NETWORK PROGRAMMING
UNIT 3 ELEMENTARY TCP SOCKETS

Author

Prof.Prasad M.Sawant

Assistant Professor

Department Of Computer Science

Pratibha College Of Commerce And Computer Studies ,Chichwad Pune

Prasad Sawant

UNIT 3 ELEMENTARY TCP SOCKETS

INTRODUCTION :

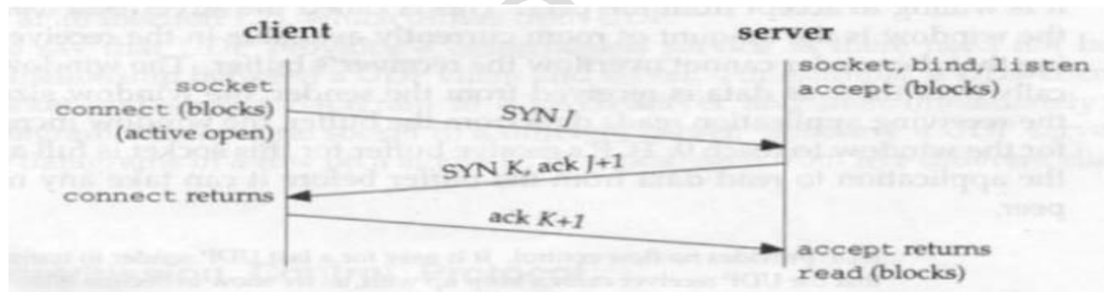
ELEMENTARY TCP SOCKETS

1. ***socket* function**
2. ***connect* function**
3. ***bind* function**
4. ***listen* function**
5. ***accept* function**
6. ***fork* and *exec* functions**
7. **Concurrent servers**
8. ***close* function**
9. ***getsockname* and *getpeername*** functions , or complicated terms.

Prof. Sawant

TCP Three-Way Handshake

To understand the ***connect, accept and close*** functions and to debug TCP application using *neststat* we need to follow the state transition diagram.



The **three way handshake** that take place is as follows:

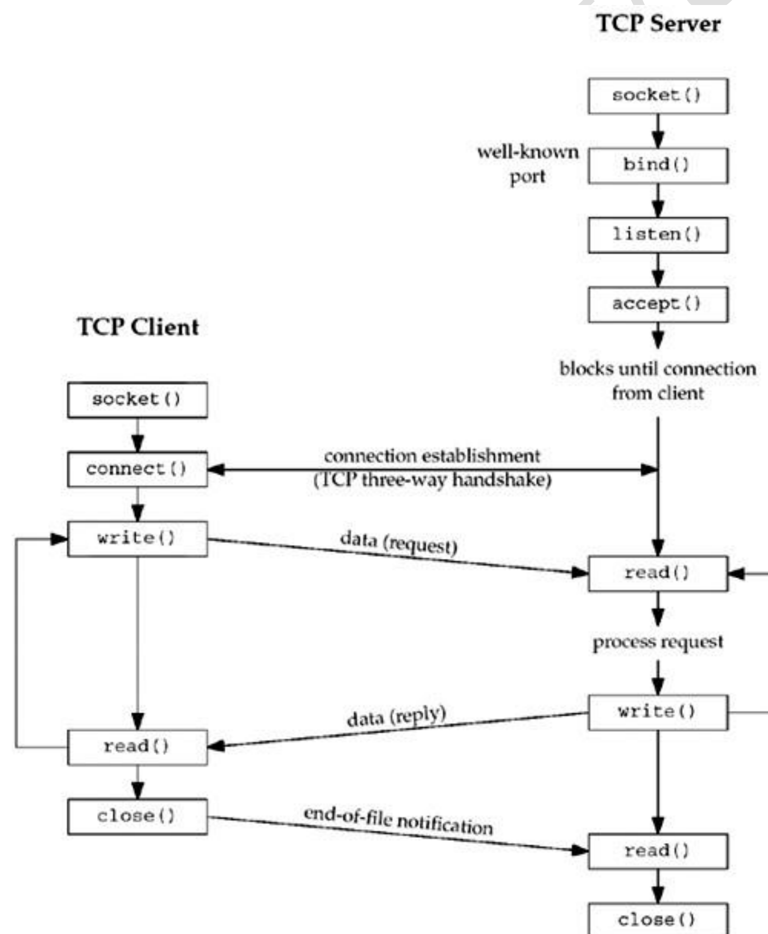
1. The server must be prepared to accept an incoming connection. This is normally done by calling ***socket, bind and listen functions*** and is called passive open.
2. The *client* issues an ***active open*** by calling ***connect***. This causes the client TCP to send ***SYN* segment** to tell the server that the client's initial **sequence number** for the data that the client will send on that connection. No data is sent with SYN. It contains an **IP header, TCP header and possible TCP options.**

© Prof.Prasad Sawant PCCCS Chichwad
Courtesy Unix Network Programming, Volume 1: The Sockets Networking API W. Richard Stevens

3. The server acknowledges the client's SYN and sends its own SYN and the ACK of the client's SYN in a single segment.
4. The client must ACK the server's SYN .

System calls used with sockets:

Socket calls are those functions that provide access to the underlying functionality and utility routines that help the programmer. A socket can be used by client or by a server, for a stream transfer (TCP) or datagram (UDP) communication with a specific endpoints address. Following figure shows a time line of the typical scenario that takes place between client and server.



First server is started, then sometimes later a client is started that connects to the server. The client sends a request to the server, the server processes the request, and the server sends back

© Prof.Prasad Sawant PCCCS Chichwad
Courtesy Unix Network Programming, Volume 1: The Sockets Networking API W. Richard Stevens

reply to the client. This continues until the client closes its end of the connection, which sends an end of file notification to the server. The server then closes its end of the connections and either terminates or waits for a new connection.

Prasad Sawant

© Prof.Prasad Sawant PCCCS Chichwad
Courtesy Unix Network Programming, Volume 1: The Sockets Networking API W. Richard Stevens

UNIT 3 ELEMENTARY TCP SOCKETS

Socket function:

#include socket (int family, int type, int protocol);

returns negative descriptor if OK & –1 on error.

Arguments specify the protocol family and the protocol or type of service it needs (stream or datagram). The protocol argument is set to 0 except for raw sockets.

Family	Description		Type	Description
AF_INET	IPv4 protocols		SOCK_STREAM	Stream Socket
AF_INET6	IpV6 Protocols		SOCK_DGRAM	Datagram socket
AF_ROUTE	Unix domain protocol		SOCK_RAW	Raw socket
AF_ROUTE	Routing sockets			
AF_KEY	Key Socket			

Not all combinations of socket family and type are valid. Following figure shows the valid combination.

	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

connect Function

The connect function is by a TCP client to establish an active connection with a remote server. The arguments allow the client to specify the remote end points which includes the remote machines IP address and protocol port number.

```
# include <sys/socket.h>
int connect (int sockfd, const struct sockaddr * servaddr, socklen_t addrelen)
returns 0 if ok -1 on error.
```

sockfd is the socket descriptor that was returned by the socket function. The second and third arguments are a pointer to a socket address structure and its size.

Prasad Sawant

© Prof.Prasad Sawant PCCCS Chichwad
Courtesy Unix Network Programming, Volume 1: The Sockets Networking API W. Richard Stevens

In case of TCP socket, the **connect**() function **initiates TCP's three way handshake**. The function returns only when the connection is established or an error occurs. Different type of **errors** are

1. If the client TCP receives no response to its **SYN** segment, **ETIMEDOUT** is returned. This is done after the **SYN** is sent after, ***6sec*, *24sec*** and if no response is received after a total period of ***75 seconds***, the error is returned.

2. In case for **SYN** request, a **RST** is returned (hard error), this indicates that no process is waiting for connection on the server. In this case **ECONNREFUSED** is returned to the client as soon the **RST** is received. RST is received when (a) a **SYN** arrives for a port that has no listening server (b) when **TCP** wants to abort an existing connection, (c) when **TCP** receives a segment for a connection does not exist.

3. If the **SYN** elicits an **ICMP** destination is unreachable from some intermediate router, this is considered a soft error. The client server saves the message but keeps sending **SYN** for the time period of 75 seconds. If no response is received, **ICMP** error is returned as **EHOSTUNREACH** or **ENETUNREACH**.

bind() Function:

When a socket is created, it does not have any notion of end points addresses An application calls bind to specify the local endpoint address for a socket. That is the bind function assigns a local port and address to a socket.

```
#include <sys/socket.h>
int bind (int sockfd, const struct sockaddr *myaddr, socklen_t addrlen);
```

Returns: 0 if OK,-1 on error

The second argument is a pointer to a **protocol specific address** and the third argument is the **size of this address structure**. Server binds their well-known port when they start. (A TCP client does not bind an IP address to its socket.)

listen Function:

The listen function is called only by **TCP** server and it performs following functions.

© Prof.Prasad Sawant PCCCS Chichwad
Courtesy Unix Network Programming, Volume 1: The Sockets Networking API W. Richard Stevens

Prasad Sawant

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 TCP transmission diagram the call to listen moves the socket from the **CLOSED** state to the **LISTEN** state.

```
#include <sys/socket.h>
#int listen (int sockfd, int backlog);
```

Returns: 0 if OK, -1 on error

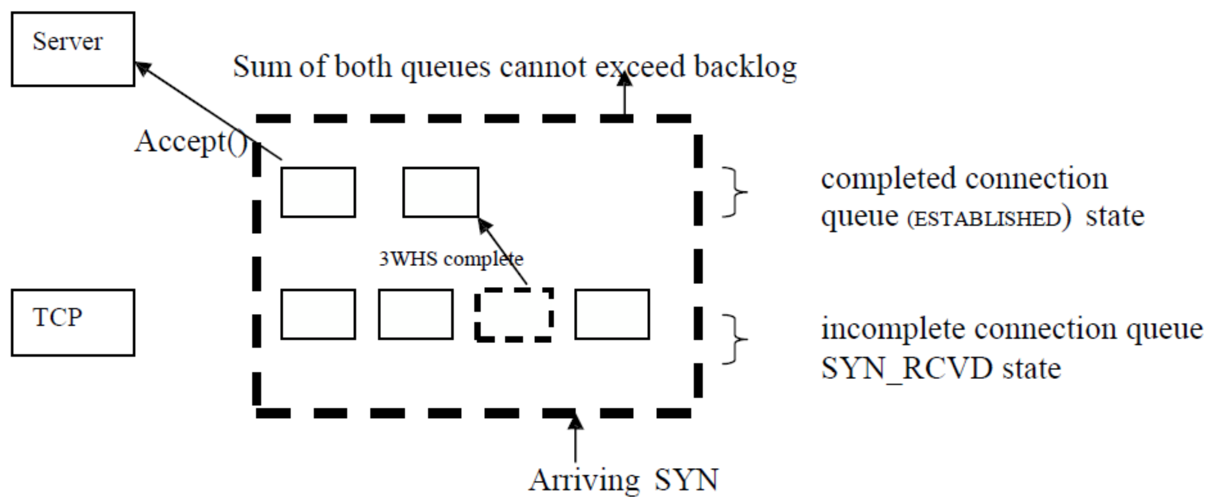
- The second argument to this function specifies the maximum number of connections that the kernel should queue for this socket.
- This function is normally called after both the socket and bind functions and must be called before calling the accept function.

The kernel maintains two queues and the backlog is the sum of these two queues

- 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 handshakes. These sockets are in the **SYN_RECD** state.
- A **Completed Connection Queue** which contains an entry for each client with whom three handshakes has completed. These sockets are in the **ESTABLISHED** state.
- Following figure depicts these two queues for a given listening socket.

Prasad Sawant

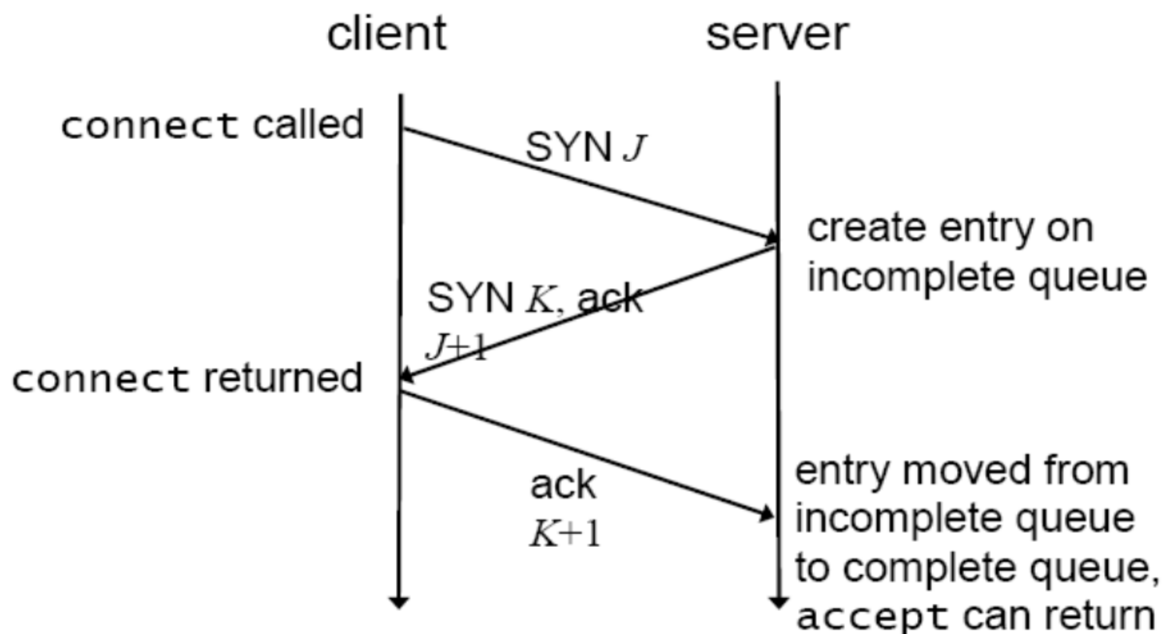
© Prof.Prasad Sawant PCCCS Chichwad
Courtesy Unix Network Programming, Volume 1: The Sockets Networking API W. Richard Stevens



The two queues maintained by TCP 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 clients SYN. 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 the entry times out. If the three way hand shake completes normally, the entry moves from the incomplete queue to the completed queue. When the process calls accept, 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. If the queue are full when a client arrives, TCP ignores the arriving SYN, it does not send an RST. This is because the condition is considered temporary and the client TCP will retransmit its SYN with the hope of finding room in the queue.

© Prof.Prasad Sawant PCCCS Chichwad
Courtesy Unix Network Programming, Volume 1: The Sockets Networking API W. Richard Stevens



accept Function

accept is called by a TCP server to return the next completed connection from the from of the completed connection queue. If the completed queue is empty, the process is put to sleep.

```
#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 before the call, we set the integer value pointed to by **addrlen* to the size of the socket address structure pointed to by *cliaddr* and 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 that was 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 and we call the return value from a accept the connected socket.

© Prof.Prasad Sawant PCCCS Chichwad
Courtesy Unix Network Programming, Volume 1: The Sockets Networking API W. Richard Stevens

fork function

fork is the function that enables the Unix to create a new process

```
#include <unistd.h>
pid_t fork(void);
```

Returns: 0 in child, process ID of child in parent, -1 on error

Typical uses of fork function:

1. A process makes a copy of itself so that one copy can handle one operation while the other copy does another task. This is normal way of working in a network servers.
2. A process wants to execute another program. Since the only way to create a new process is by calling fork, the process first calls fork to make a copy of itself, and then one of the copies(typically the child process) calls exec function to replace itself with a the new program. This is typical for program such as shells.
3. ***fork*** function although called once, it returns twice. It returns once in the calling process (called the parent) with a return value that is process ID of the newly created process (the child). It also returns once in the child, with a return value of 0. Hence the return value tells the process whether it is the parent or the child.
4. The reason ***fork*** returns 0 in the child, instead of parent's process ID is because a child has only one parent and it can always obtain the parent's process ID by calling ***getppid*** A parent, on the other hand, can have any number of children, and there is no way to obtain the process Ids of its children. If the parent wants to keep track of the process Ids of all its children, it must record the return values form ***fork***

Prasad Sawant

© Prof.Prasad Sawant PCCCS Chichwad
Courtesy Unix Network Programming, Volume 1: The Sockets Networking API W. Richard Stevens

exec function

The only way in which an executable program file on disk is executed by Unix is for an existing process to call one of the six ***exec*** functions. ***exec*** replaces the current process image with the new program file and this new program normally starts at the main function. The process ID does not change. The process that calls the ***exec*** is the *calling process* and the newly executed program as the *new program*.

```
#include <unistd.h>
int execl (const char *pathname, const char *arg0, ... /* (char *) 0 */ );
int execv (const char *pathname, char *const argv[]);
int execle (const char *pathname, const char *arg0, ...
                    /* (char *) 0, char *const envp[] */ );
int execve (const char *pathname, char *const argv[], char *const envp[]);
int execlp (const char *filename, const char *arg0, ... /* (char *) 0 */ );
int execvp (const char *filename, char *const argv[]);
```

All six return: -1 on error, no return on success

l : needs a list of arguments

v : needs an *argv[]* vector (*l* and *v* are mutually exclusive)

e : needs an *envp[]* array

p : needs the PATH variable to find the executable file

The differences in the six ***exec*** functions are:

a. whether the program file to execute is specified by a file name or a pathname.

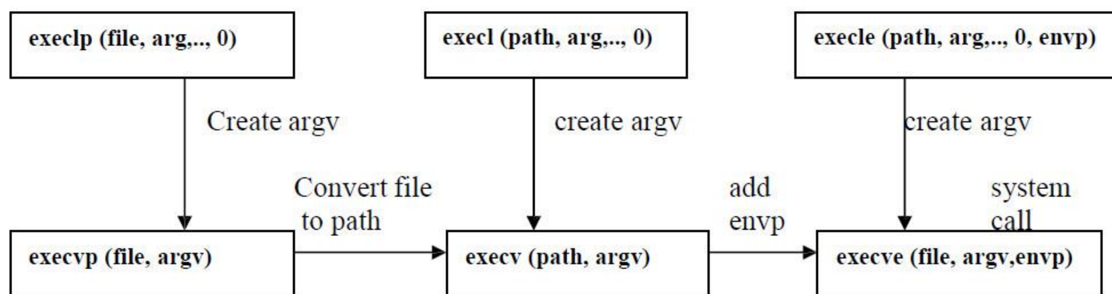
b. Whether the arguments to the new program are listed one by one or reference through an array of pointers.

c. Whether the environment of the calling process is passed to the new program or whether a new environment is specified.

The relationship among these six functions is shown in the following figure . Normally only ***execve*** is a system call within the kernal and the other five are library functions that call ***execve***.

© Prof.Prasad Sawant PCCCS Chichwad
Courtesy Unix Network Programming, Volume 1: The Sockets Networking API W. Richard Stevens

Prasad Sawant



1. The three functions in the top row specify each argument string as a separate argument to the `exec` function, with a null pointer terminating the variable number of arguments. The three functions in the second row have an argv array, containing pointers to the argument strings. This argv array must contain a null pointer to specify its end, since a count is not specified.
2. The two functions in the left column specify a filename argument. This is converted into a pathname using the current `PATH` environment variable. If the filename argument to `execlp` or `execvp` contains a slash (/) anywhere in the string, the `PATH` variable is not used. The four functions in the right two columns specify a fully qualified pathname argument.
3. The four functions in the left two columns do not specify an explicit environment pointer. Instead, the current value of the external variable `environ` is used for building an environment list that is passed to the new program. The two functions in the right column specify an explicit environment list. The envp array of pointers must be terminated by a null pointer.

Prasad Sawant

© Prof.Prasad Sawant PCCCS Chichwad
Courtesy Unix Network Programming, Volume 1: The Sockets Networking API W. Richard Stevens

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. That is , it cannot be used as an argument to read or write.

getsockname () and getpeername():

These two functions return either the local protocol address associated with a socket or the foreign 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

These functions are required for the following reasons.

1. After connect successfully returns a TCP client that does not call **bind(), getsocketname()** returns the local IP address and local port number assigned to the connection by the kernel
2. After calling bind with a port number of 0, **getsockname()** returns the local port number that was assigned

© Prof.Prasad Sawant PCCCS Chichwad
Courtesy Unix Network Programming, Volume 1: The Sockets Networking API W. Richard Stevens

Prasad Sawant

3. When the server is ***exce***ed by the process that calls **accept(),** the only way the server can obtain the identity of the client is to call **getpeername().**

Prasad Sawant

© Prof.Prasad Sawant PCCCS Chichwad
Courtesy Unix Network Programming, Volume 1: The Sockets Networking API W. Richard Stevens

Concurrent Servers

A server that handles a simple program such as daytime server is a ***iterative*** server. But when the client request can take longer to service, the server should not be tied upto a single client. The server must be capable of handling multiple clients at the same time. The simplest way to write a ***concurren***t server under Unix is to ***fork*** a child process to handle each client.

Outline for 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 */
}
```


© Prof.Prasad Sawant PCCCS Chichwad
Courtesy Unix Network Programming, Volume 1: The Sockets Networking API W. Richard Stevens

Prasad Sawant

When a connection is established , ***accept*** returns, the server calls ***fork,*** and then 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 this new client.

In the above program, the function ***doit*** does whatever is required to service the client. When this functions 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 closing all open descriptors by the kernal. Whether to include this explicit call to ***close*** or not is a matter of personal programming taste.

Below fig. shows the status of the client and server while the server is blocked in the call to accept and the connection request arrives from the client.

Status of client/server before call to accept returns.

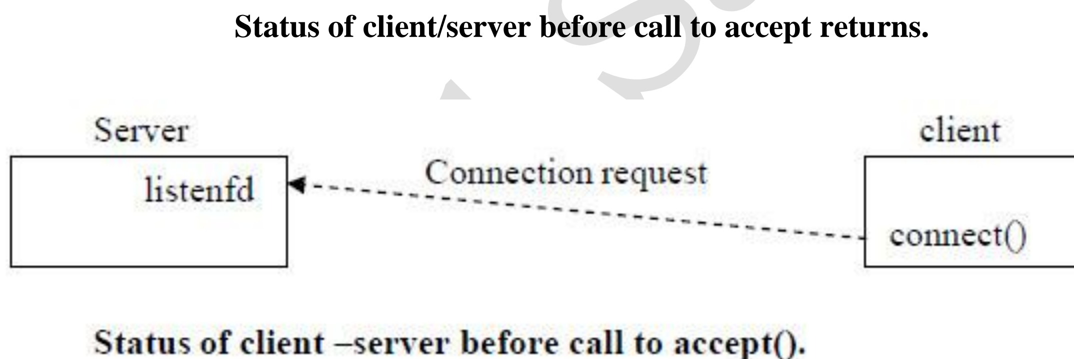


Fig.A 1

Immediately after accept returns, we have the scenario shown in Figure B. The connection is accepted by the kernel and a new socket, connfd, is created. This is a connected socket and data can now be read and written across the connection.

© Prof.Prasad Sawant PCCCS Chichwad
Courtesy Unix Network Programming, Volume 1: The Sockets Networking API W. Richard Stevens

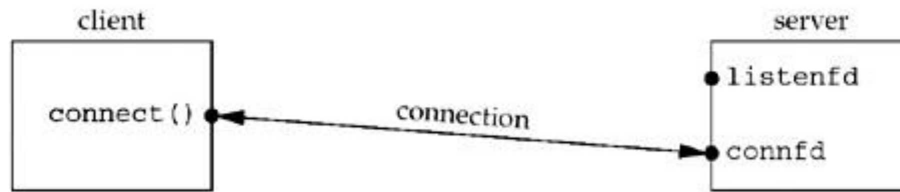


Fig.B 1

The next step in the concurrent server is to call fork. Fig c Show the status after fork returns.

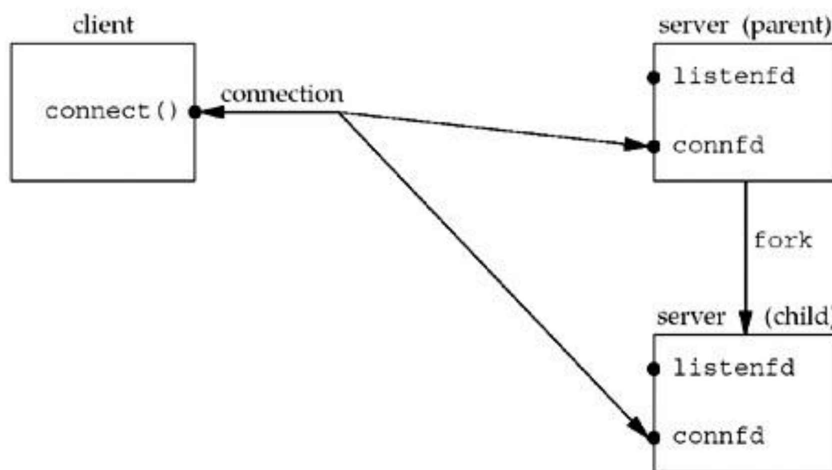
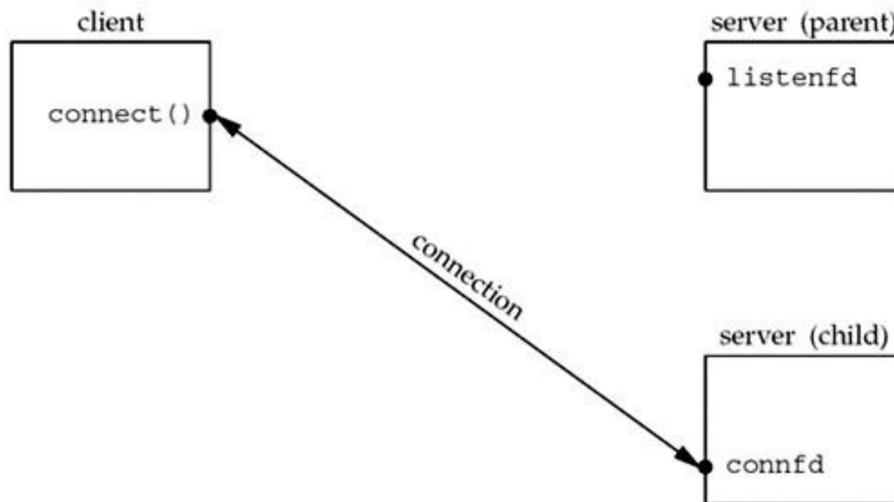


Fig.C

Notice that both descriptors, listenfd and connfd, are shared (duplicated) between the parent and child.

© Prof.Prasad Sawant PCCCS Chichwad
Courtesy Unix Network Programming, Volume 1: The Sockets Networking API W. Richard Stevens

UNIT 3 ELEMENTARY TCP SOCKETS



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.

Questions

1. Write a note on socket Function
2. Write a note on connect Function
3. Write a note on bind Function
4. Write a note on listen Function
5. Write a note on accept Function
6. What is purpose of fork () and exec ()
7. Describe close function
8. Describe getsockname and getpeername Functions
9. When Hard error and soft error occurs
10. What is diff. between complete connection queue and incomplete connection queue.

Prasad Saw

© Prof.Prasad Sawant PCCCS Chichwad
Courtesy Unix Network Programming, Volume 1: The Sockets Networking API W. Richard Stevens

11. Explain Concurrent Server

Prasad Sawant

© Prof.Prasad Sawant PCCCS Chichwad
Courtesy Unix Network Programming, Volume 1: The Sockets Networking API W. Richard Stevens