RV College of Engineering®

Go, change the world

Concurrent Servers

UNIT 2

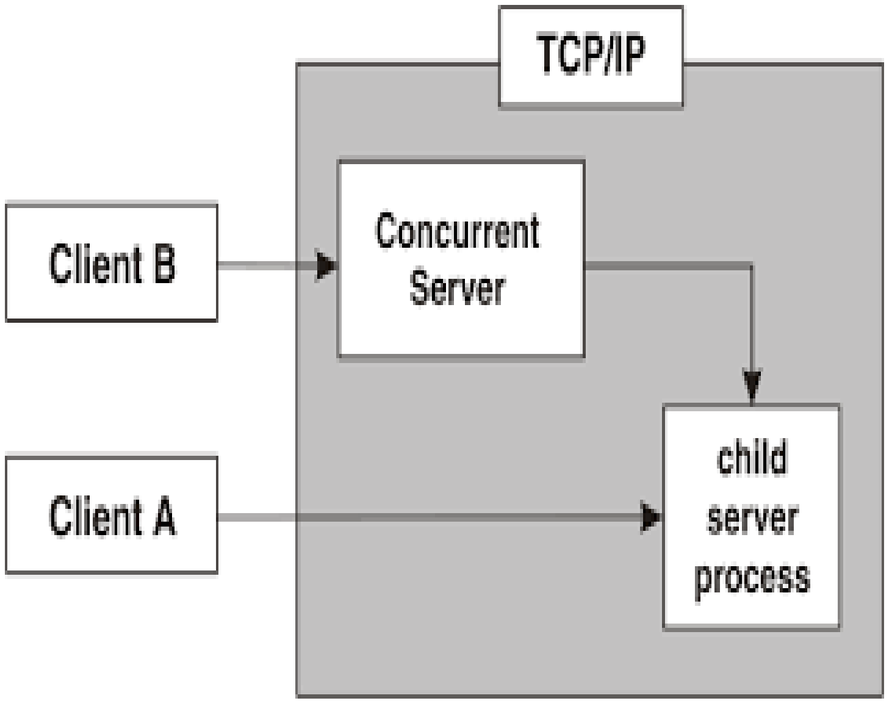
TCP client/server

Contents

1.Socket function
2.connect function, bind, listen, accept, fork, exec functions
3.Concurrent Server
4.close function
5.getsockname and getpeername functions
6.TCP Echo server – main – str_echo
7.TCP Echo client - main – str_echo
8.Normal startup, normal termination

Concurrent Server

Concurrent Server: The **server** can be iterative, i.e. it iterates through each client and serves one request at a time. Alternatively, a **server** can handle multiple clients at the same time in parallel, and this type of a **server** is called a **concurrent server**.



How many concurrent connections can a server handle?
65535 simultaneous connections.

On the TCP level the tuple (source ip, source port, destination ip, destination port) must be unique for each simultaneous connection. That means a single client cannot open more than **65535 simultaneous connections** to a server. But a server can (theoretically) serve **65535 Simultaneous connections** per client.

There are several ways we can implement this server: The simplest technique is to call the Unix **fork()** function. Other techniques are to use threads or to pre-fork a fixed number of children when the function starts.

When the server *receives and accepts the client's connection, it forks a copy of itself and lets the* child handle the client as shown in the figure below:

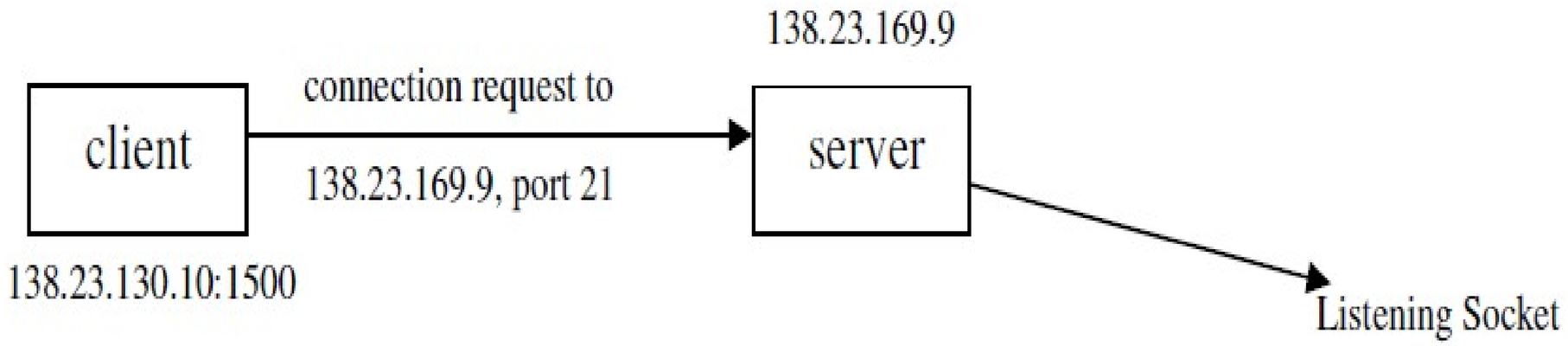


Figure 1: Client with IP address 138.23.130.10 requests to connect to Server (with IP 138.23.169.9 listening from port 21) from its local port number 1500.

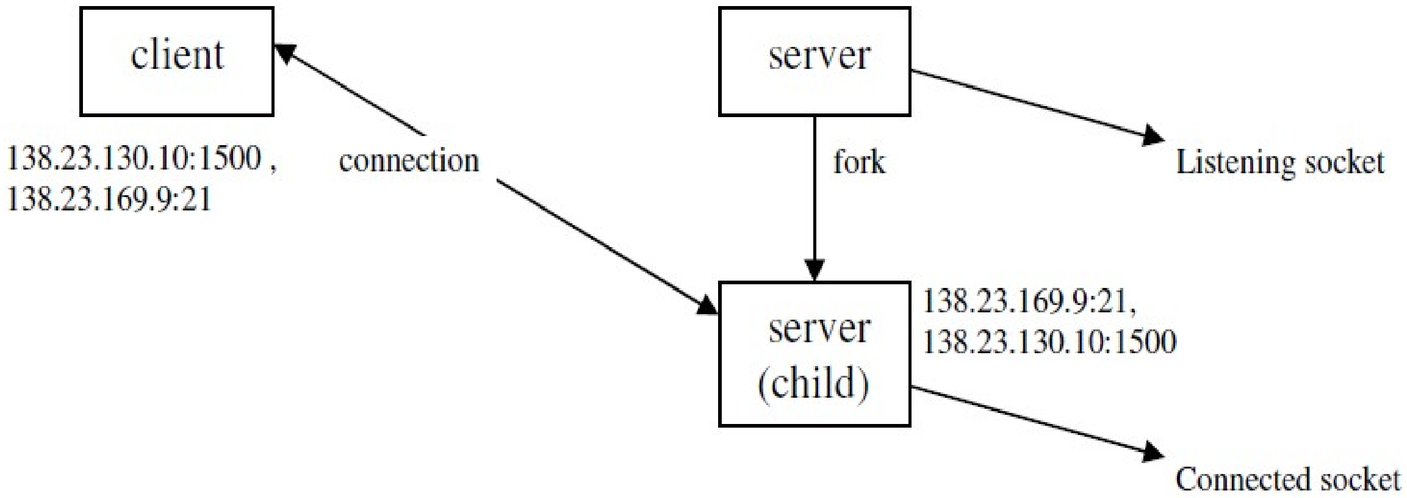


Figure 2: Server fork()s a copy of itself and lets the child handle this client from port 21, but it concurrently keeps listening (by the parent process) from port 21 as well.

Concurrent Server

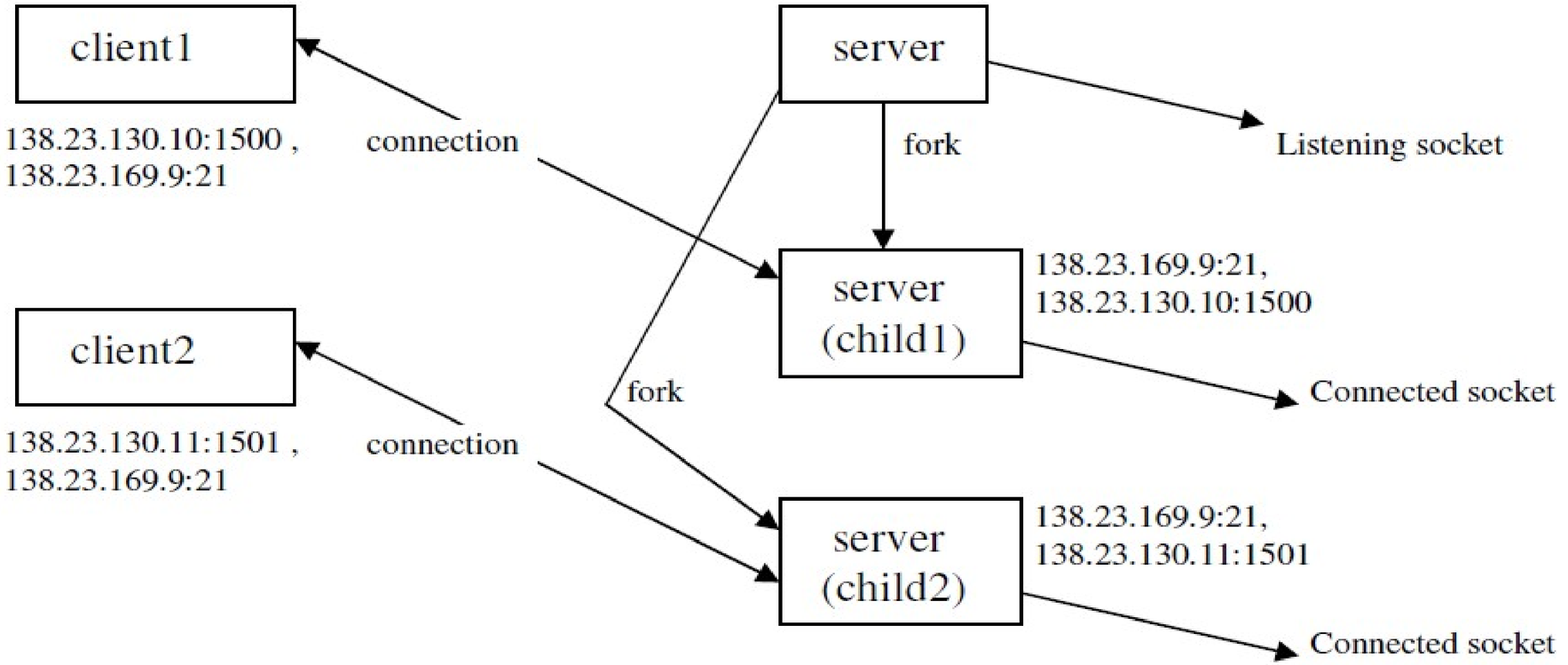


Figure 3: Another client requests to connect to this server. The server fork()s another copy of itself and lets this second child handle this client, whereas it still concurrently keeps listening from port 21 (by the parent process) and keeps serving client1 from port1 (by the first child process).

The listening socket must be distinguished from the connected socket on the server host. Although both sockets use the same local port on the server machine, they are indicated by distinct socket file descriptors, returned server's call of functions socket() and accept() respectively for listening and connected sockets.

Notice from figure 3 above that TCP must look at all four segments in the socket pair to determine which endpoint receives an arriving segment.

- In this figure, there are 3 sockets with the same local port 21 on the server.

- If a segment arrives from 138.23.130.10 port 1500 destined for 138.23.169.9 port 21, it is delivered to the
- first child.

- If a segment arrives from 138.23.130.11 port 1501 destined for 138.23.169.9 port 21, it is delivered to
- the second child.

- All other TCP segments destined for port 21 are delivered to the original listening socket.

The server described in intro/daytimetcpsrv1.c is an **iterative server**.

But when a client request can take longer to service, we do not want to tie up a single server with one client; we want to handle multiple clients at the same time.

The simplest way to write a concurrent server under Unix is to fork a child process to handle each client.

The following code shows the outline for a typical concurrent server:

```
pid_t pid;
int listenfd, connfd;
listenfd = Socket( ... );   /* fill in sockaddr_in{} with server's well-known port */
Bind(listenfd, ... );
Listen(listenfd, LISTENQ);
for ( ; ; )
{ connfd = Accept (listenfd, ... );   /* probably blocks */
if( (pid = Fork()) == 0)
{ Close(listenfd);    /* child closes listening socket */
doit(connfd);         /* process the request */
Close(connfd);        /* done with this client */
exit(0);              /* child terminates */
}
Close(connfd);        /* parent closes connected socket */
}
```

- When a connection is established, accept returns, the server calls fork, and the child process services the client (on connfd, the connected socket) and the parent process waits for another connection (on listenfd, the listening socket).

- The parent closes the connected socket since the child handles the new client.

- We assume that the function doit does whatever is required to service the client.

- When this function returns, we explicitly close the connected socket in the child.

- This is not required since the next statement calls exit, and part of process termination is to close all open descriptors by the kernel.

- Whether to include this explicit call to close or not is a matter of personal programming taste.

Reference count of sockets

Calling close on a TCP socket causes a FIN to be sent, followed by the normal TCP connection termination sequence.

Why doesn't the close of connfd by the parent terminate its connection with the client? To understand what's happening, we must understand that every file or socket has a **reference count**.

The reference count is maintained in the file table entry. This is a count of the number of descriptors that are currently open that refer to this file or socket.

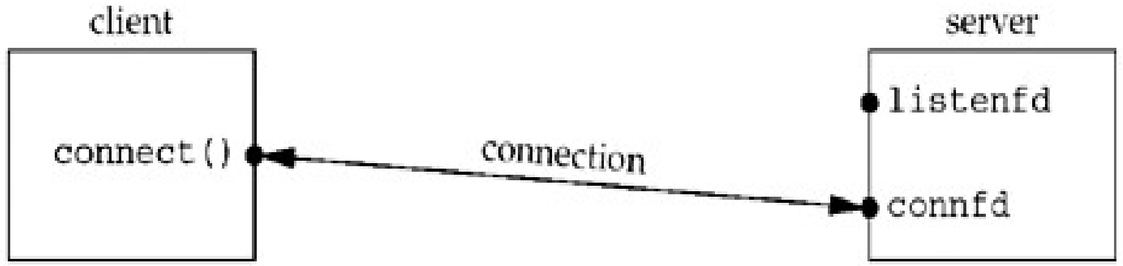
Reference count of sockets

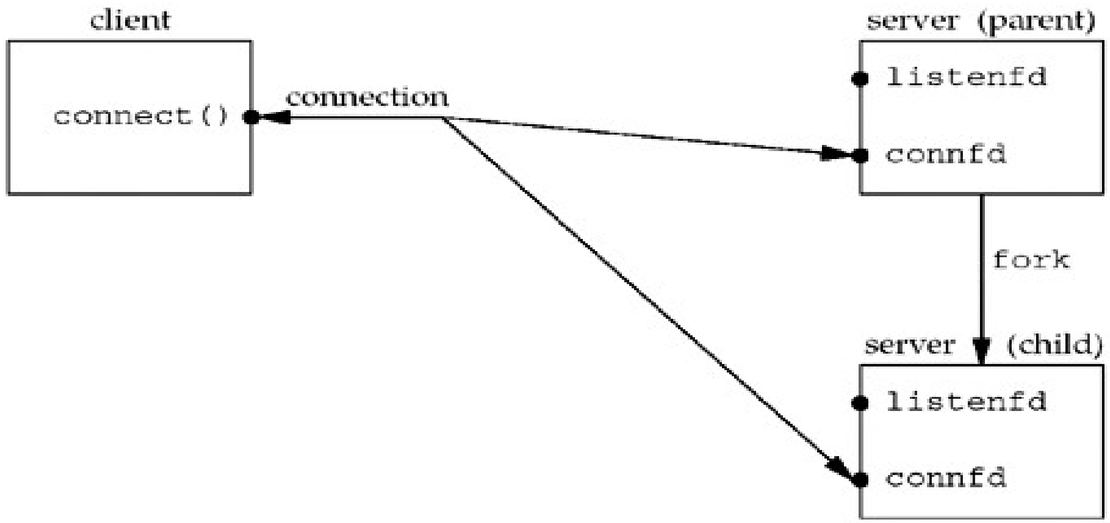
In the above code:

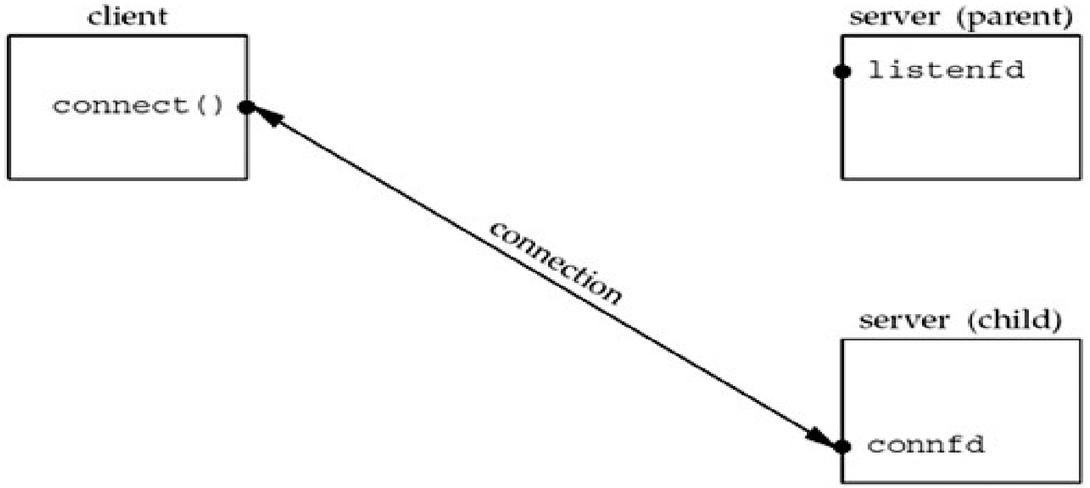
- After socket returns, the file table entry associated with listenfd has a reference count of 1.
- After accept returns, the file table entry associated with connfd has a reference count of 1.
- But, after fork returns, both descriptors are shared (duplicated) between the parent and child, so the file table entries associated with both sockets now have a reference count of 2.
- Therefore, when the parent closes connfd, it just decrements the reference count from 2 to 1 and that is all.
- The actual cleanup and de-allocation of the socket does not happen until the reference count reaches 0.
- This will occur at some time later when the child closes connfd.

Visualizing the sockets and connection

The following figures visualize the sockets and connection in the code above: Before call to accept returns, the server is blocked in the call to accept and the connection request arrives from the client:







This is the desired final state of the sockets. The child is handling the connection with the client and the parent can call accept again on the listening socket, to handle the next client connection.

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.

SO_LINGER socket option (detailed in Section 7.5) lets us change this default action with a TCP socket.

Descriptor Reference Counts

As mentioned in end of Section 4.8, when the parent process in our concurrent server closes the connected socket, this just decrements the reference count for the descriptor.

Since the reference count was still greater than 0, this call to close did not initiate TCP's four-packet connection termination sequence. This is the behavior we want with our concurrent server with the connected socket that is shared between the parent and child.

If we really want to send a FIN on a TCP connection, the shutdown function can be used (Section 6.6) instead of close.

Beware if the parent does not call close for each connected socket returned by accept:
1. The parent will eventually run out of descriptors (there is usually a limit to the number of descriptors that any process can have open at any time).
2. None of the client connections will be terminated. When the child closes the connected socket, its reference count will go from 2 to 1 and it will remain at 1 since the parent never closes the connected socket. This will prevent TCP's connection termination sequence from occurring, and the connection will remain open.

getsockname and getpeername Functions

- getsockname returns the local protocol address associated with a socket.
- getpeername returns the foreign protocol address associated with a socket.

```
#include <sys/socket.h>
int getsockname(int sockfd, struct sockaddr *localaddr, socklen_t *addrlen);
int getpeername(int sockfd, struct sockaddr *peeraddr, socklen_t *addrlen);

/* Both return: 0 if OK, -1 on error */
```

getsockname and getpeername Functions

These two functions are required for the following reasons:

- After connect successfully returns in a TCP client that does not call bind, *getsockname* returns the local IP address and local port number assigned to the connection by the kernel.

- After calling bind with a port number of 0 (telling the kernel to choose the local portnumber), *getsockname* returns the local port number that was assigned. *getsockname* can be called to obtain the address family of a socket.

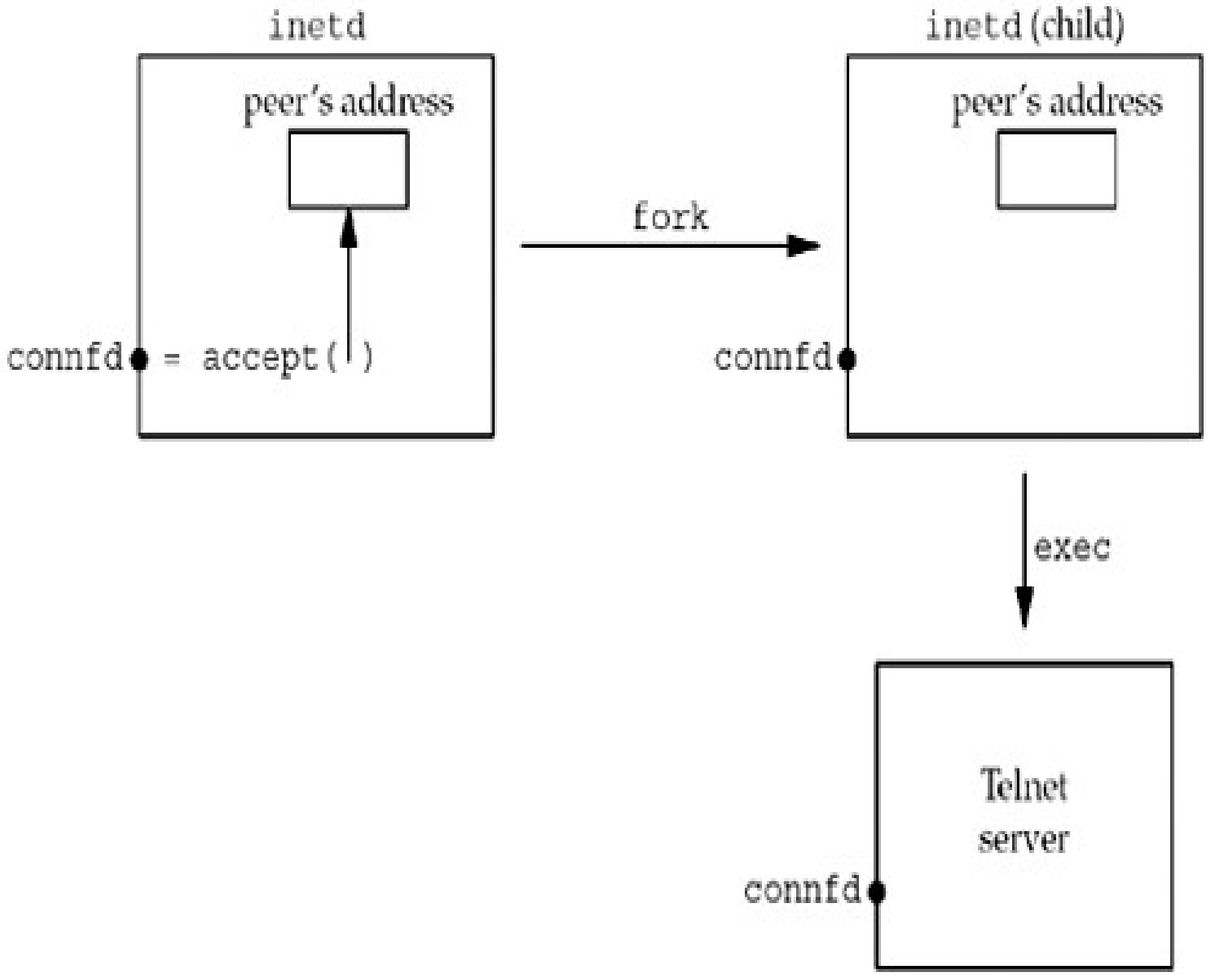
- In a TCP server that binds the wildcard IP address (intro/daytimetcpsrv.c), once a connection is established with a client (accept returns successfully), the server can call *getsockname* to obtain the local IP address assigned to the connection.

- The socket descriptor argument to *getsockname* must be that of the connected socket, and not the listening socket.

- When a server is execed by the process that calls accept, the only way the server can obtain the identity of the client is to call *getpeername.*

getsockname and getpeername Functions

For example, *inetd fork*s and *exec*s a TCP server (following figure):



- *inetd* calls *accept*, which returns two values: the connected socket descriptor (connfd, return value of the function) and the "peer's address" (an Internet socket address structure) that contains the IP address and port number of the client.
- *fork* is called and a child of *inetd* is created, with a copy of the parent's memory image, so the socket address structure is available to the child, as is the connected socket descriptor (since the descriptors are shared between the parent and child).
- When the child *exec*s the real server (e.g. Telnet server that we show), the memory image of the child is replaced with the new program file for the Telnet server (the socket address structure containing the peer's address is lost), and the connected socket descriptor remains open across the exec.
- One of the first function calls performed by the Telnet server is *getpeername* to obtain the IP address and port number of the client.

getsockname and getpeername Functions

In this example, the Telnet server must know the value of connfd when it starts. There are two common ways to do this.

The process calling exec pass it as a command-line argument to the newly execed program.

A convention can be established that a certain descriptor is always set to the connected socket before calling exec.

The second one is what inetd does, always setting descriptors 0, 1, and 2 to be the connected socket.

Example: Obtaining the Address Family of a Socket

The sockfd_to_family function shown in the code below returns the address family of a socket.

https://github.com/shichao-an/unpv13e/blob/master/lib/sockfd_to_family.c

```
 int
sockfd_to_family(int sockfd)
{
        struct sockaddr_storage ss;
    socklen_t len;

    len = sizeof(ss);
    if (getsockname(sockfd, (SA *) &ss, &len) < 0)
        return(-1);
    return(ss.ss_family);
}
```

Example: Obtaining the Address Family of a Socket

The sockfd_to_family function shown in the code below returns the address family of a socket.

https://github.com/shichao-an/unpv13e/blob/master/lib/sockfd_to_family.c

This program does the following:

Allocate room for largest socket address structure. Since we do not know what type of socket address structure to allocate, we use a sockaddr_storage value, since it can hold any socket address structure supported by the system.

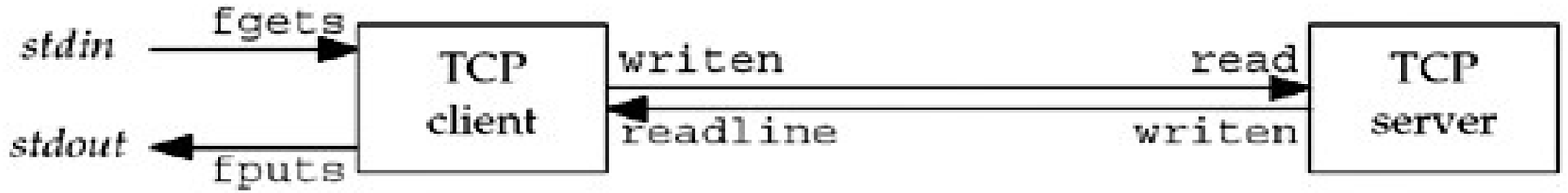
Call getsockname. We call getsockname and return the address family. The POSIX specification allows a call to getsockname on an unbound socket.

Introduction

We will now use the elementary functions from the previous chapter to write a complete TCP client/server example. Our simple example is an echo server that performs the following steps:

1. The client reads a line of text from its standard input and writes the line to the server.
2. The server reads the line from its network input and echoes the line back to the client.
3. The client reads the echoed line and prints it on its standard output.

The figure below depicts this simple client/server:



TCP Client/Server

The echo client/server is a valid, simple example of a network application. To expand this example into your own application, all you need to do is change what the server does with the input it receives from its clients.

Besides running the client/server in normal mode (type in a line and watch it echo), we examine lots of boundary conditions:

- What happens when the client and server are started?
- What happens when the client terminates normally?
- What happens to the client if the server process terminates before the client is done?
- What happens to the client if the server host crashes?

TCP Client/Server

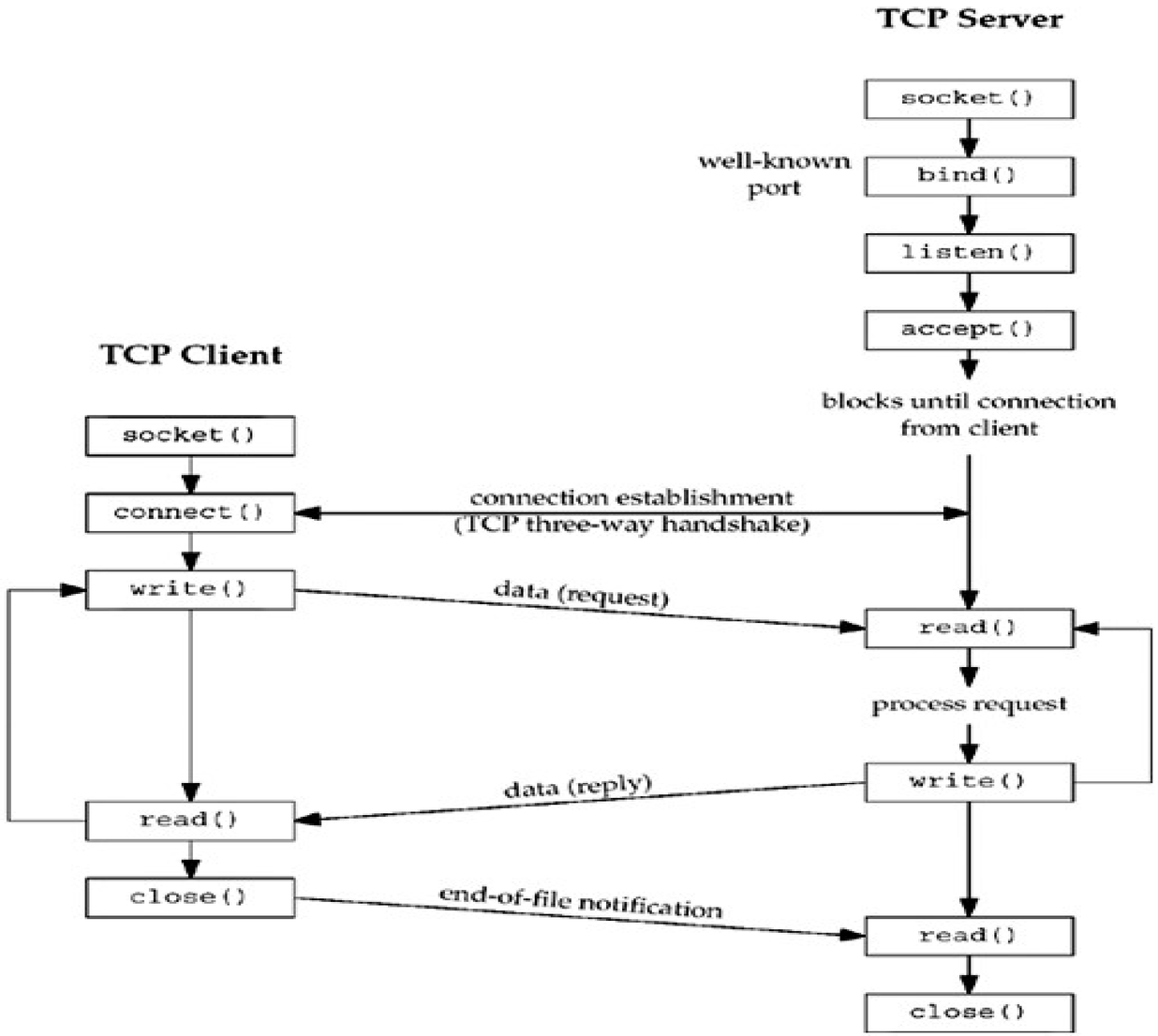
In all these examples, we have "hard-coded" protocol-specific constants such as addresses and ports.

There are two reasons for this:

- We must understand exactly what needs to be stored in the protocol-specific address structures
- We have not yet covered the library functions that can make this more portable

TCP Echo Server: main Function

Our TCP client and server follow the flow of functions that we diagrammed



TCP Echo Server: main Function


```
#include "unp.h"
int
main(int argc, char **argv)
{
int listenfd, connfd;
pid_t childpid;
socklen_t clilen;
struct sockaddr_in cliaddr, servaddr;
listenfd = Socket(AF_INET, SOCK_STREAM, 0);
bzero(&servaddr, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
servaddr.sin_port = htons(SERV_PORT);
Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));
Listen(listenfd, LISTENQ);
for ( ; ; ) {
clilen = sizeof(cliaddr);
connfd = Accept(listenfd, (SA *) &cliaddr, &clilen);
if ( (childpid = Fork()) == 0) { /* child process */
Close(listenfd); /* close listening socket */
str_echo(connfd); /* process the request */
exit(0);
}
Close(connfd); /* parent closes connected socket */
}
}
```

The below code is the concurrent server program:

The above code does the following:

Create socket, bind server's well-known port

- A TCP socket is created.
- An Internet socket address structure is filled in with the wildcard address (INADDR_ANY) and the server's well-known port (SERV_PORT, which is defined as 9877 in our unp.h header).
- Binding the wildcard address tells the system that we will accept a connection destined for any local interface, in case the system is multihomed.
- Our choice of the TCP port number is based on figure shown in the next slide. *It should be greater than 1023 (we do not need a reserved port), greater than 5000 (to avoid conflict with the ephemeral ports allocated by many Berkeley-derived implementations), less than 49152 (to avoid conflict with the "correct" range of ephemeral ports), and it should not conflict with any registered port.*
- The socket is converted into a listening socket by listen.

TCP Echo Server: main Function

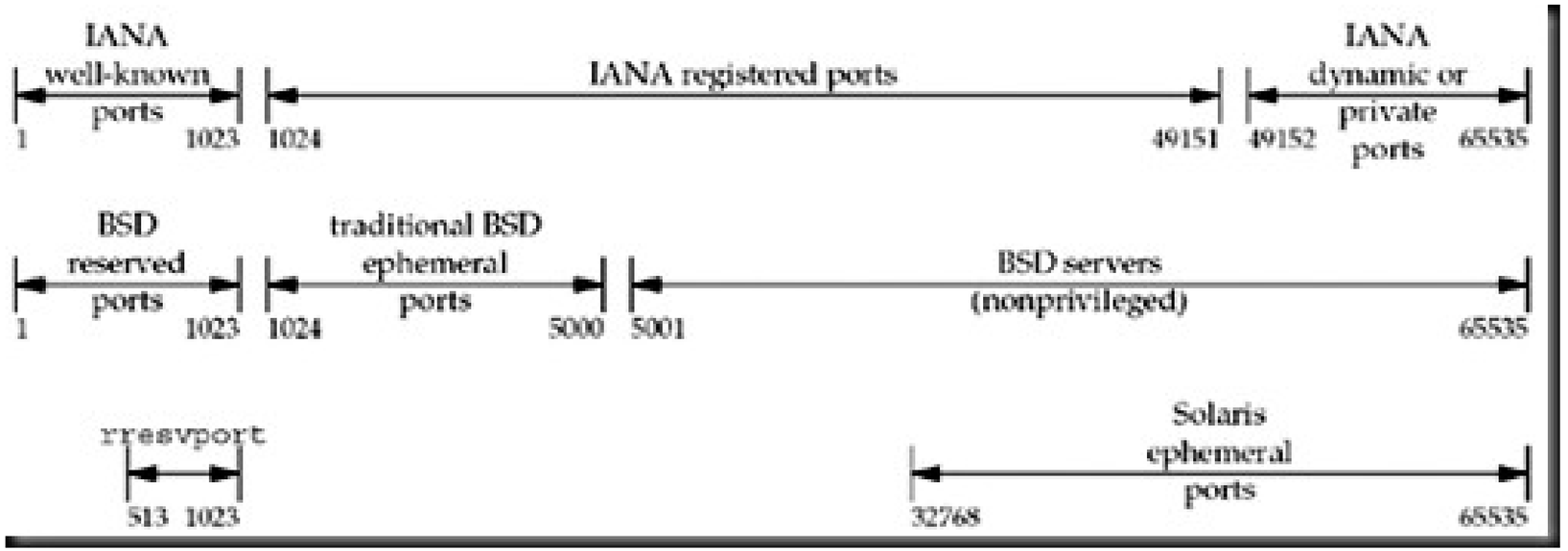


Figure 2.10

Wait for client connection to complete

The server blocks in the call to accept, waiting for a client connection to complete.

Concurrent server

For each client, fork spawns a child, and the child handles the new client. The child closes the listening socket and the parent closes the connected socket. (Section 4.8)

TCP Echo Server: str_echo Function


```
#include "unp.h"
void
str_echo(int sockfd)
{
ssize_t n;
char buf[MAXLINE];
again:
while ( (n = read(sockfd, buf,
MAXLINE)) > 0)
Writen(sockfd, buf, n);
if (n < 0 && errno == EINTR)
goto again;
else if (n < 0)
err_sys("str_echo: read error");
}
```

The function str_echo performs the server processing for each client: It reads data from the client and echoes it back to the client.

The above code does the following:

Read a buffer and echo the buffer:

- read reads data from the socket and the line is echoed back to the client by writen.
- If the client closes the connection (the normal scenario), the receipt of the client's FIN causes the child's read to return 0. This causes the str_echo function to return, which terminates the child.

TCP Echo Client : main Function


```
#include "unp.h"
int
main(int argc, char **argv)
{
int sockfd;
struct sockaddr_in servaddr;
if (argc != 2)
err_quit("usage: tcpcli <IPaddress>");
sockfd = Socket(AF_INET, SOCK_STREAM, 0);
bzero(&servaddr, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_port = htons(SERV_PORT);
Inet_pton(AF_INET, argv[1],
&servaddr.sin_addr);
Connect(sockfd, (SA *) &servaddr,
sizeof(servaddr));
str_cli(stdin, sockfd); /* do it all */
exit(0);
}
```

The above code does the following:

Create socket, fill in Internet socket address structure

A TCP socket is created and an Internet socket address structure is filled in with the server's IP address and port number. The server's IP address is taken from the command-line argument and the server's well-known port (SERV_PORT) is from our unp.h header.

Connect to server

connect establishes the connection with the server. The function str_cli handles the rest of the client processing.

TCP Echo Client : str_cli Function


```
#include "unp.h"
void
str_cli(FILE *fp, int sockfd)
{
char sendline[MAXLINE], recvline[MAXLINE];

while (Fgets(sendline, MAXLINE, fp) != NULL)
 {
        Writen(sockfd, sendline, strlen(sendline));

        if (Readline(sockfd, recvline, MAXLINE) == 0)
        err_quit("str_cli: server terminated prematurely");

     Fputs(recvline, stdout);
}
}
```

The str_cli function handles the client processing loop: It reads a line of text from standard input, writes it to the server, reads back the server's echo of the line, and outputs the echoed line to standard output.

The above code does the following:

Read a line, write to server

fgets reads a line of text and writen sends the line to the server.

Read echoed line from server, write to standard output

readline reads the line echoed back from the server and fputs writes it to standard output.

Return to main

The loop terminates when fgets returns a null pointer, which occurs when it encounters either an end-of-file (EOF) or an error. Our Fgets wrapper function checks for an error and aborts if one occurs, so Fgets returns a null pointer only when an end-of-file is encountered.

Normal Startup

Although the TCP example is small, it is essential that we understand:
How the client and server start and end,
What happens when something goes wrong:

- the client host crashes,
- the client process crashes,
- network connectivity is lost

Only by understanding these boundary conditions, and their interaction with the TCP/IP protocols, can we write robust clients and servers that can handle these conditions.

Start the server in the background

First, we start the server in the background:

```
linux % tcpserv01 &
[1] 17870
```

When the server starts, it calls socket, bind, listen, and accept, blocking in the call to accept.

Normal Startup

Run netstat

Before starting the client, we run the netstat program to verify the state of the server's listening socket.

```
linux % netstat -a
Active Internet connections (servers and established)
Proto      Recv-Q      Send-Q      Local Address      Foreign Address      State
tcp        0           0           *:9877             *:*                  LISTEN
```

- This command shows the status of all sockets on the system. We must specify the -a flag to see listening sockets.
- In the output, a socket is in the LISTEN state with a wildcard for the local IP address and a local port of 9877. netstat prints an asterisk for an IP address of 0 (INADDR_ANY, the wildcard) or for a port of 0.

Normal Startup

Start the client on the same host

We then start the client on the same host, specifying the server's IP address of 127.0.0.1 (the loopback address). We could have also specified the server's normal (nonloopback) IP address.

linux % tcpcli01 127.0.0.1

The client calls socket, and connect which causes TCP's three-way handshake. When the three-way handshake completes, connect returns in the client and accept returns in the server. The connection is established. The following steps then take place:

1. The client calls str_cli, which will block in the call to fgets.
2. When accept returns in the server, it calls fork and the child calls str_echo. This function calls readline, which calls read, which blocks while waiting for a line to be sent from the client.
3. The server parent, on the other hand, calls accept again, and blocks while waiting for the next client connection.

Notes from the previous three steps:

- All three processes are asleep (blocked): client, server parent, and server child.
- We purposely list the client step first, and then the server steps when the three-way handshake completes. This is because accept returns one-half of the RTT after connect returns.
  - On the client side, connect returns when the second segment of the handshake is received
  - On the server side, accept does not return until the third segment of the handshake is received.

Normal Startup

Run netstat after connection completes

Since we are running the client and server on the same host, netstat now shows two additional lines of output, corresponding to the TCP connection:

linux % netstat -a

Active Internet connections (servers and established)

Proto	*Recv-Q*	*Send-Q*	*Local Address*	*Foreign Address*	*State*
tcp	*0*	*0*	*local host:9877*	*localhost:42758*	*ESTABLISHED*
tcp	*0*	*0*	*local host:42758*	*localhost:9877*	*ESTABLISHED*
Tcp	*0*	*0*	**:9877*	**:**	*LISTEN*

- The first ESTABLISHED line corresponds to the server child's socket, since the local port is 9877.
- The second ESTABLISHED lines is the client's socket, since the local port is 42758.

If we were running the client and server on different hosts, the client host would display only the client's socket, and the server host would display only the two server sockets.

Normal Startup

Run ps to check process status and relationship

linux % ps -t pts/0 -o pid,ppid,tty,stat,args,wchan

PID	PPID	TT	STAT	COMMAND	WCHAN
22038	22036	pts/6	S	-bash	wait4
17870	22038	pts/6	S	./tcpserv01	wait_for_connect
19315	17870	pts/6	S	./tcpserv01	tcp_data_wait
19314	22038	pts/6	S	./tcpcli01 127.0	read_chan

Very specific arguments to ps are used:

- The TT column (pts/6): client and server are run from the same window, pseudo-terminal number 6.
- The PID and PPID columns show the parent and child relationships.
  - The first tcpserv01 line is the parent and the second tcpserv01 line is the child since the PPID of the child is the parent's PID.
  - The PPID of the parent is the shell (bash).
- The STAT column for all three of our network processes is "S", meaning the process is sleeping (waiting for something).
- The WCHAN column specifies the condition when a process is asleep.
  - Linux prints wait_for_connect when a process is blocked in either accept or connect, tcp_data_wait when a process is blocked on socket input or output, or read_chan when a process is blocked on terminal I/O.
  - In ps(1), WCHAN column indicates the name of the kernel function in which the process is sleeping, a "-" if the process is running, or a "*" if the process is multi-threaded and ps is not displaying threads.

Normal Termination

At this point, the connection is established and whatever we type to the client is echoed back.

```
linux % tcpcli01 127.0.0.1        # we showed this line earlier
hello, world                      # we now type this
hello, world                      # and the line is echoed
good bye
good bye
^D                                # Control-D is our terminal EOF character
```

Normal Termination

If we immediately execute netstat, we have:

```
linux % netstat -a | grep 9877
 tcp        0      0      *:9877                    *:*                    LISTEN
tcp         0      0      localhost:42758           localhost:9877    TIME_WAIT
```

This time we pipe the output of netstat into grep, printing only the lines with our server's well-known port:

- The client's side of the connection (since the local port is 42758) enters the TIME_WAIT state
- The listening server is still waiting for another client connection.

Normal Termination

The following steps are involved in the normal termination of client and server:

When we type our EOF character, fgets returns a null pointer and the function str_cli (Section 5.5) returns.

str_cli returns to the client main function (Section 5.5), which terminates by calling exit.

Part of process termination is the closing of all open descriptors, so the client socket is closed by the kernel. This sends a FIN to the server, to which the server TCP responds with an ACK. This is the first half of the TCP connection termination sequence. At this point, the server socket is in the CLOSE_WAIT state and the client socket is in the FIN_WAIT_2 state (Figure 2.4 and Figure 2.5)

When the server TCP receives the FIN, the server child is blocked in a call to read (Section 3.8), and read then returns 0. This causes the str_echo function to return to the server child main. [Errata] [p128]

The server child terminates by calling exit. (Section 5.2)

All open descriptors in the server child are closed.

The closing of the connected socket by the child causes the final two segments of the TCP connection termination to take place: a FIN from the server to the client, and an ACK from the client.

Finally, the SIGCHLD signal is sent to the parent when the server child terminates.

This occurs in this example, but we do not catch the signal in our code, and the default action of the signal is to be ignored. Thus, the child enters the zombie state. We can verify this with the ps command

Normal Termination

```
linux % ps -t pts/6 -o pid,ppid,tty,stat,args,wchan
PID    PPID TT       STAT COMMAND                     WCHAN
22038 22036 pts/6    S    -bash                       read_chan
17870 22038 pts/6    S    ./tcpserv01                 wait_for_connect
19315 17870 pts/6    Z    [tcpserv01 <defu            do_exit
```

The STAT of the child is now Z (for zombie).

We need to clean up our zombie processes and doing this requires dealing with Unix signals. The next section will give an overview of signal handling.