LAB EXERCISE - 6

Write a Program in C/ C++ for client server communication using TCP or IP sockets to make client send the name of the file and server to send back the contents of the requested file if present.

Objective: To Use TCP/IP sockets and write a client server program in which the client sends the file name in request message and the server sends back the contents of the requested file if present.

Sockets is a method for communication between a client program and a server program in a network. A socket is defined as "the endpoint in a connection." Sockets are created and used with a set of programming requests or "function calls" sometimes called the sockets application programming interface (API). The most common sockets API is the Berkeley UNIX C interface for sockets. Sockets can also be used for communication between processes within the same computer.

Creating a Socket

The socket system call is given below:

```
s = socket(domain, type, protocol);
```

The domain is either AF_UNIX or AF_INET. An AF_UNIX socket can only be used for interprocess communications on a single system, while an AF_INET socket can be used for communications between systems.

The type specifies the characteristics of communication on the socket. SOCK_STREAM creates a socket that will reliably deliver bytes in-order, but does not respect messages boundaries; SOCK_DGRAM creates a socket that does respect message boundaries, but does not guarantee to deliver data reliably, uniquely or in order. A SOCK_STREAM socket corresponds to TCP; a SOCK_DGRAM socket corresponds to UDP

The protocol selects a protocol. Ordinarily this is 0, allowing the call to select a protocol. This is almost always the right thing to do, though in some special cases you may want to select the protocol yourself. Remember that this refers to the underlying network protocol: such well-known protocols as http, ftp, and ssh are all built on top of tcp, so tcp is the right choice for any of those protocols.

For example:

```
s = socket(AF_INET, SOCK_STREAM, 0);
```

will create a socket that will use TCP to communicate. The return value (s) is a file descriptor for the socket.

Binding a Socket

At this point created a socket, but name is not given to it. To give a name to the socket by using the bind system call:

```
bind(s, name, namelen);
```

This call gives the socket a name. For an Internet socket, the name is a struct defined as

```
struct sockaddr_in {
   sa_family_t    sin_family; /* address family: AF_INET */
   u_int16_t      sin_port;   /* port in network byte order */
   struct in_addr  sin_addr;  /* internet address */
};

/* Internet address. */
struct in_addr {
   u_int32_t      s_addr;     /* address in network byte order */
};
```

sin_family is always AF_INET; sin_port is the port number, and sin_addr is the IP address. Port numbers below 1024 are reserved - that means only processes with an effective user id of 0 (ie the root) can bind to those ports.

Listening to the Socket

Once the socket has been created and bound, the daemon needs to indicate that it is ready to listen to it. It does this with the listen system call, as in

```
listen(s, 5);
```

The main thing this does is to set a limit on how many would-be clients can be queued up trying to connect to the socket (the limit in this example is 5). If the limit is exceeded the clients don't actually get refused, instead their connection requests get dumped on the floor. Eventually they will end up retrying.

Accepting Connections

The server is able to accept connections by calling accept:

```
newsock = accept(s, (struct sockaddr *) &from, &fromlen);
```

For this call, s is, as you'd expect the socket that was returned by the socket call. The accept() call blocks until the client connects to the socket.

Connecting to the daemon

A client connects to the socket using the connect call. First it creates a socket using the socket call, then it connects it to the daemon's socket using connect:

```
connect(s, (struct sockaddr *)&server, sizeof(server));
```

This call returns *a new socket*. This means the daemon can communicate with the client using the newly created (and unnamed) socket, while continuing to listen on the old one.

At this point, the server normally forks a child process to handle the client, and goes back to its accept loop.

The child doing the communication can either use standard read and write calls, or it can use send and recv. These calls work like read and write, except that you can also pass flags allowing for some options.

Sending Data

There are a variety of functions that may be used to send outgoing messages.

write() may be used in exactly the same way as it is used to write to files. This call may only be used with SOCK_STREAM type sockets.

#include<sys/types.h>

#include<sys/socket.h>

int write(int fd, char *msg, int len);

fd is the socket descriptor. **msg** specifies the buffer holding the text of the message, and **len** specifies the length of the message.

write() is similar to write() except that it writes from a set of buffers. This is called a gather write. This call may only be used with SOCK_STREAM type sockets.

send() may be used in the same way as write(). The prototype is

#include<sys/types.h>

#include<sys/socket.h>

int send(int fd, char *msg, int len, int flags);

fd is the socket descriptor. **msg** specifies the buffer holding the text of the message, and **len** specifies the length of the message. **flags** may be formed by ORing MSG_OOB and MSG_DONTROUTE. The latter is only useful for debugging. This call may only be used with SOCK_STREAM type sockets.

Receiving Data

There are a variety of functions that may be used to receive incoming messages.

read() may be used in the exactly the same way as for reading from files. There are some complications with non-blocking reads. This call may only be used with SOCK_STREAM type sockets.

#include <sys/types.h>

#include <sys/socket.h>

int read(int fd, char *buff, int len)

fd is the socket descriptor. **buff** is the address of a buffer area. **len** is the size of the buffer.

readv() may be used in the same way as read() to read into several separate buffers. This is called a scatter read. This call may only be used with SOCK_STREAM type sockets.

recv() may be used in the same way as read(). The prototype is

#include <sys/types.h>

#include <sys/socket.h>

int recv(int s, char *buff, int len, int flags)

buff is the address of a buffer area. **len** is the size of the buffer. **flags** is formed by ORing MSG_OOB and MSG_PEEK allowing receipt of out of band data and allowing peeking at the incoming data. This call may only be used with SOCK_STREAM type sockets.

Steps to execute the exercise

- Client side
    1) sfd = Create a socket with the socket(….) system call
    2) Connect the socket to the address of the server using the connect(sfd, …) system call. The IP address of the server machine and port number of the server service need to be provided.
    3) Read file name from standard input by n = read(stdio, buffer, sizeof(buffer))

4) Write file name to the socket using write (sfd, buffer, n)
5) Read file contents from the socket by m = read(sfd, buffer1, sizeof(buffer1))
6) Display file contents to standard output by write(stdio, buffer1, m)
7) Go to step 5 if m>0
8) Close socket by close (sfd)

- Server side
  1) sfd = Create a socket with the socket(…) system call
  2) Bind the socket to an address using the bind (sfd, …) system call. If not sure of machine IP address, keep the structure member s_addr to INADDR_ANY. Assign a port number between 3000 and 5000 to sin_port.
  3) Listen for connections with the listen (sfd, …) system call
  4) sfd = Accept a connection with the accept (sfd, …) system call. This call typically blocks until a client connects with the server.
  5) Read the filename from the socket by n = read(sfd, buffer, sizeof(buffer))
  6) Open the file by fd = open(buffer)
  7) Read the contents of the file by m = read(fd, buffer1, sizeof(buffer1))
  8) Write the file content to socket by write(sfd, buffer1, m)
  9) Go to step 7 if m>0
  10) Close(sfd)

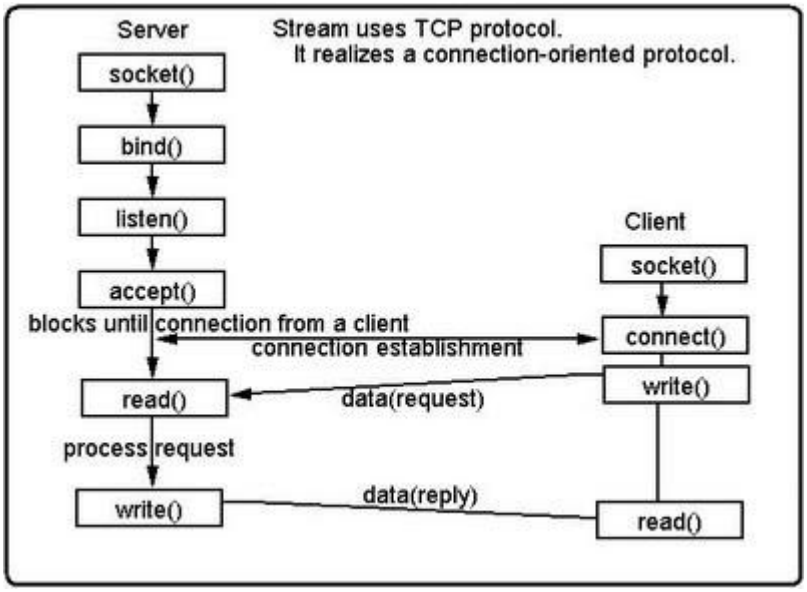

Figure 2: TCP sockets