CS6111-COMPUTER NETWORKS LAB

Ex.NO.1 :SOCKET PROGRAMMING

DATE: 2.08.2024

BATCH:N,P and Q

Introduction to Sockets :

A socket is a logical communication End-point. It is a software interface that provides a means of communicating between two networking devices or between two processes running on the same device (Unix Sockets).

The most popular analogy i have come across for a socket and quote the same is to "**look at a socket as a door of a house with the house being the process (from Computer Networking – a top down approach)**". The socket forms the interface between the application layer and the Transport Layer.

The below diagram from the book Computer Networking – a top down approach provides a good depiction of a socket. The below example is for communication between two devices over the internet.

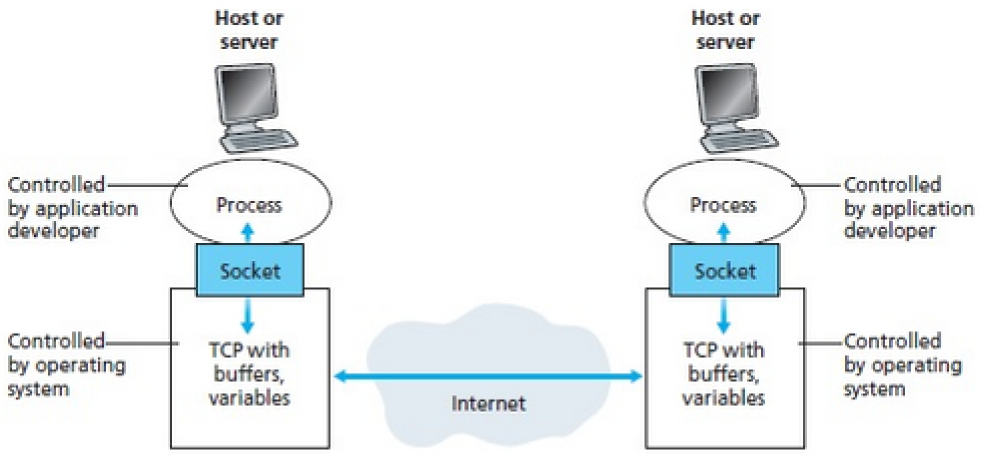

Host or server
Host or server
Controlled by application developer
Process
Socket
Controlled by operating system
TCP with buffers, variables
Internet
Process
Socket
TCP with buffers, variables
Controlled by application developer
Controlled by operating system

Figure 2.3 ♦ Application processes, sockets, and underlying transport protocol

The developer of the process (The application in this case) can choose the transport layer protocol (TCP/UDP/UNIX domain) that he wishes to communicate to. In case of communication between two devices on the internet, the socket API allows the developer choice of the particular IP address to connect to and the port on which the service connection is desired (connecting to a web-server in this case).

For e.g – different websites have their own IP addresses and the port that might be opened to communicate with those websites would be either port 80 – http port or port 443 – https port.

We will look at different socket communication methods and try and write programs that will allow a reader to get comfortable with sockets.

TCP: connection-oriented, makes sure data is delivered, sends packets again and again until everything is received, sends acknowledgments that data is received

UDP: connectionless, does not guarantee delivery

In the IP world:
IP address is a 32-bit address that we usually write in 4 numbers separated by dots: 18.0.2.231

Host name is a name that maps to an IP address that makes it easier to remember an address. Hostname **web** is the name of a web host in the *mit.edu* domain: web.mit.edu is the full name that maps to IP address: 18.69.0.38.

IP addresses
A network interface is identified by an IP address. IP version 4 addresses are 32-bit numbers written in four 8-bit parts. For example (as of this writing):

- 18.9.22.69 is the IP address of a MIT web server.
- 173.194.193.99 is the address of a Google web server.
- 104.47.42.36 is the address of a Microsoft Outlook email handler.
- 127.0.0.1 is the loopback or localhost address: it always refers to the local machine. Technically, any address whose first octet is 127 is a loopback address, but 127.0.0.1 is standard.

Hostnames

- Hostnames are names that can be translated into IP addresses. A single hostname can map to different IP addresses at different times; and multiple hostnames can map to the same IP address. For example:
- annauniv.edu is the name for anna university's web server. You can translate this name to an IP address yourself using dig, host, or nslookup on the command line, e.g.:

```
$ dig +short annauniv.edu
```

- google.com is exactly what you think it is. Try using one of the commands above to find google.com's IP address. What do you see?
- localhost is a name for 127.0.0.1. When you want to talk to a server running on your own machine, talk to localhost.

Port :

A port is an integer number that accompanies a networked application that lets it communicate independently of the other networked applications. For example, if you have a web server, it usually sits at port 80 for incoming requests. This server will not interfere with, say an FTP server on the same machine that is listening to port 21.

In this pattern there are two kinds of processes: clients and servers. A client initiates the communication by connecting to a server. The client sends requests to the server, and the server sends replies back. Finally, the client disconnects. A server might handle connections from many clients concurrently, and clients might also connect to multiple servers.

Many Internet applications work this way: web browsers are clients for web servers, an email program like Outlook is a client for a mail server, etc.

On the Internet, client and server processes are often running on different machines, connected only by the network, but it doesn't have to be that way — the server can be a process running on the same machine as the client.

Port numbers

A single machine might have multiple server applications that clients wish to connect to, so we need a way to direct traffic on the same network interface to different processes.

Network interfaces have multiple ports identified by a 16-bit number. Port 0 is reserved, so port numbers effectively run from 1 to 65535.

A server process binds to a particular port — it is now **listening** on that port. A port can have only one listener at a time, so if some other server process tries to listen to the same port, it will fail.

Clients have to know which port number the server is listening on. There are some well-known ports that are reserved for system-level processes and provide standard ports for certain services. For example:

- Port 22 is the standard SSH port. When you connect to athena.dialup.mit.edu using SSH, the software automatically uses port 22.
- Port 25 is the standard email server port.
- Port 80 is the standard web server port. When you connect to the URL http://web.mit.edu in your web browser, it connects to 18.9.22.69 on port 80.

When the port is not a standard port, it is specified as part of the address. For example, the URL http://128.2.39.10:9000 refers to port 9000 on the machine at 128.2.39.10.

Network sockets

A **socket** represents one end of the connection between client and server.

- A **listening socket** is used by a server process to wait for connections from remote clients.

In Java, use ServerSocket to make a listening socket, and use its accept method to listen to it. Remember that a port can only have one listener, so this will fail if another thread or process is currently listening at the same port.

- A **connected socket** can send and receive messages to and from the process on the other end of the connection.

In Java, clients use a Socket constructor to establish a socket connection to a server. Servers obtain a connected socket as a Socket object returned from ServerSocket.accept.

The word “socket” arises by analogy to a hole for a physical plug, like a USB cable. A listening socket is similar to a USB socket waiting for a cable to be plugged into it. A connected socket is like a USB socket with a cable plugged into it, and like a cable with two ends, the connection involves *two* sockets – one at the client end, and one at the server end.

Be careful with this physical-socket analogy, though, because it’s not quite right. In the real world, a USB socket allows only one cable to be connected to it, so a “listening” USB socket becomes a “connected” USB socket when a cable is physically plugged into it, and stops being available for new connections. But that’s not how network sockets work. Instead, when a new client arrives at a listening socket, a fresh connected socket is created on the server to manage the new connection. The listening socket continues to exist, bound to the same port number and ready to accept another arriving client when the server calls accept on it again. This allows a server to have multiple active client connections that originally targeted the same port number.

Buffers

The data that clients and servers exchange over the network is sent in chunks. These are rarely just byte-sized chunks, although they might be. The sending side (the client sending a request or the server sending a response) typically writes a large chunk (maybe a whole string like “HELLO, WORLD!” or maybe 20 megabytes of video data). The network chops that chunk up into packets, and each packet is routed separately over the network. At the other end, the receiver reassembles the packets together into a stream of bytes.

The result is a bursty kind of data transmission — the data may already be there when you want to read them, or you may have to wait for them to arrive and be reassembled.

When data arrive, they go into a **buffer**, an array in memory that holds the data until you read it.

Byte streams

The data going into or coming out of a socket is a **stream** of bytes.

In Java, InputStream objects represent sources of data flowing into your program. For example:

- Reading from a file on disk with a FileInputStream
- User input from System.in
- Input from a network socket

OutputStream objects represent data sinks, places we can write data to. For example:

- FileOutputStream for saving to files
- System.out for normal output to the user

- System.err for error output
- Output to a network socket

Character streams

The stream of bytes provided by InputStream and OutputStream is often too low-level to be useful. We may need to interpret the stream of bytes as a stream of *Unicode characters*, because Unicode can represent a wide variety of human languages (not to mention emoji). A String is a sequence of Unicode characters, not a sequence of bytes, so if we want to use strings to manipulate the data inside our program, then we need to convert incoming bytes into Unicode, and convert Unicode back to bytes when we write it out.

In Java, Reader and Writer represent incoming and outgoing streams of Unicode characters. For example:

- FileReader and FileWriter treat a file as a sequence of characters rather than bytes
- the wrappers InputStreamReader and OutputStreamWriter adapt a byte stream into a character stream

One of the pitfalls of I/O is making sure that your program is using the right *character encoding*, which means the way a sequence of bytes represents a sequence of characters. The most common character encoding for Unicode characters is UTF-8. For network communication, UTF-8 is the right choice. Usually, when you create a Reader or Writer, Java will default to UTF-8 encoding. But problems occur when other programs on your computer use a different character encoding to read and write files, which means your Java program can't interoperate with them. To compensate for this file-compatibility problem, Java on your platform may instead default to using a different character encoding, picking it up from a system setting – and then messing up Java code that does network communication, for which the better default is UTF-8. For example, Microsoft Windows has a non-standard encoding called CP-1252, and for Java programs running on Windows, this may be the default.

Character-encoding bugs can be hard to detect. UTF-8, CP-1252, and most other character encodings happen to be supersets of one of the oldest standardized character encoding, ASCII. ASCII is big enough to represent English, so English text tends to be unaffected by character-encoding bugs. But the bug is lying in wait for accented Latin characters, or scripts other than the Latin alphabet, or emoji, or even just ‘fancy’ “curved” quotes. When there is a character encoding disagreement, these characters turn into garbage.

To avoid character-encoding problems, make sure to explicitly specify the character encoding whenever you construct a Reader or Writer object. The example code in this reading always specifies UTF-8.

Blocking

Input/output streams exhibit blocking behavior. For example, for socket streams:

- When an incoming socket's buffer is empty, calling read blocks until data are available.
- When the destination socket's buffer is full, calling write blocks until space is available.

Blocking is very convenient from a programmer's point of view, because the programmer can write code as if the read (or write) call will always work, no matter what the timing of data arrival. If data (or for write, space) is already available in the buffer, the call might return very quickly. But if the read or write can't succeed, the call **blocks**. The operating system takes care of the details of delaying that thread until read or write *can* succeed.

Sockets

Basically, we need to send and receive data between two different hosts. The process of receiving and sending that data is Input/Output or I/O. The process of sending and receiving data over a network is called network I/O. Sockets constitute an interface to enable network I/O. File I/O based on the UNIX open-read-write-close is neither flexible nor complete enough to allow good network I/O

Problems with that paradigm include:

1. There is no ability to create server code that waits for connections passively or client code that forms connections actively
2. By sending datagrams in a connectionless scheme, programs might want to specify a destination address along with each datagram, instead of binding destinations at the time when *open()* is called
3. We also need to be able to specify the quality of a connection and the underlying protocol used (datagram, stream, etc.)

No standard way devised for this I/O but a de facto interface: sockets.

Sockets independent of network technology and therefore work with several network protocol families:

TCP/IP (PF_INET),
AppleTalk (PF_APPLETALK),
Unix file system (PF_UNIX)

They allow different types of data to be transmitted:

SOCK_STREAM (e.g. Web access),
SOCK_DGRAM (e.g. Real Audio and Real Video),
SOCK_RAW (bit access)

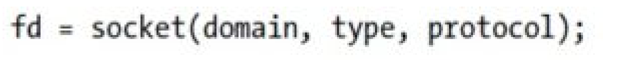
Can specify the protocol. For example, {PF_INET, SOCK_STREAM} defines TCP and {PF_INET, SOCK_DGRAM} defines UDP.

Sockets are used in most networked applications.

The Different Socket Communication APIs that need to be understood for setting up a socket communication link are

- **Socket API**

1. The Socket API allows the creation of a socket and creates a file descriptor which is returned to the application requesting the same. The Socket API is provided below



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

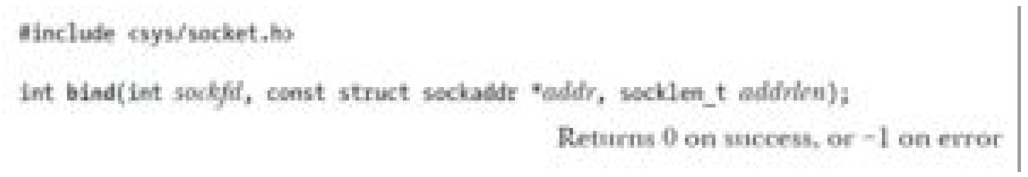
domain – The domain parameter indicates the communication domain (for e.g. AF_INET -IPv4, AF_INET6 -IPv6, AF_UNIX for UNIX domain sockets etc)

Type – the type parameter indicates whether the socket being opened is a byte stream (Stream sockets (SOCK_STREAM)) or message based (SOCK_DGRAM) or Raw socket stream (SOCK_RAW)

protocol field – The protocol field was included under the belief that a single protocol family will support multiple families. however, it has not occurred that multiple address families have been defined for a particular protocol family. Usually this parameter is set to 0 (zero). In special cases such as RAW sockets or netlink sockets – it might be set to a particular value which we will see later. (NOTE: AF stands for Address Family and PF stands for Protocol Family – both are synonymous in with one another in most cases)

- **Bind API**

The Bind API binds the socket obtained via socket API to an address or path (AF_UNIX). The API is shown below



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

int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
                                        Returns 0 on success, or –1 on error
```

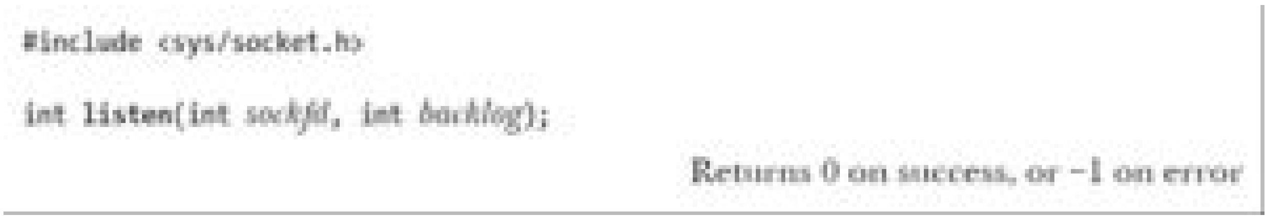
sockfd – The sockfd is the socket file descriptor obtained via socket API call

struct sockaddr *addr – The addr structure pointer indicates an address to which the socket needs to be bound. (Note : the socket structures for different address families are different- but the bind API takes in a generic sockaddr structure. The different address family socket structures are mapped to sockaddr during bind. We shall see this in the next article)

socklen_t addrlen – the addrlen parameter indicates the actual length of the structure element passed in the address parameter (addr). This parameter allows different structures to be mapped to sockaddr structure as is done in the second parameter of Bind API

- **listen API**

The Listen API is used in Stream socket conections and is used to mark the socket as Passive. (Refer Active and Passive sockets here). The Listen API also has a backlog parameter which indicates the number of connections that can be pending for connection to a server. The Listen API is provided below

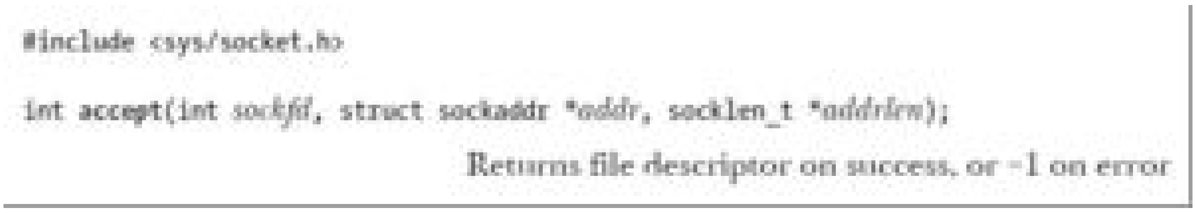


```
#include <sys/socket.h>

int listen(int sockfd, int backlog);
                                    Returns 0 on success, or -1 on error
```

- **<u>accept API</u>**

The accept API accepts an incoming connection (Stream sockets). The accept call is a blocking call and the invocation of accept API will block if there are no pending connection requests. The accept API is provided below



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

int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
                          Returns file descriptor on success, or -1 on error
```

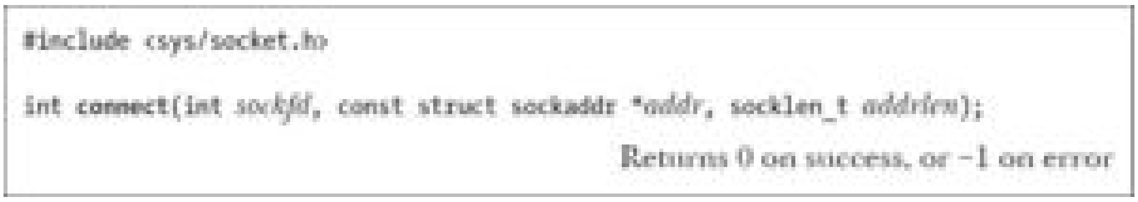
<u>sockfd</u> – the socket File descriptor

<u>struct sockaddr *addr</u> – the sockaddr structure pointer. This returns the address of the peer socket

<u>socklen_t addrlen</u> – the length of the structure member passed in parameter 2 of the API

- **<u>connect API</u>**

The connect API is used to connect to a particular listening server socket. The server address and address length are passed as a parameter to the connect API.



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

int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
                                    Returns 0 on success, or -1 on error
```

<u>sockfd</u> – The socket file descriptor from the socket API call on the client side

<u>struct sockaddr *addr</u> – the socket address which is being connected to

<u>socklen_t addrlen</u> – length of the structure member passed in parameter 2 of the API

- **<u>remove and unlink (in case of Unix Domain Sockets)</u>**

The Remove and unlink APIs are used to remove any residual previous socket connection. This is used particularly in the case of a UNIX domain socket wherein the previous file link for the socket might still be existing. The Remove and unlink APIs are provided below

int remove(const char ***pathname*);**

remove calls unlink API for files and rmdir API for directories

int unlink(const char ***pathname*);**

unlink deletes a name from the file system

Lab 1 Execution:

Implement Socket Programming to build a client-server architecture using TCP.

A server in a networked application is a program that sits on one host and listens passively for incoming traffic at a certain port.

The web server sits web.mit.edu listens to port 80 or in other words (18.69.0.38:80).

The client sits at the other end and is active. It send connection requests, writes data, receives data and closes the connection.

When a server receives an incoming connect request, it answers by an accept, reads incoming data and writes outgoing data. It also closes the socket once the communication is finished.

Client Side	**Server Side**
create: socket()	create: socket()
	bind()
	listen()
connect()	accept()
read()/write()	read()/write()
close()	close()

Note: all example code in provided in C, but you can also use any other language that provides a socket library.

1. create one or more sockets on demand:

```
int socket (int domain, int type, int protocol)
```

e.g.

```
client_socket = socket (PF_INET, SOCK_STREAM, 0);

server_socket = socket (PF_INET, SOCK_STREAM, 0);
```

client_socket and server_socket will be set to a negative number if no socket was created, positive or null otherwise

2. for a client, connect to a (remote) server:

int connect (int socket, struct sockaddr* destaddr, int addrlen)

e.g.

connect (client_socket, &server_address, sizeof (server_address));

server_address is a structure that contains information about the server you are building, what port it listens to, what clients can connect to the server and what protocol is used:

bzero(&server_address, sizeof(server_address)); // reset it

server_address.sin_family = AF_INET; // internet protocol family

server_address.sin_addr.s_addr = htonl(INADDR_ANY); // all addresses welcome

server_address.sin_port = htons(PORT); // PORT is an integer e.g. 5501

In the case of a server application, 3 steps:

bind (socket, localaddr, addrlen)

// establishes a local address for the socket

listen (socket, backlog)

// puts a socket in a passive mode, ready to accept a number (backlog) of connections at a time

accept (socket, address, addrlen)

// waits then accepts connection requests

// address refers to the address of the client that has placed a request

3. read() and write() function in pair: client can send

int write (int socket, char* buffer, int length)

e.g. write (server_socket, buffer, 1);

in this case server should have:

int read (int socket, char* buffer, int length)

e.g. read (client_socket, buffer, 1);

or vice versa

4. close the socket

close (socket)

e.g.

close (client_socket)

Important and Helpful functionality

How to look up a host by its name:

gethostbyname(host)

How to look up a host by its address:

gethostbyaddr(address, length, type)

where length = sizeof(address) and type = AF_INET

In order to use connectionless service, such as UDP, one can use *recvfrom()* and *sendto()* instead of *read()* and *write()*.
recvfrom() and *writeto()* are usually associated with connectionless traffic though because both source address is always sent with the data.

recvfrom(socket, buffer, length, flags, address, &addrlen);

where you will use flags as 0 and address of the sending host

sendto(socket, buffer, length, flags, address, addrlen);

where you will use flags as 0 and address of the sending host

To find about the functionality of a certain command (function), type at the athena prompt, the *man* command followed by the function name:

athena% man gethostbyname
will give help information on *gethostbyname()*

Client and Server files

TCP:

TCP is a connection oriented protocol. The socket connection will follow the TCP 3- way handshake to establish the connection.

In This Article, we take a look at a simple server and client example. The Internet IPv4 domain will be considered for the below example.

The Stream socket will follow the below connection establishment process shown in the diagram.

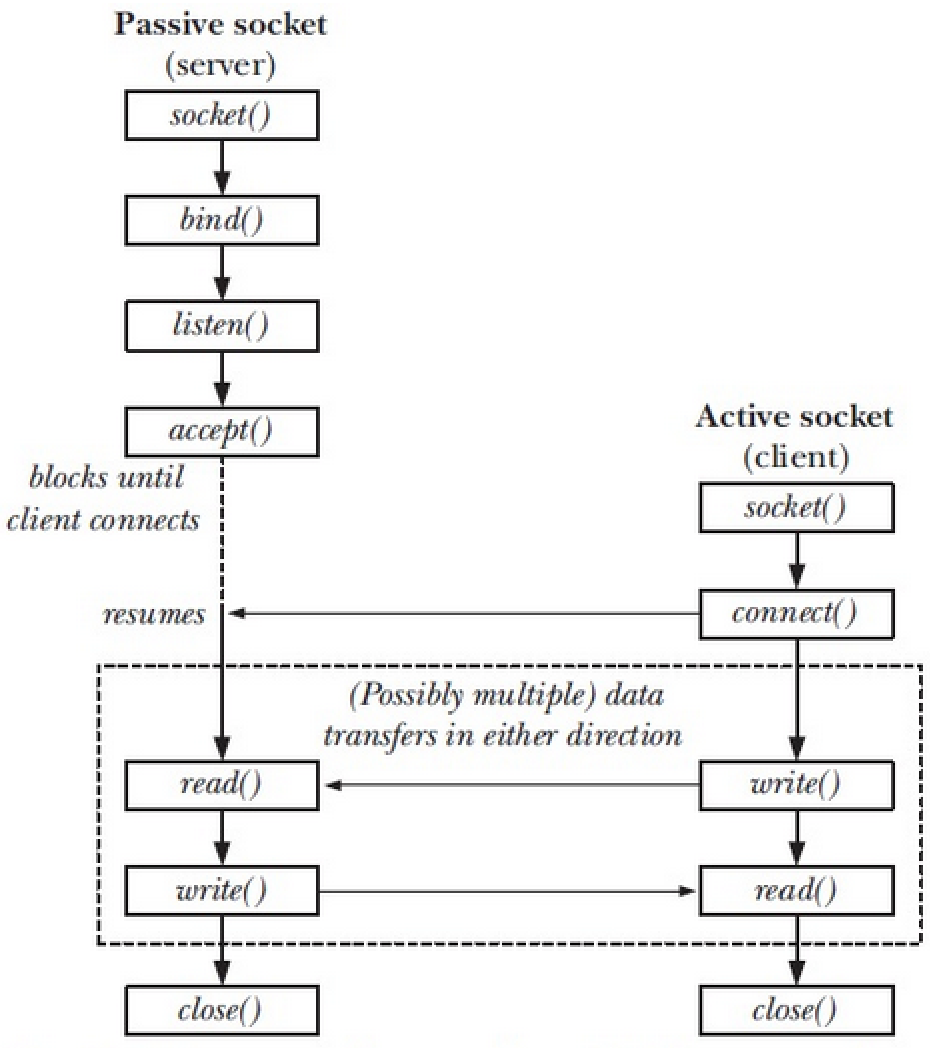


Figure 56-1: Overview of system calls used with stream sockets

```
#include <stdio.h>
#include <sys/socket.h>
#include <stdlib.h>
#include <string.h>
/* Internet socket structures
* are defined in below header */
#include <netinet/in.h>
/* inet_addr is defined in
* below header */
#include <arpa/inet.h>
#include <signal.h>

/* for Read/write API calls*/
#include <unistd.h>

#define PORT 55000
#define TRUE 1
```

```
/* IPv4 socket structures for
* server and client */

struct sockaddr_in *serveraddr = NULL, *clientaddr = NULL;

/* socket file descriptor */
int socket_fd, clientfd;

/* Interrupt_handler – so that CTRL + C can be used to
* exit the program */
void interrupt_handler (int signum) {
        close(socket_fd);
        close(clientfd);

        free(serveraddr);
        free(clientaddr);
        printf("socket connection closed\n");
        exit(0);
}

int main () {
         /* length of client structure received on accept */
         int length;
         char buffer[50];
         char *string = "Hello from Server";

        /* signal handler to exit out of while 1 loop */
        signal (SIGINT, interrupt_handler);
        signal (SIGPIPE, interrupt_handler);

         /* Part 1 – create the socket */

         if((socket_fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
                 printf("unable to create the socket\n");
                 exit(0);
         }

         /* Part 2 – create the server connection – fill the structure*/
         serveraddr = (struct sockaddr_in *)malloc(sizeof(struct sockaddr_in));

         if (serveraddr == NULL) {
                 printf("Unable to allocate memory\n");
                 goto end;
         }

         serveraddr->sin_family = AF_INET;

         /* use loopback address (127.0.0.1)so that
          * the server/client connection
          * can be created on the same machine
          * inet_addr API – Change the IP address from
          * dotted decimal presentation form to an integer form
          * suitable as an internet address */

          serveraddr->sin_addr.s_addr = inet_addr("127.0.0.1");
          serveraddr->sin_port = htons(PORT);

          /* Part 3 – bind and start accepting connections */

          if(0 != bind(socket_fd, (struct sockaddr *)serveraddr, sizeof(struct
sockaddr_in))) {
                printf("unable to bind to the socket\n");
                goto end1;
          }

          /* accept upto 5 pending requests
           * and make the server socket a passive socket
           */
          if (listen(socket_fd, 5) == -1) {
                 printf("Listen Failed\n");
                goto end1;
        }

        clientaddr = (struct sockaddr_in *)malloc(sizeof(struct sockaddr_in));
        if (clientaddr == NULL) {
                 printf("unable to allocate memory\n");
                 goto end1;
        }

       /* accept is a blocking call and the call would stop further invocation of          *  code till a
client connection is accepted */
        printf("waiting for connection\n");
        clientfd = accept(socket_fd, (struct sockaddr *)clientaddr, &length);

        if (clientfd < 0) {
                 printf("Unable to connect to Client FD\n");
                 goto end2;
        }

        /* read and write data from/to client socket */
        while (TRUE) {
             /* Use the client socket FD to read data from the client */
             memset(buffer, 0, sizeof(buffer));
             read(clientfd, buffer, sizeof(buffer));
             printf("%s\n", buffer);

            /* write data to the client */
            memset(buffer, 0, sizeof(buffer));
            snprintf(buffer, (strlen(string) + 1), "%s\n", string);
            write(clientfd, buffer, sizeof(buffer));
        }
end3:
        if (clientfd >= 0)
                close(clientfd);
end2:
        free(clientaddr);
end1:
        free(serveraddr);
end:
        close(socket_fd);
}
```

Client Code:

```
#include <stdio.h>
#include <sys/socket.h>
/* Internet socket structures
* are defined in below header */
#include <netinet/in.h>
/* inet_addr is defined in
* below header */
#include <arpa/inet.h>
#include <string.h>
#include <stdlib.h>
#include <signal.h>

/* For read/write APIs*/
#include <unistd.h>
#define PORT 55000

/* socket file descriptor */
int socket_fd;
/* IPv4 socket structure */
struct sockaddr_in *sockaddrin = NULL;

/* Interrupt_handler so that CTRL +C can be used
* to exit the program */
void interrupt_handler (int signum) {
        close(socket_fd);
        free(sockaddrin);
        printf("socket connection closed\n");
        exit(0);
}

int main () {
        char buffer[50] = {0};
        char *string = “Hello from client”;

         /* signal handler to exit out of while 1 loop */
         signal (SIGINT, interrupt_handler);
         signal (SIGPIPE, interrupt_handler);

         /* Part 1 – create a socket */
         /* create the socket FD */
         socket_fd = socket(AF_INET, SOCK_STREAM, 0);

         if (socket_fd < 0) {
                 printf(“The Socket API call failed\n”);
                 exit(0);
         }

          /* Part 2 – allocate server connection details
           * structure for connection */

          /* allocate memory and fill the sockaddr_in structure
           * so that it can be connected to the socket */

          sockaddrin = (struct sockaddr_in *)malloc(sizeof(struct sockaddr_in));
         if (sockaddrin == NULL) {
                  /* unable to allocate memory */
                  goto end;
          }
        sockaddrin->sin_family = AF_INET;
        sockaddrin->sin_port = htons(PORT);

         /* use loopback address (127.0.0.1)so that
          * the server/client connection
          * can be created on the same machine
          * inet_addr API – Change the IP address from
          * dotted decimal presentation form to an integer form
          * suitable as an internet address */
         sockaddrin->sin_addr.s_addr = inet_addr(“127.0.0.1”);

         /* part 3 – connect to the server defined by IP address and port*/
         /* connect to the server */
        if (0 != connect(socket_fd, (struct sockaddr *)sockaddrin,
                (socklen_t)(sizeof(struct sockaddr_in)))) {
                printf(“Unable to connect\n”);
                /* unable to connect to server */
                goto end1;
        }

        /* Part 4 – send/receive data */
        /* read and write data from/to client socket */
         while (1) {

                memset(buffer, 0, sizeof(buffer));
                /* write data to the server */
                snprintf(buffer, (strlen(string) + 1), “%s\n”, string);
                write(socket_fd, buffer, sizeof(buffer));

                memset(buffer, 0, sizeof(buffer));
                read(socket_fd, buffer, sizeof(buffer));
                printf(“%s\n”, buffer);
        }
end1:
                free(sockaddrin);
end:
                close(socket_fd);
}
```

Analysis:

The initial code to create a socket on the server or the client is provided below

```
        /* Part 1 – create a socket */
        /* create the socket FD */
        socket_fd = socket(AF_INET, SOCK_STREAM, 0);

        if (socket_fd < 0) {
                printf(“The Socket API call failed\n”);
                exit(0);
        }
```

If the above call to **“socket API”** succeeds, then a socket File Descriptor is provided to the application invoking the socket call. The socket is currently not bound to any address. The “Bind API” will bind the socket to an address at the server side. The code for bind is shown below

```
         serveraddr->sin_addr.s_addr = inet_addr(“127.0.0.1”);
         serveraddr->sin_port = htons(PORT);

         /* Part 3 – bind and start accepting connections */

         if(0 != bind(socket_fd, (struct sockaddr *)serveraddr, sizeof(struct
sockaddr_in))) {
                      printf(“unable to bind to the socket\n”);
```

```
            goto end1;
        }
        /* accept up-to 5 pending requests
         * and make the server socket a passive socket to accept clients
         */
        if (listen(socket_fd, 5) == -1) {
                printf("Listen Failed\n");
               goto end1;
       }
```

When Bind API succeeds, the socket File descriptor will be bound to a Port and IP address. The socket is placed in listen state to accept client connections. The output of the **"ss shell command"** after the successful invocation of listen shows the current state of the bound socket to be in listen state. The socket state is marked in Red. It can be seen that the socket is in listen state and bound to 127.0.0.1:55000 and does not have a peer connection yet

```
State        Recv-Q Send-Q Local Address:Port                Peer Address:Port
LISTEN        0       128        127.0.0.53%lo:domain             0.0.0.0:*
LISTEN        0       5            127.0.0.1:ipp                          0.0.0.0:*
LISTEN        0       5            127.0.0.1:55000                      0.0.0.0:*
TIME-WAIT 0       0         192.168.0.6%enp0s3:51034       35.224.99.156:http
LISTEN        0       5            [::1]:ipp                              [::]:*
```

After entering into listen state, the server code will await for incoming connections on an accept API call. The code for the same is provided below

```
    /* accept is a blocking call and the call would stop further invocation of              *  code till a
client connection is accepted */
       printf("waiting for connection\n");
       clientfd = accept(socket_fd, (struct sockaddr *)clientaddr, &length);

       if (clientfd < 0) {
              printf("Unable to connect to Client FD\n");
              goto end2;
       }
```

clientfd is a socket file descriptor and is the client connection socket. Note that, accept is a blocking call and the server will wait for incoming connections at the instance of accept API invocation.

On the client side, after a socket is created, the client tries to connect to the IPaddress and port combination of the server. Notice that the client does not bind in this example case. The Kernel will provide a port number for the client socket. The code snippet achieving that operation for the client is provided below

```
       sockaddrin->sin_family = AF_INET;
       sockaddrin->sin_port = htons(PORT);

       sockaddrin->sin_addr.s_addr = inet_addr("127.0.0.1");
```

```
   /* part 3 – connect to the server defined by IP address and port*/
   /* connect to the server */
   if (0 != connect(socket_fd, (struct sockaddr *)sockaddrin,
        (socklen_t)(sizeof(struct sockaddr_in)))) {
        printf("Unable to connect\n");
        /* unable to connect to server */
       goto end1;
   }
```

if connect attempt to the server succeeds, then the socket connection is established and the socket can be used to transmit and receive data on the socket. The port number for the client is allocated by the linux kernel.

The connect API on a TCP stream socket invokes a TCP 3-way handshake. The output of the **"ss shell command"** after connect and the 3-way handshake is also provided below for the reader's understanding and reference

```
State      Recv-Q   Send-Q Local Address:Port        Peer Address:Port
LISTEN      0        128     127.0.0.53%lo:domain      0.0.0.0:*
LISTEN      0         5       127.0.0.1:ipp              0.0.0.0:*
LISTEN      0         5       127.0.0.1:55000            0.0.0.0:*
ESTAB       50        0       127.0.0.1:57798            127.0.0.1:55000
ESTAB       0         50      127.0.0.1:55000            127.0.0.1:57798
LISTEN      0         5       [ ::1]:ipp [::]:*
```

Simplified Client Code

```
#include<stdio.h>
#include<sys/types.h>
#include<netinet/in.h>
#include<sys/socket.h>
#include<string.h>
#include<unistd.h>
int main()
{
        int sockdesc;
        struct sockaddr_in servaddr;
        sockdesc=socket(AF_INET,SOCK_STREAM,0);
        if(sockdesc==-1)
        {
                printf("Socket not created");
                return -1;
        }

        servaddr.sin_family=AF_INET;
        servaddr.sin_port=htons(1025);
        servaddr.sin_addr.s_addr=htonl(INADDR_ANY);

        if (connect(sockdesc,(struct sockaddr*)&servaddr,sizeof(servaddr)) < 0)
        {
                printf("Connect Failed");
                return -1;
        }

        char buffer[10];
        fgets(buffer,sizeof(buffer),stdin);
        write(sockdesc,buffer,sizeof(buffer));

        read(sockdesc,buffer,10);
        printf("Message from server: %s", buffer);


        close(sockdesc);
        return 0;

}
```

Simplified Server

```
#include<stdio.h>
#include<sys/types.h>
#include<netinet/in.h>
#include<sys/socket.h>
#include<string.h>
#include<unistd.h>
int main()
{
        int sockdesc;
        struct sockaddr_in servaddr,cliaddr;

        sockdesc=socket(AF_INET,SOCK_STREAM,0);
        if(sockdesc==-1)
        {
                printf("Socket not created");
                return -1;
        }

        servaddr.sin_family=AF_INET;
        servaddr.sin_port=htons(1025);
        // PORT number ranges from 1024 to 49151
        servaddr.sin_addr.s_addr=htonl(INADDR_ANY);
        if(bind(sockdesc,(struct sockaddr *)&servaddr,sizeof(servaddr)) < 0)
```

```
	{
		printf("Bind Failed");
		return -1;
	}

	if(listen(sockdesc,5)<0)
	{
		printf("Listen Failed");
		return -1;
	}


	while(1)
	{
		int len=sizeof(cliaddr);
		int connfd=accept(sockdesc,(struct sockaddr*)&cliaddr,&len);
		if (connfd<0)
		{
			printf("Accept failed");
			return -1;
		}


		char buffer[10];
		strcpy(buffer," ");
		read(connfd,buffer,10);
		printf("Message received from client: %s", buffer);

		write(connfd,buffer,sizeof(buffer));

	}

	close(sockdesc);



	return 0;
}
```

SPOT QUESTION :
Lab 1 – Socket Programming – TCP
Exercise 1

Design a client-server application using connection oriented protocol (TCP) with the server performing bit stuffing by inserting a bit 0 after every consecutive five 1s. For example, if the data sent to the server is 01011111100111111001010, after bit stuffing, the server sends back 0101111101001111100001010.
The communication goes as follows:
The client sends a message consisting of binary data to the server.
The server
receives the message,
performs bit stuffing on the data, and
sends the resultant bit stuffed message back to the client.

The messages sent and received should be displayed on both the client and receiver.