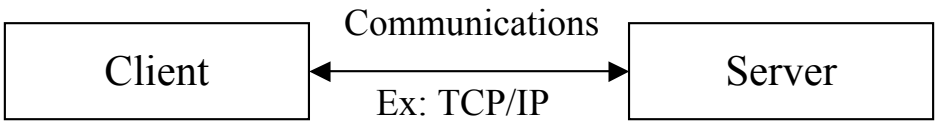
INTRODUCTION"UNIX NETWORK PROGRAMMING" Vol 1, Third Edition by Richard Stevens

Read: Chapters 1,2, 3, 4



Example: Telnet client on local machine to Telnet server on a remote machine

Client and server are "user" processes
TCP and IP are normally part on the "kernel" protocol stack

Aside on compiling programs in linux:

To compile a program in Linux (not using make) for example the program intro/byteorder.c on p. 78

gcc byteorder.c

This places the executable in a default file a.out

gcc -o byteorder byteorder.c

Places executable in byteorder ; can run by **.** / byteorder

Linux Configuration Tip:

Instead of having to type **.** / program every time you run a “program”
what you can do is in **.**bash_profile in your /root directory add to the path line.

Path = . :
Add this

Now you can just type program instead of ./program since **.** is now included in your path
(From a security standpoint this is a bad thing to do, thus it is not already there for you.)

On another Linux subject:
To switch from one default text login screen to another (not in the graphics GUI x windows)

Alt	F1	Default screen
Alt	F2	A second screen
Alt	F3	A third screen
etc		

Name	TCP port	UDP port	RFC	Description
echo	7	7	862	Server returns whatever the client sends.
discard	9	9	863	Server discards whatever the client sends.
daytime	13	13	867	Server returns the time and date in a human-readable format.
chargen	19	19	864	TCP server sends a continual stream of characters, until the connection is terminated by the client. UDP server sends a datagram containing a random number of characters each time the client sends a datagram.
time	37	37	868	Server returns the time as a 32-bit binary number. This number represents the number of seconds since midnight January 1, 1900, UTC.

Figure 2.13 Standard TCP/IP services provided by most implementations.

Now Figure 2.18 page 61

Example: a simple daytime client:

What this example client program does is connect to a server and gets the date and time if we run:

. / daytimetcpcli reachable_IP_address

for example: 127.0.0.1

Gets date & time back printed out on screen
(However this will not run unless we have a daytime server running on port 13. We can do this by using server code later in this lecture or by in a config file turning on the daytime daemon which is off by default in our linux install.)

Details:

Line 1 : Includes a bunch on code (unp.h) in appendix D.1 (pages 899-912) with commonly used stuff like MAXLINE (on line 6 of program).

Line 2-3: Allows values to be passed to our program for example an IP address is considered to be the second value.

intro/daytimetcpcli.c

```
 1 #include    "unp.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     sockfd, n;
 6     char    recvline[MAXLINE + 1];
 7     struct sockaddr_in servaddr;

 8     if (argc != 2)
 9         err_quit("usage: a.out <IPaddress>");

10     if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
11         err_sys("socket error");

12     bzero(&servaddr, sizeof(servaddr));
13     servaddr.sin_family = AF_INET;
14     servaddr.sin_port = htons(13);  /* daytime server */
15     if (inet_pton(AF_INET, argv[1], &servaddr.sin_addr) <= 0)
16         err_quit("inet_pton error for %s", argv[1]);
```

```
17      if (connect(sockfd, (SA *) &servaddr, sizeof(servaddr)) < 0)
18          err_sys("connect error");

19      while ( (n = read(sockfd, recvline, MAXLINE)) > 0) {
20          recvline[n] = 0;        /* null terminate */
21          if (fputs(recvline, stdout) == EOF)
22              err_sys("fputs error");
23      }
24      if (n < 0)
25          err_sys("read error");

26      exit(0);
27  }
```

intro/daytimetcpcli.c

Figure 1.5 TCP daytime client.

Continued:

Line 8 : If you do not have an IP value stop.

Line 10-11: Your first use of the sockets application programming interface calling the function "socket."

- This creates an Internet (AF_ INET) stream (SOCK_STREAM) socket => means TCP socket.
- Function returns an integer descriptor used to identify this particular socket in later code.
- Failure calls a function err_sys in our unp.h code

Line 12: Function bzero clears out by filling with zeros the structure servaddr

Line 13: We fill in the structure member servaddr.sin_family with the type AF_INET meaning internet type.

Line 14: We set servaddr.sin_port to the port number 13. This is "well-known" daytime port on TCP/IP hosts.

[Fig 2.13]

Call function htons

" Host To Network Short" to convert the decimal value into format we need.

Line 15-16 : Call function inet_pton
"Presentation to numeric" to convert ASCII value input in command line to format needed. Put result in servaddr.sin_addr If return error value of a negative number, call err_sys function

Line 17-18 : Connect function establishes TCP connection with server specified by the socket address structure pointed to by &servaddr which we set in line 15

Lines 19-25 : Read server's reply to an enquiry on port 13, display result using function fputs
TCP is a BYTE-Stream Protocol with no "record" boundaries.
Therefore for TCP socket always need to code read in a loop which terminates on 0 or end.

Line 26: Exit terminates program. TCP socket is closed automatically

Error Handling:

Stevens defines wrapper functions to perform function call, test of return value and error handling.

Stevens indicates one of his wrapper functions with an upper case letter.

Our wrapper function is shown in Figure 1.7.

lib/wrapsock.c

```
172 int
173 Socket(int family, int type, int protocol)
174 {
175     int     n;

176     if ( (n = socket(family, type, protocol)) < 0)
177         err_sys("socket error");
178     return (n);
179 }
```

lib/wrapsock.c

Figure 1.7 Our wrapper function for the `socket` function.

intro/daytimetcpsrv.c

```
 1 #include    "unp.h"
 2 #include    <time.h>

 3 int
 4 main(int argc, char **argv)
 5 {
 6     int     listenfd, connfd;
 7     struct sockaddr_in servaddr;
 8     char    buff[MAXLINE];
 9     time_t  ticks;

10     listenfd = Socket(AF_INET, SOCK_STREAM, 0);

11     bzero(&servaddr, sizeof(servaddr));
12     servaddr.sin_family = AF_INET;
13     servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
14     servaddr.sin_port = htons(13);  /* daytime server */
```

```
15      Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));

16      Listen(listenfd, LISTENQ);

17      for ( ; ; ) {
18          connfd = Accept(listenfd, (SA *) NULL, NULL);

19          ticks = time(NULL);
20          snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
21          Write(connfd, buff, strlen(buff));

22          Close(connfd);
23      }
24  }
```

intro/daytimetcpsrv.c

Figure 1.9 TCP daytime server.

Line 10: Creates the TCP socket. Now we are using the wrapper function with a capital S. Instead of sockfd we put returned integer descriptor in listenfd. Otherwise identical to Fig 1.5 client code line 10.

Line 11-15: Bind the servers port 13 to the socket by filling in internet socket address structure.

Line 11: Function bzero clears out by filling with zeros the structure servddr.

Line 12: Fill in the structure member servaddr.sin_family with type AF_INET meaning internet type.

Line 13: Fill in the structure member servaddr.sin_addr.s_addr with the IP address INADDR _ ANY This allows server to accept a client connection on any interface. Use function htonl which is host to network long to convert declimal value into format needed.

Line 14: Set structure member servaddr.sin_ port to port number 13

Line 15: Use function bind on socket pointed to by listenfd to assign a local protocol address to the socket.

Line 16: Using the function listen the socket is converted into one that accepts incoming connections from clients.

The Three Steps:

- Socket
- Bind
- Listen

•Are used to prepare a TCP server "Listening Descriptor" which is listenfd in our example.

•The constant LISTENQ is from header "unp.h" and specifies max number of client connections.

Line 17-21: For (; ;) { } This loops forever
We wait at line 18 until a client has connected to us and the TCP 3 way handshake has occurred.
=> Function accept returns description connfd which is the "connected descriptor."

Line 19: Function time returns time.

Line 20: Function ctime converts to human readable form.
snprintf adds a return and line feed.

Line 21: Function write sends result back to client.

Line 22: Server closes connection with client by calling close.

Simple Daytime Server

To run this server and then client in directory with the source code.

. /daytimetcpsrv &

↑— puts in background mode

. /daytimetcpcli 127.0.0.1

ps shows . / daytimetcpsrv is running

kill <number>

ps shows no longer running

Internet Requests For Comments (RFC)

RFC documents are definitions of the protocols and policies of the internet. RFCs contain all the nuts and bolts information.

Examples:

• RFC 793	Transmission Control Protocol	1981
• RFC 768	User Datagram Protocol	1980
• RFC 791	Internet Protocol	1981

You can get your own copies and do keyword searches at:

http://www.cis.ohio-state.edu/hypertext/information/rfc.html

Socket Pair

The socket pair for a TCP connection is the 4- tuple that defines the two end points

Local	IP	Address
Local	TCP	Port
Foreign	IP	Address
Foreign	TCP	Port

A socket pair uniquely identities every TCP connection on an internet. We extend this same concept to UDP.

Chapter 4 Elementary TCP Sockets

Examine a TCP client and server:

- First server starts, later the client that connects to server is started.
- In this example the client sends a request to the server.
- Server then processes the request and sends a response back to the client.
- Continues until client closes its end of the connection by sending end of the connection by sending end of file notification to server.
- Server then closes its end of the connection.

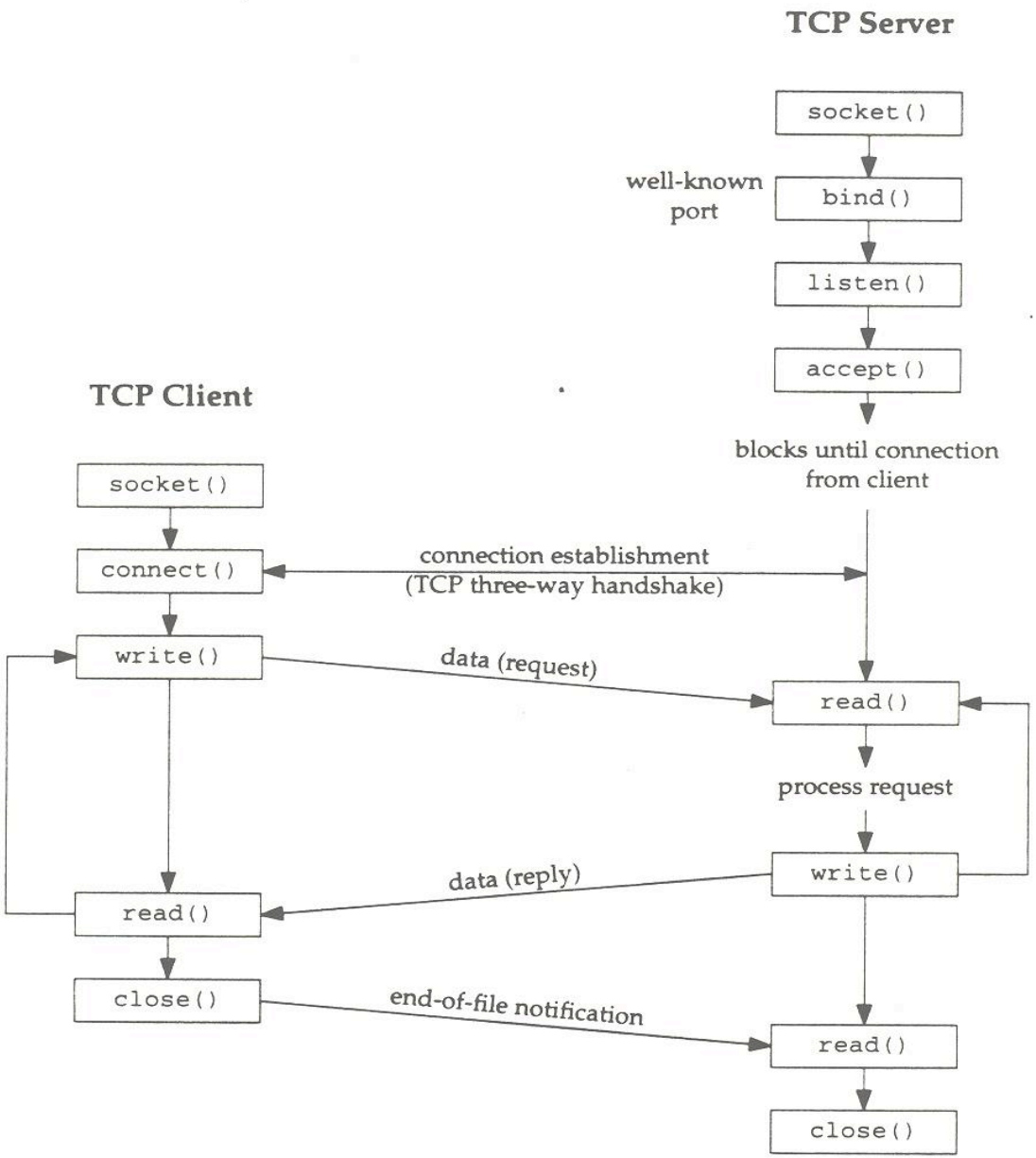


Figure 4.1 Socket functions for elementary TCP client–server.

Socket Function

Specifies type of communication protocol desired (ie. TCP using IPV4)

- Family specifies protocol family. AF_INET is IPV4
- Socket type is for example SOCK_STREAM for TCP, SOCK_DGRAM for UDP.
- Protocol argument is usually 0. (Always 0 for us)
- On success socket function returns a non-negative integer value. This returned value is called a socket descriptor or sockfd.

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

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

Returns: nonnegative descriptor if OK, −1 on error

family	Description
AF_INET	IPv4 protocols
AF_INET6	IPv6 protocols
AF_LOCAL	Unix domain protocols (Chapter 14)
AF_ROUTE	Routing sockets (Chapter 17)
AF_KEY	Key socket

Figure 4.2 Protocol *family* constants for `socket` function.

type	Description
SOCK_STREAM	stream socket
SOCK_DGRAM	datagram socket
SOCK_RAW	raw socket

Figure 4.3 *type* of socket for `socket` function.

	AF_INET	AF_INET6	AF_LOCAL	AF_ROUTE	AF_KEY
SOCK_STREAM	TCP	TCP	Yes		
SOCK_DGRAM	UDP	UDP	Yes		
SOCK_RAW	IPv4	IPv6		Yes	Yes

Figure 4.4 Combinations of *family* and *type* for the `socket` function.

Connect Function

Used by TCP client to establish a connection with a TCP server. Causes a Client to send SYN

[Fig. connect]

- sockfd is the socket descriptor returned by socket function.
- 2nd & 3rd arguments are a pointer to a socket address structure and its size.
- The socket address structure must contain IP address and port number of server.

[See fig. 1.5 again for how we set this in our example]

- Connect function initiates TCP's 3-way handshake
- Connect moves from CLOSED state to SYN_SENT state and then with success on to ESTABLISHED

[Fig. 2.2 & 2.4]

4.3 connect Function

The connect function is used by a TCP client to establish a connection with a TCP server.

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

int connect(int sockfd, const struct sockaddr *servaddr, socklen_t addrlen);
```

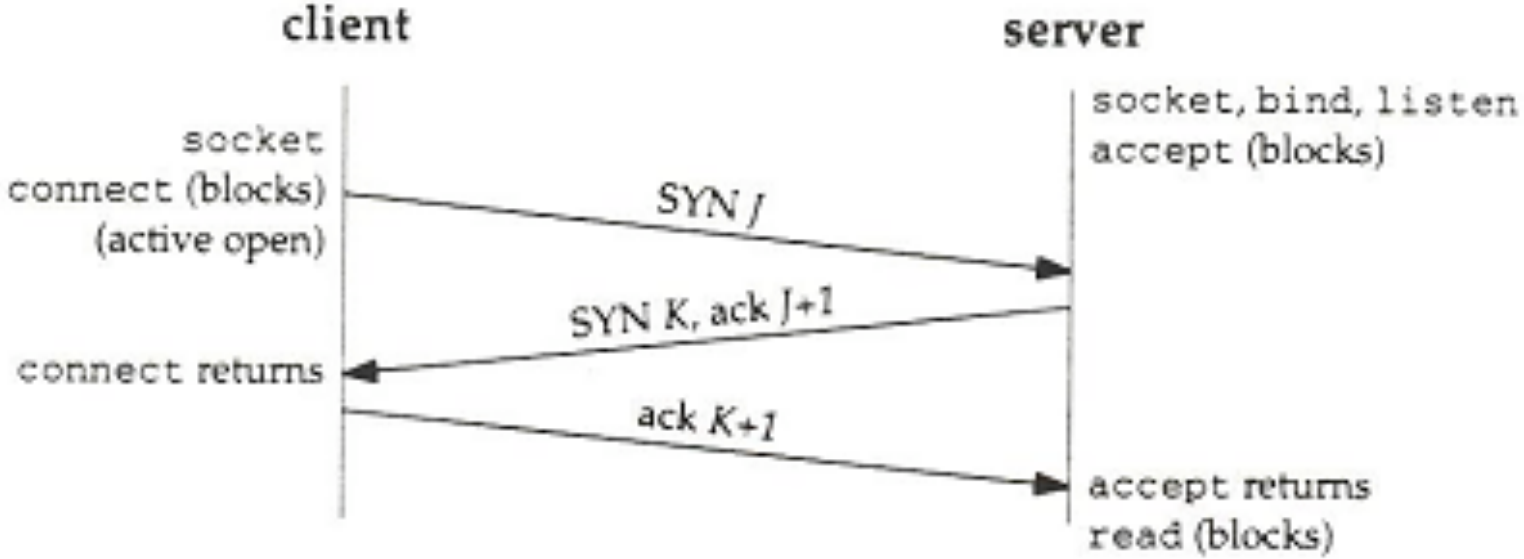


Figure 2.2 TCP three-way handshake.

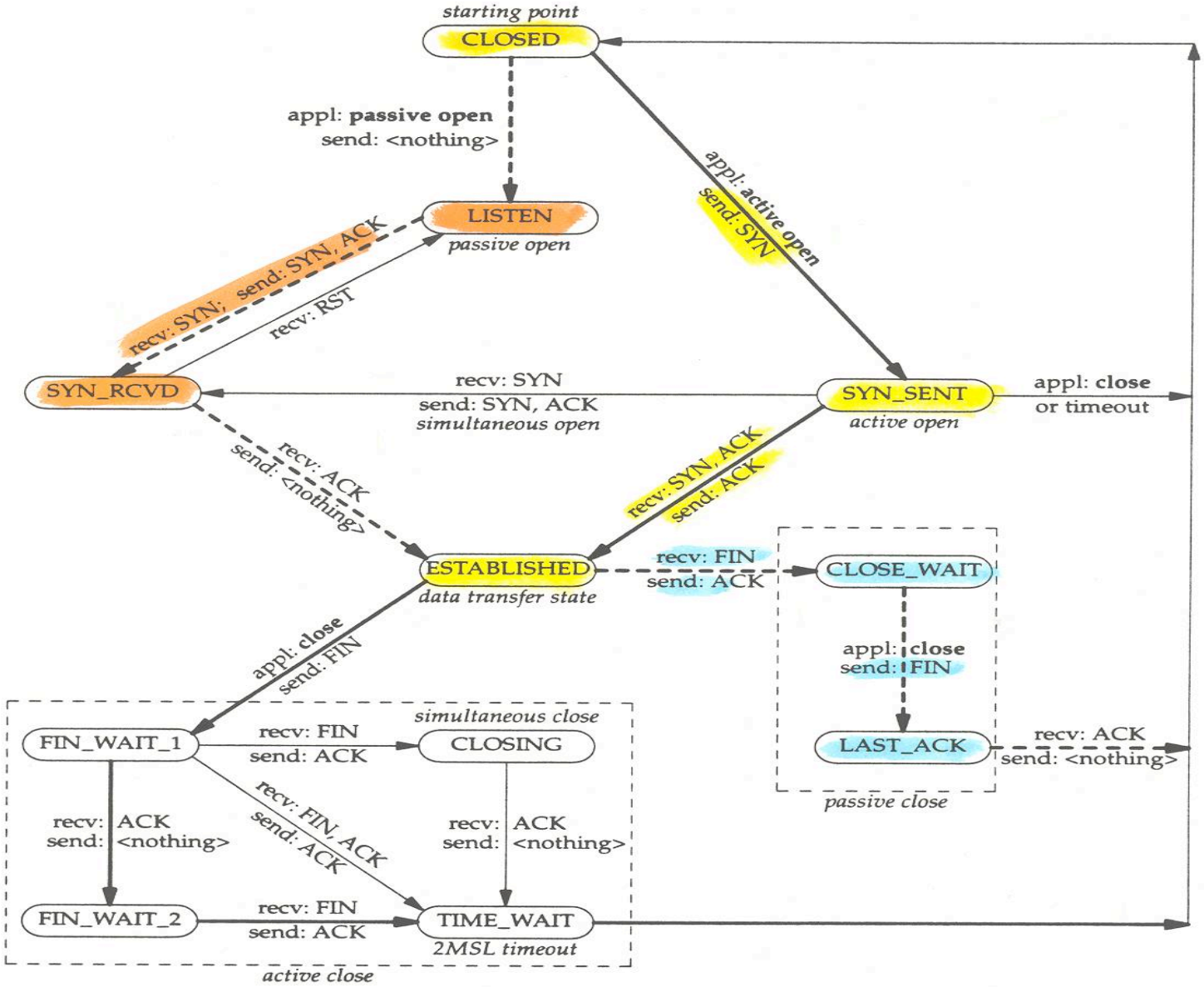


Figure 2.4 TCP state transition diagram.

Generic versus our specific use of Sockets API

Applications using sockets must specify IP addresses and port numbers to the kernel

The **generic** (not just for IP) data type for doing this is the sockaddr structure:

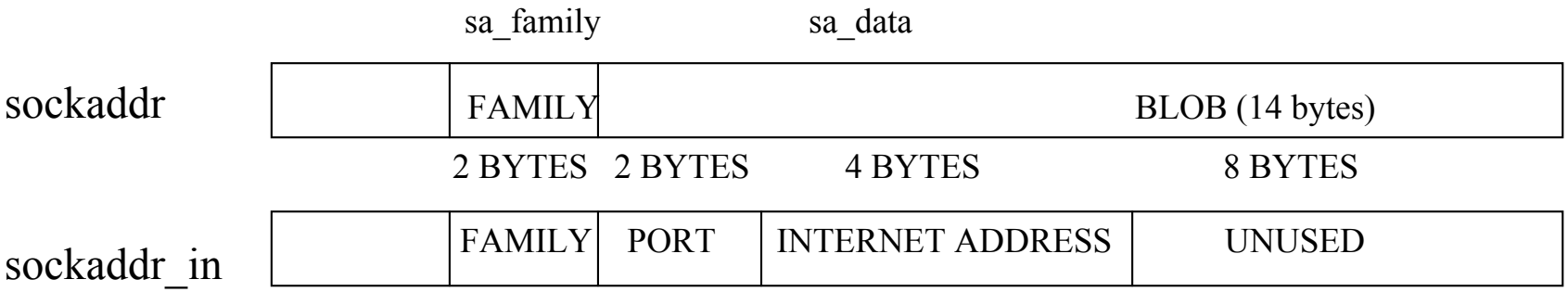
```
struct sockaddr {
uint8_t         sa_len
sa_family_t     sa_family       /*address family: AF_XXX value*/
char            sa_data[14]     /*protocol specific address*/
```

The **generic** sockets application programming interface (API) functions like bind require this structure because this is how they were first defined:

```
int         bind(int, sockaddr  *, socklen_t);
```

The **specific** form of the sockaddr structure that is used for the TCP/IP socket addresses is the sockaddr_in structure.

Comparing the generic structure with the one we need for TCP/IP:



This requires that any calls to these functions must cast the pointer to the protocol specific socket address structure (our TCP/IP stuff) to be a pointer to a generic socket address structure (the original generic stuff).

For example:

struct sockaddr_in serv
bind (sockfd, (struct sockaddr *) &serv, sizeof(serv));

Since serv is type sockaddr_in and we must use type sockaddr

Stevens defines in unp.h:
SA is equal to the string “struct sockaddr” just to pretty up the book!
(See page 9)

So when Stevens does the following he is casting the pointer to the Protocol specific structure we want to use into the generic structure we must use if we want to use the old already defined functions.

bind (sockfd, (SA *) &serv, sizeof(serv));

Bind Function

Assigns a **local protocol** address to a socket. For example a 32 bit IPV4 address along with a 16 bit port number.

[Fig. Bind]

- Second argument is pointer to a protocol-specific address (ie IPV4)
- Third argument is size of this address structure.
- Servers bind to a port when they start. If a TCP client or server does not do this, Kernel chooses an "Ephemeral" Port for the socket when either connect or listen is called.

[Fig. 2.10] (slide #30)

- bind allows us to specify the IP address, the port both or neither.
- In bind, if we specify a port number of 0 kernel chooses an "ephemeral" Port.
- We can sometimes not specify a IP address so we can receive from any IP address:

Example from Fig. 1.9 where we do not specify an IP address

struck sockaddr_in servaddr

servaddr.sin_addr.s_addr = htonl (INADDR_ANY)

This constant is defined 0

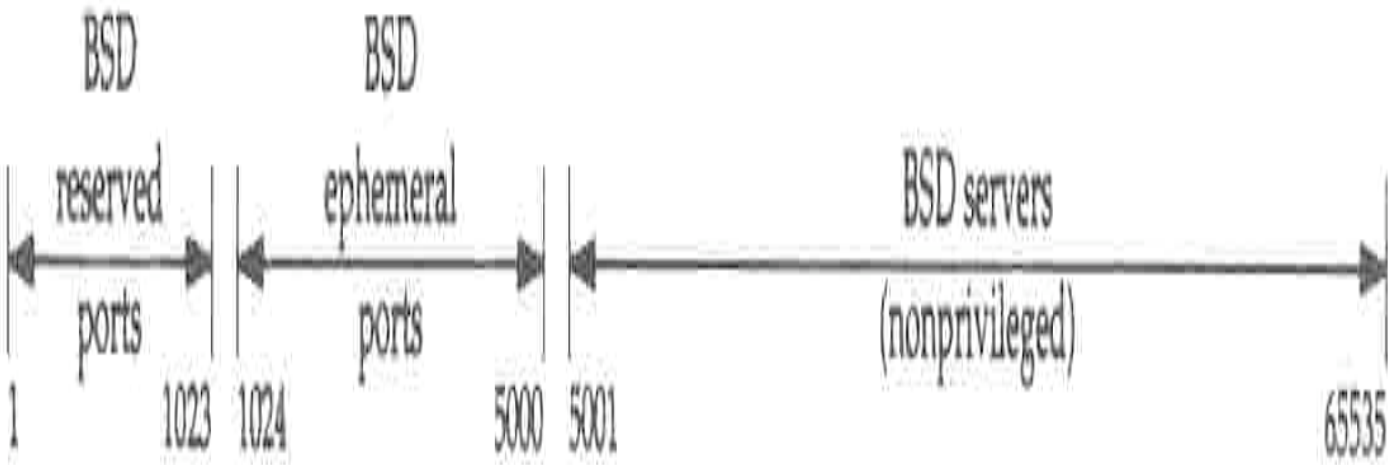
4.4 `bind` Function

The `bind` function assigns a local protocol address to a socket. With the Internet protocols the protocol address is the combination of either a 32-bit IPv4 address or a 128-bit IPv6 address, along with a 16-bit TCP or UDP port number.

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

int bind(int sockfd, const struct sockaddr *myaddr, socklen_t addrlen);
```

Returns: 0 if OK, –1 on error


BSD
reserved
ports
BSD
ephemeral
ports
BSD servers
(nonprivileged)
1
1023
1024
5000
5001
65535

Listen Function

Call By Servers:

1. Converts unconnected socked into a passive socket, meaning kernel should accept incoming connection requests.

   Moves socket from CLOSED to LISTEN in Fig 2.4

2. Second argument specifics maximum number of connections 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.

Accept Function

Called By TCP Server:

Returns next completed connection from front of completed connection queue.

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 4.6). 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: nonnegative descriptor if OK, –1 on error

Close Function

Closes a socket and terminates TCP connection

4.9 `close` Function

The normal Unix `close` function is also used to close a socket and terminate a TCP connection.

```
#include <unistd.h>

int close(int sockfd);
```

Returns: 0 if OK, –1 on error

The default action of `close` with a TCP socket is to mark the socket as closed and return to the process immediately. The socket descriptor is no longer usable by the process: it cannot be used as an argument to `read` or `write`. But TCP will try to send any data that is already queued to be sent to the other end, and after this occurs the normal TCP connection termination sequence takes place (Section 2.5).

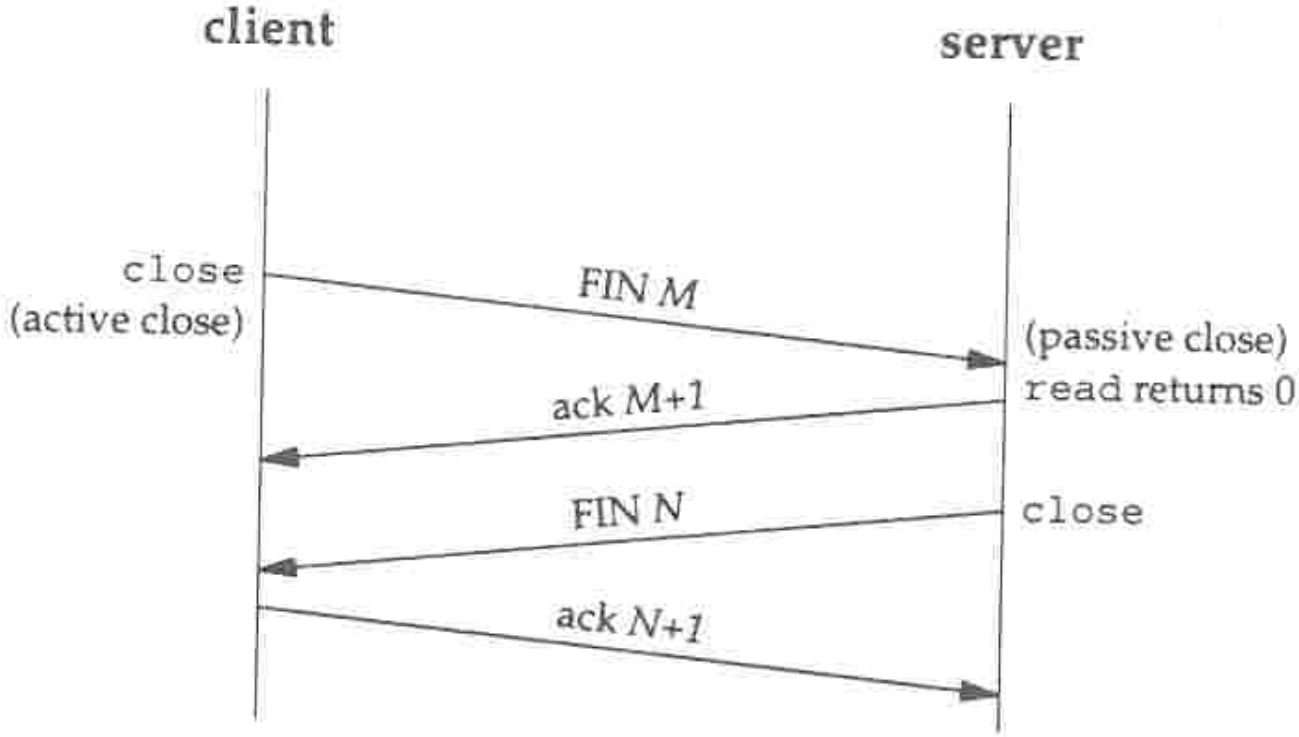


Figure 2.3 Packets exchanged when a TCP connection is closed.

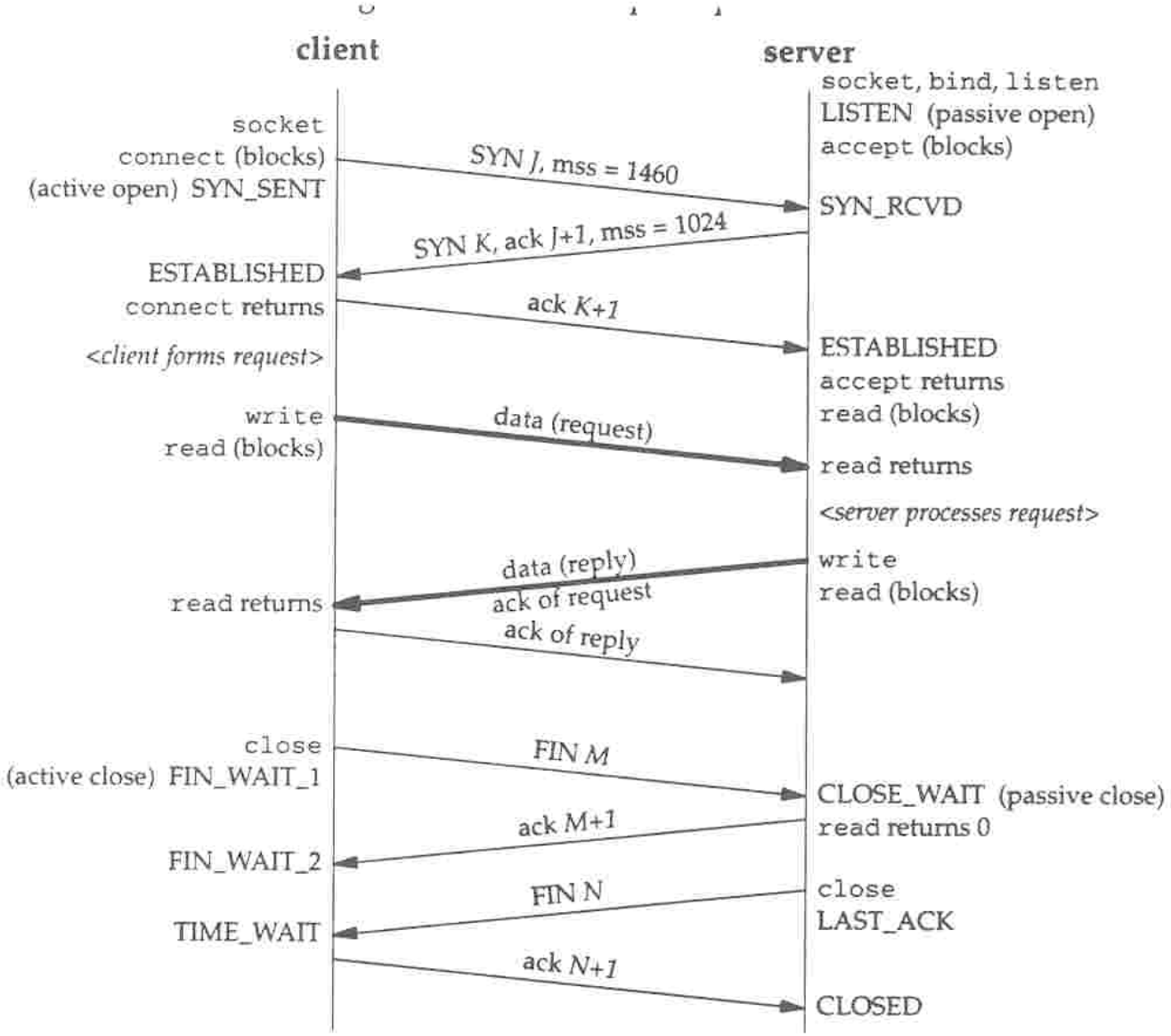


Figure 2.5 Packet exchange for TCP connection.

Example:

Modify Fig 1.9 to print out IP address and port of the client.

[Figure. 3.2 with Fig. 4.11]

Lines 7&8 len will be size of socket address structure.

cliaddr is new to hold client protocol address (IP + Port).

Lines 19 Set len to size of protocol address.

Lines 20 New 2nd & 3rd arguments in Accept.

Lines 21-23 Use inet_ntop to convert 32 Bit IP address in the socket address in the socket address structure into dotted decimal ASCII string.
Use ntohs to convert 16 Bit port number from network BYTE order to host BYTE order.

Note: You will want to use this idea in your programs.

There are details on:

inet_ntop in § 3.7
ntohs in § 3.4

intro/daytimetcpsrv1.c

```
 1 #include    "unp.h"
 2 #include    <time.h>

 3 int
 4 main(int argc, char **argv)
 5 {
 6     int     listenfd, connfd;
 7     socklen_t len;
 8     struct sockaddr_in servaddr, cliaddr;
 9     char    buff[MAXLINE];
10     time_t  ticks;

11     listenfd = Socket(AF_INET, SOCK_STREAM, 0);

12     bzero(&servaddr, sizeof(servaddr));
13     servaddr.sin_family = AF_INET;
14     servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
15     servaddr.sin_port = htons(13);  /* daytime server */
```

```
16      Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));

17      Listen(listenfd, LISTENQ);

18      for ( ; ; ) {
19          len = sizeof(cliaddr);
20          connfd = Accept(listenfd, (SA *) &cliaddr, &len);
21          printf("connection from %s, port %d\n",
22                 Inet_ntop(AF_INET, &cliaddr.sin_addr, buff, sizeof(buff)),
23                 ntohs(cliaddr.sin_port));

24          ticks = time(NULL);
25          snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
26          Write(connfd, buff, strlen(buff));

27          Close(connfd);
28      }
29  }
```

intro/daytimetcpsrv1.c

Figure 4.11 Daytime server that prints client IP address and port.

If we run our new server and then run our client on the same host, connecting to our server twice in a row, we have the following output from the client:

```
solaris % daytimetcpcli 127.0.0.1
Wed Jan 17 15:42:35 1996
solaris % daytimetcpcli 206.62.226.33
Wed Jan 17 15:42:53 1996
```

We first specify the server's IP address as the loopback address (127.0.0.1) and then as its own IP address (206.62.226.33). Here is the corresponding server output.

```
solaris # daytimetcpsrv1
connection from 127.0.0.1, port 33188
connection from 206.62.226.33, port 33189
```

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 Posix.1g.