socket functions are then defined as taking a pointer to the generic socket address structure, as shown here in the ANSI C function prototype for the `bind` function:

4.5 `listen` Function

The `listen` function is called only by a TCP server and it performs two actions:

1. When a socket is created by the `socket` function, it is assumed to be an active socket, that is, a client socket that will issue a `connect`. The `listen` function converts an unconnected socket into a passive socket, indicating that the kernel should accept incoming connection requests directed to this socket. In terms of the TCP state transition diagram (Figure 2.4), the call to `listen` moves the socket from the CLOSED state to the LISTEN state.
2. The second argument to this function specifies the maximum number of connections the kernel should queue for this socket.

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

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

Returns: 0 if OK, −1 on error

This function is normally called after both the `socket` and `bind` functions and must be called before calling the `accept` function.

To understand the *backlog* argument, we must realize that for a given listening socket, the kernel maintains two queues:

1. An *incomplete connection queue*, which contains an entry for each SYN that has arrived from a client for which the server is awaiting completion of the TCP three-way handshake. These sockets are in the SYN_RCVD state (Figure 2.4).
2. A *completed connection queue*, which contains an entry for each client with whom the TCP three-way handshake has completed. These sockets are in the ESTABLISHED state (Figure 2.4).

Figure 4.7 depicts these two queues for a given listening socket.

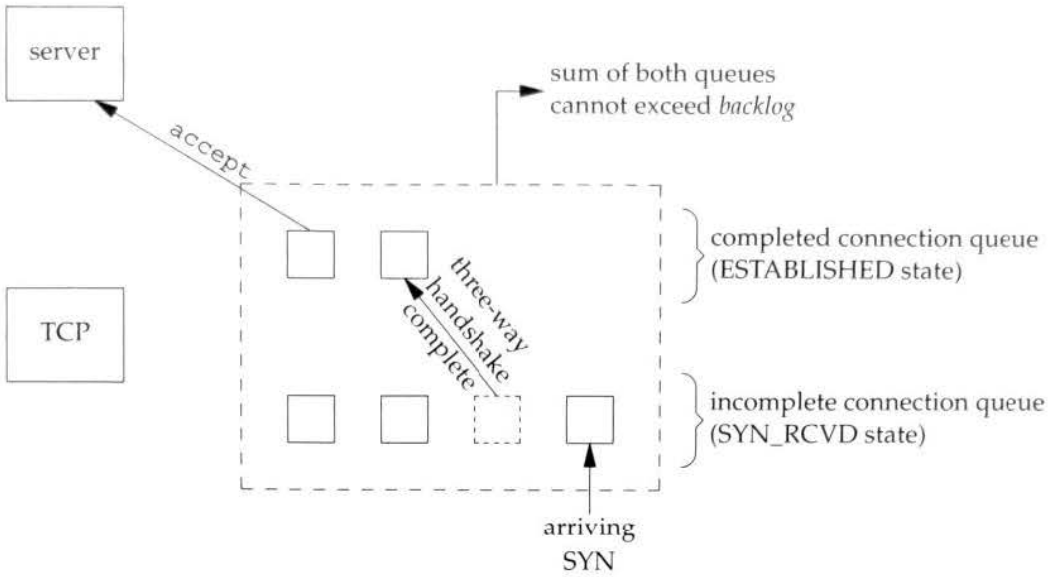


Figure 4.7 The two queues maintained by TCP for a listening socket.

When an entry is created on the incomplete queue, the parameters from the listen socket are copied over to the newly created connection. The connection creation mechanism is completely automatic; the server process is not involved. Figure 4.8 depicts the packets exchanged during the connection establishment with these two queues.

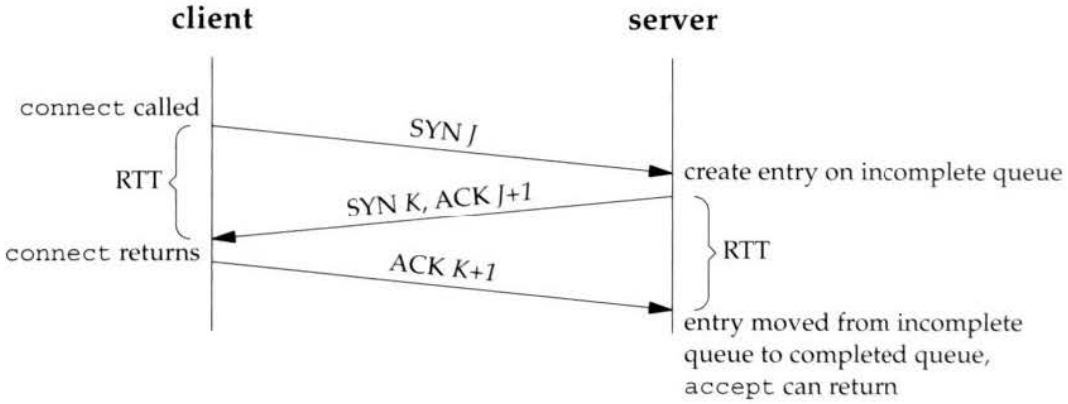


Figure 4.8 TCP three-way handshake and the two queues for a listening socket.

When a SYN arrives from a client, TCP creates a new entry on the incomplete queue and then responds with the second segment of the three-way handshake: the server's SYN with an ACK of the client's SYN (Section 2.6). This entry will remain on the incomplete queue until the third segment of the three-way handshake arrives (the client's ACK of the server's SYN), or until the entry times out. (Berkeley-derived implementations have a timeout of 75 seconds for these incomplete entries.) If the three-way handshake completes normally, the entry moves from the incomplete queue to the end of the completed queue. When the process calls `accept`, which we will describe in the next section, the first entry on the completed queue is returned to the process, or if the

queue is empty, the process is put to sleep until an entry is placed onto the completed queue.

There are several points to consider regarding the handling of these two queues.

- The *backlog* argument to the `listen` function has historically specified the maximum value for the sum of both queues.

  There has never been a formal definition of what the *backlog* means. The 4.2BSD man page says that it "defines the maximum length the queue of pending connections may grow to." Many man pages and even the POSIX specification copy this definition verbatim, but this definition does not say whether a pending connection is one in the SYN_RCVD state, one in the ESTABLISHED state that has not yet been accepted, or either. The historical definition in this bullet is the Berkeley implementation, dating back to 4.2BSD, and copied by many others.

- Berkeley-derived implementations add a fudge factor to the *backlog*: It is multiplied by 1.5 (p. 257 of TCPv1 and p. 462 of TCPv2). For example, the commonly specified *backlog* of 5 really allows up to 8 queued entries on these systems, as we show in Figure 4.10.

  The reason for adding this fudge factor appears lost to history [Joy 1994]. But if we consider the *backlog* as specifying the maximum number of completed connections that the kernel will queue for a socket ([Borman 1997b], as discussed shortly), then the reason for the fudge factor is to take into account incomplete connections on the queue.

- Do not specify a *backlog* of 0, as different implementations interpret this differently (Figure 4.10). If you do not want any clients connecting to your listening socket, close the listening socket.

- Assuming the three-way handshake completes normally (i.e., no lost segments and no retransmissions), an entry remains on the incomplete connection queue for one RTT, whatever that value happens to be between a particular client and server. Section 14.4 of TCPv3 shows that for one Web server, the median RTT between many clients and the server was 187 ms. (The median is often used for this statistic, since a few large values can noticeably skew the mean.)

- Historically, sample code always shows a *backlog* of 5, as that was the maximum value supported by 4.2BSD. This was adequate in the 1980s when busy servers would handle only a few hundred connections per day. But with the growth of the World Wide Web (WWW), where busy servers handle millions of connections per day, this small number is completely inadequate (pp. 187–192 of TCPv3). Busy HTTP servers must specify a much larger *backlog*, and newer kernels must support larger values.

  Many current systems allow the administrator to modify the maximum value for the *backlog*.

- A problem is: What value should the application specify for the *backlog*, since 5 is often inadequate? There is no easy answer to this. HTTP servers now specify a larger value, but if the value specified is a constant in the source code, to increase the constant requires recompiling the server. Another method is to assume some default but allow a command-line option or an environment variable to override

the default. It is always acceptable to specify a value that is larger than supported by the kernel, as the kernel should silently truncate the value to the maximum value that it supports, without returning an error (p. 456 of TCPv2).

We can provide a simple solution to this problem by modifying our wrapper function for the `listen` function. Figure 4.9 shows the actual code. We allow the environment variable `LISTENQ` to override the value specified by the caller.

lib/wrapsock.c

```
137 void
138 Listen(int fd, int backlog)
139 {
140     char    *ptr;

141         /* can override 2nd argument with environment variable */
142     if ( (ptr = getenv("LISTENQ")) != NULL)
143         backlog = atoi(ptr);

144     if (listen(fd, backlog) < 0)
145         err_sys("listen error");
146 }
```

lib/wrapsock.c

Figure 4.9 Wrapper function for `listen` that allows an environment variable to specify *backlog*.

- Manuals and books have historically said that the reason for queuing a fixed number of connections is to handle the case of the server process being busy between successive calls to `accept`. This implies that of the two queues, the completed queue should normally have more entries than the incomplete queue. Again, busy Web servers have shown that this is false. The reason for specifying a large *backlog* is because the incomplete connection queue can grow as client SYNs arrive, waiting for completion of the three-way handshake.

- If the queues are full when a client SYN arrives, TCP ignores the arriving SYN (pp. 930–931 of TCPv2); it does not send an RST. This is because the condition is considered temporary, and the client TCP will retransmit its SYN, hopefully finding room on the queue in the near future. If the server TCP immediately responded with an RST, the client's `connect` would return an error, forcing the application to handle this condition instead of letting TCP's normal retransmission take over. Also, the client could not differentiate between an RST in response to a SYN meaning "there is no server at this port" versus "there is a server at this port but its queues are full."

  Some implementations do send an RST when the queue is full. This behavior is incorrect for the reasons stated above, and unless your client specifically needs to interact with such a server, it's best to ignore this possibility. Coding to handle this case reduces the robustness of the client and puts more load on the network in the normal RST case, where the port really has no server listening on it.

- Data that arrives after the three-way handshake completes, but before the server calls `accept`, should be queued by the server TCP, up to the size of the connected socket's receive buffer.

Figure 4.10 shows the actual number of queued connections provided for different values of the *backlog* argument for the various operating systems in Figure 1.16. For seven different operating systems there are five distinct columns, showing the variety of interpretations about what *backlog* means!

	Maximum actual number of queued connections				
backlog	MacOS 10.2.6 AIX 5.1	Linux 2.4.7	HP-UX 11.11	FreeBSD 4.8 FreeBSD 5.1	Solaris 2.9
0	1	3	1	1	1
1	2	4	1	2	2
2	4	5	3	3	4
3	5	6	4	4	5
4	7	7	6	5	6
5	8	8	7	6	8
6	10	9	9	7	10
7	11	10	10	8	11
8	13	11	12	9	13
9	14	12	13	10	14
10	16	13	15	11	16
11	17	14	16	12	17
12	19	15	18	13	19
13	20	16	19	14	20
14	22	17	21	15	22

Figure 4.10 Actual number of queued connections for values of *backlog*.

AIX and MacOS have the traditional Berkeley algorithm, and Solaris seems very close to that algorithm as well. FreeBSD just adds one to *backlog*.

The program to measure these values is shown in the solution for Exercise 15.4.

As we said, historically the *backlog* has specified the maximum value for the sum of both queues. During 1996, a new type of attack was launched on the Internet called *SYN flooding* [CERT 1996b]. The hacker writes a program to send SYNs at a high rate to the victim, filling the incomplete connection queue for one or more TCP ports. (We use the term *hacker* to mean the attacker, as described in [Cheswick, Bellovin, and Rubin 2003].) Additionally, the source IP address of each SYN is set to a random number (this is called *IP spoofing*) so that the server's SYN/ACK goes nowhere. This also prevents the server from knowing the real IP address of the hacker. By filling the incomplete queue with bogus SYNs, legitimate SYNs are not queued, providing a *denial of service* to legitimate clients. There are two commonly used methods of handling these attacks, summarized in [Borman 1997b]. But what is most interesting in this note is revisiting what the `listen` *backlog* really means. It should specify the maximum number of *completed* connections for a given socket that the kernel will queue. The purpose of having a limit on these completed connections is to stop the kernel from accepting new connection requests for a given socket when the application is not accepting them (for whatever reason). If a system implements this interpretation, as does BSD/OS 3.0, then the application need not specify huge *backlog* values just because the server handles lots of client requests (e.g., a busy

Web server) or to provide protection against SYN flooding. The kernel handles lots of incomplete connections, regardless of whether they are legitimate or from a hacker. But even with this interpretation, scenarios do occur where the traditional value of 5 is inadequate.

4.6 accept Function

accept is called by a TCP server to return the next completed connection from the front of the completed connection queue (Figure 4.7). If the completed connection queue is empty, the process is put to sleep (assuming the default of a blocking socket).

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

int accept(int sockfd, struct sockaddr *cliaddr, socklen_t *addrlen);
```

Returns: non-negative descriptor if OK, −1 on error

The *cliaddr* and *addrlen* arguments are used to return the protocol address of the connected peer process (the client). *addrlen* is a value-result argument (Section 3.3): Before the call, we set the integer value referenced by **addrlen* to the size of the socket address structure pointed to by *cliaddr*; on return, this integer value contains the actual number of bytes stored by the kernel in the socket address structure.

If accept is successful, its return value is a brand-new descriptor automatically created by the kernel. This new descriptor refers to the TCP connection with the client. When discussing accept, we call the first argument to accept the *listening socket* (the descriptor created by socket and then used as the first argument to both bind and listen), and we call the return value from accept the *connected socket*. It is important to differentiate between these two sockets. A given server normally creates only one listening socket, which then exists for the lifetime of the server. The kernel creates one connected socket for each client connection that is accepted (i.e., for which the TCP three-way handshake completes). When the server is finished serving a given client, the connected socket is closed.

This function returns up to three values: an integer return code that is either a new socket descriptor or an error indication, the protocol address of the client process (through the *cliaddr* pointer), and the size of this address (through the *addrlen* pointer). If we are not interested in having the protocol address of the client returned, we set both *cliaddr* and *addrlen* to null pointers.

Figure 1.9 shows these points. The connected socket is closed each time through the loop, but the listening socket remains open for the life of the server. We also see that the second and third arguments to accept are null pointers, since we were not interested in the identity of the client.

Example: Value-Result Arguments

We will now show how to handle the value-result argument to accept by modifying the code from Figure 1.9 to print the IP address and port of the client. We show this in Figure 4.11.