## 10.Illustrate the significance of socket functions for elementary TCP client/server with a neat block diagram.

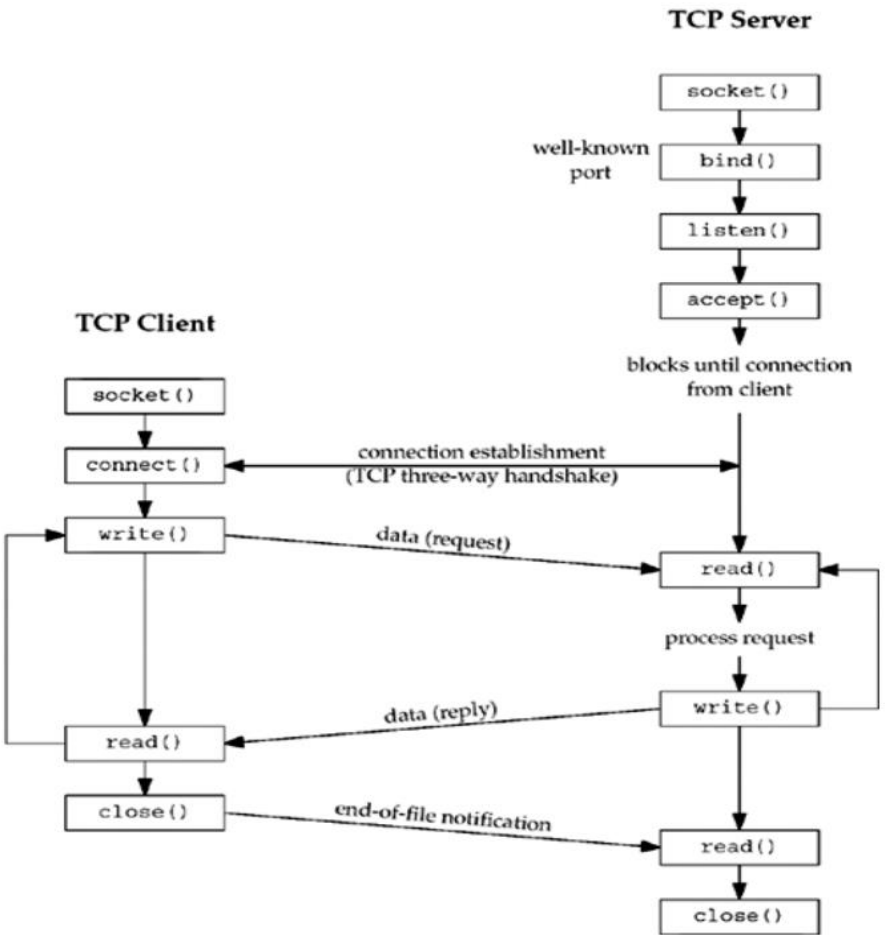


Figure 4.1 shows a timeline of the typical scenario that takes place between a TCP client and server. First, the server is started, then sometime later, a client is started that connects to the server. We assume that the client sends a request to the server, the server processes the request, and the server sends a reply back 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 connection and either terminates or waits for a new client connection.

## 11. Explain the following arguments of the socket function:

a. Family
b. Type
c. Protocol

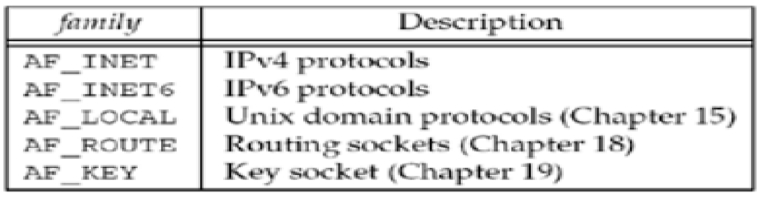
## 4.2 `socket` Function

To perform network I/O, the first thing a process must do is call the `socket` function, specifying the type of communication protocol desired (TCP using IPv4, UDP using IPv6, Unix domain stream protocol, etc.).

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

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

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

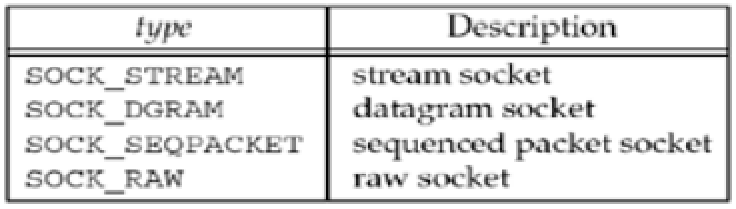
**family** specifies the protocol family,

**Protocol *family* constants for socket function.**



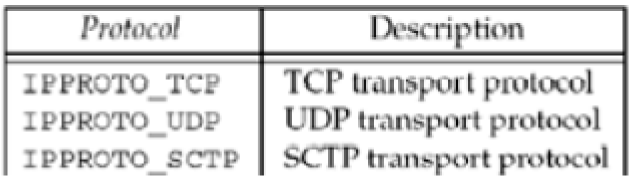
| *family* | Description |
|---|---|
| AF_INET | IPv4 protocols |
| AF_INET6 | IPv6 protocols |
| AF_LOCAL | Unix domain protocols (Chapter 15) |
| AF_ROUTE | Routing sockets (Chapter 18) |
| AF_KEY | Key socket (Chapter 19) |

The socket **type** is one of the constants.



| *type* | Description |
|---|---|
| SOCK_STREAM | stream socket |
| SOCK_DGRAM | datagram socket |
| SOCK_SEQPACKET | sequenced packet socket |
| SOCK_RAW | raw socket |

***protocol* of sockets for AF_INET or AF_INET6.**



| *Protocol* | Description |
|---|---|
| IPPROTO_TCP | TCP transport protocol |
| IPPROTO_UDP | UDP transport protocol |
| IPPROTO_SCTP | SCTP transport protocol |

## 12. Explain the following functions of TCP socket:

a. connect
b. bind
c. listen
d. accept
e. close

## 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);

                                                    Returns: 0 if OK, -1 on error
```

*sockfd* is a socket descriptor returned by the socket function.

The second and third arguments are a pointer to a socket address structure and its size.

## 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
```

## 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.

## 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).

## 4.9 `close` Function

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

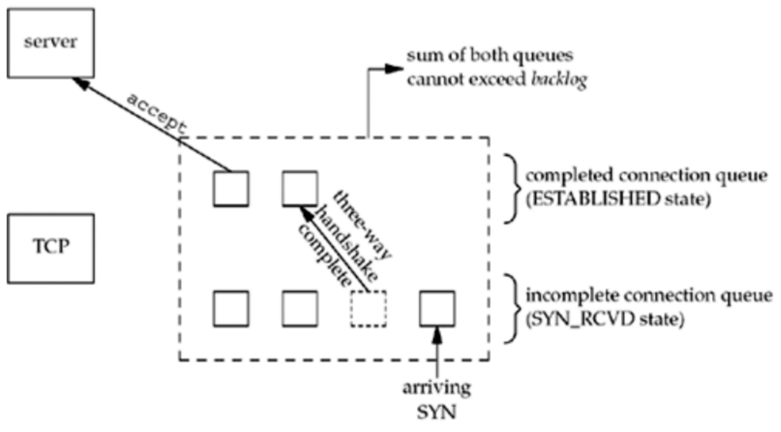
| |
|---|
| `#include <unistd.h>` |
| `int close (int` *sockfd*`);` |
| Returns: 0 if OK, -1 on error |

**13. With a neat block diagram explain the queues maintained by TCP for a listening socket. Also show the packets exchanged during the connection establishment with these two queues.**

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).

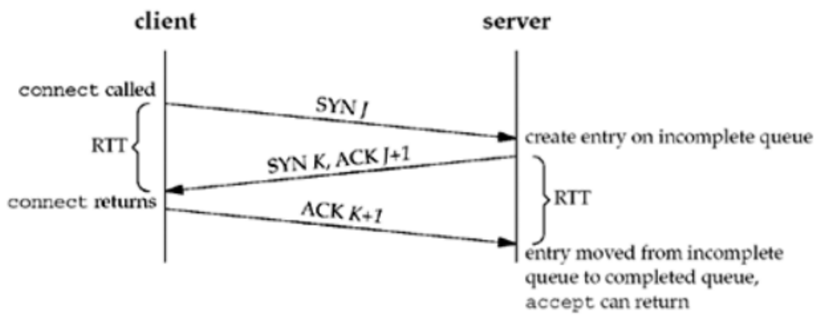
**Figure 4.7. The two queues maintained by TCP for a listening socket.**


server
accept
sum of both queues cannot exceed *backlog*
completed connection queue (ESTABLISHED state)
three-way handshake complete
TCP
incomplete connection queue (SYN_RCVD state)
arriving SYN

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.

**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.**


client
server
connect called
RTT
connect returns
SYN J
SYN K, ACK J+1
ACK K+1
create entry on incomplete queue
RTT
entry moved from incomplete queue to completed queue, accept can return

- 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
- 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.
- 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.

## 14.Illustrate the significance of fork and exec functions.

### fork and exec Functions

| |
|---|
| `#include <unistd.h>` |
| `pid_t fork(void);` |
| Returns: 0 in child, process ID of child in parent, -1 on error |

- It returns once in the calling process (called the parent) with a return value that is the 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.
- The reason **fork** returns 0 in the child, instead of the 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**.

There are two typical uses of fork:

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 typical for network servers. We will see many examples of this later in the text.
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 (described next) to replace itself with the new program. This is typical for programs such as shells.

## exec function:

- The only way in which an executable program file on disk can be 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.

```
#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
```

## 15.Outline the typical concurrent server with the help of pseudocode.

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

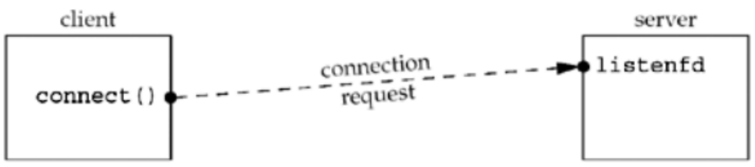
**Figure 4.13 Outline for typical concurrent server.**

```
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 */
}
```

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.

## 16.Demonstrate the status of client/ server before and after call to *accept* returns with a neat block diagram.

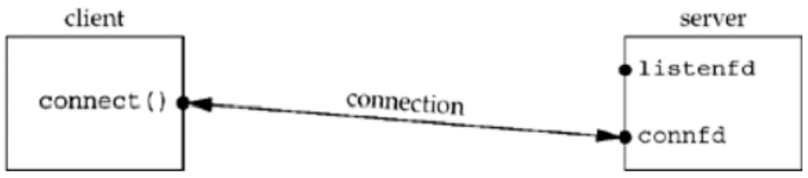
**Figure 4.14. Status of client/server before call to accept returns.**



Above figure 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.

Immediately after accept returns, we have the scenario shown in Figure 4.15. 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.

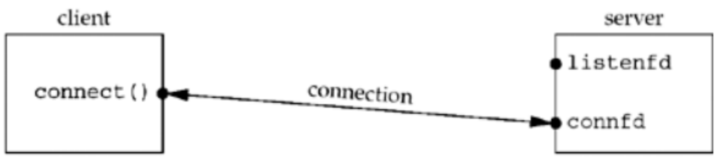
**Figure 4.15. Status of client/server after return from accept.**



## 17.Demonstrate the status of client/ server after fork returns with a neat block diagram.

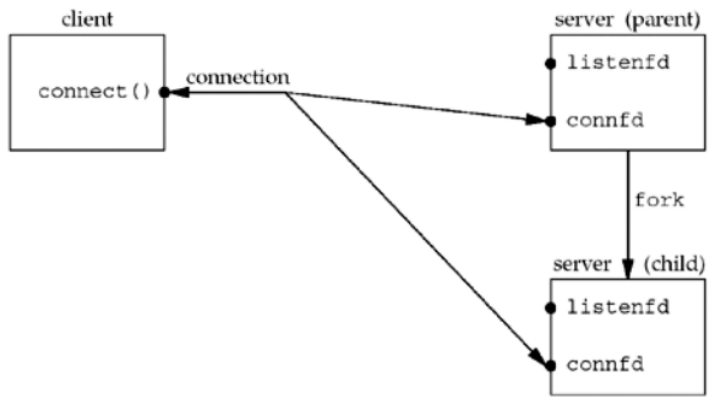
Immediately after accept returns, we have the scenario shown in Figure 4.15. The connection is accepted by the kernel and a new socket, connfd, is created. This is a connected socket and data can now be read and written across the connection.

**Figure 4.15. Status of client/server after return from accept.**



The next step in the concurrent server is to call `fork`. Figure 4.16 shows the status after `fork` returns.

**Figure 4.16. Status of client/server after `fork` returns.**

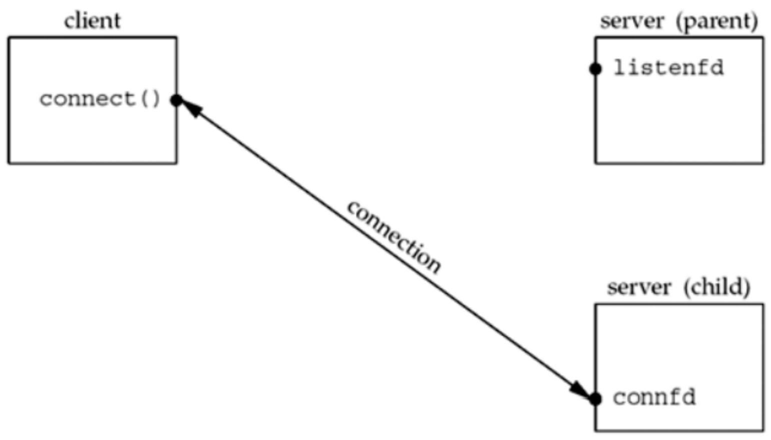


Notice that both descriptors, `listenfd` and `connfd`, are shared (duplicated) between the parent and child.

## 18.Demonstrate the status of client/ server after parent and child close appropriate sockets with a neat block diagram.

The next step is for the parent to close the connected socket and the child to close the listening socket. This is shown in Figure 4.17.

**Figure 4.17. Status of client/server after parent and child close appropriate 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.

## 19.Comment on the significance of *getsockname* and *getpeername* functions.

## 4.10 `getsockname` and `getpeername` Functions

These two functions return either the local protocol address associated with a socket (`getsockname`) or the foreign protocol address associated with a socket (`getpeername`).

```
#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
```

Notice that the final argument for both functions is a value-result argument. That is, both functions fill in the socket address structure pointed to by *localaddr* or *peeraddr*.

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 port number), `getsockname` returns the local port number that was assigned.
- `getsockname` can be called to obtain the address family of a socket, as we show in
- When a server is `exec`ed by the process that calls `accept`, the only way the server can obtain the identity of the client is to call `getpeername`. This is what happens

## 20.Develop the pseudocode that returns the address family of a socket.

The `sockfd_to_family` function shown in Figure 4.19 returns the address family of a socket.

### Figure 4.19 Return the address family of a socket.

*lib/sockfd_to_family.c*

```
#include     "unp.h"
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);
}
```