Government College of Engineering, Jalgaon

(An Autonomous Institute of Government of Maharashtra)

Name : Dewang Mankar **Semester :** V **PRN :** 2241034

Class : T. Y. B.Tech Computer **Academic Year :** 2024-25 **Subject : CNTL**

Course Teacher : Mrs. Prajakta Cheke

Date of Performance : **Date of Completion :**

Practical No:- 1A

Writeup	Correctness of program.	Documentation of program	Viva	Attendence For practical	Timely completion	Total	Dated sign of course teacher
4	2	2	5	2	5	20	

Aim : UNIX Sockets: WAP program in C/C++ /Python/Java sockets API.

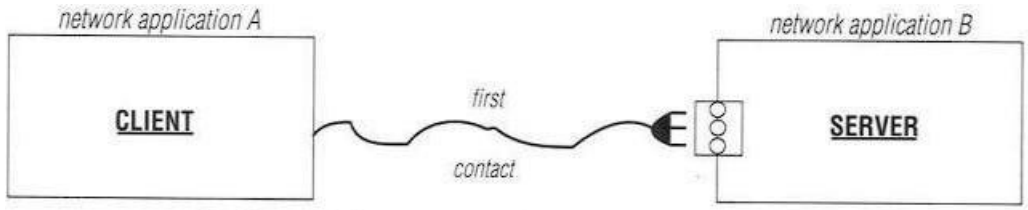
a. TCP sockets

b. UDP sockets

Server accepts operation and floating point numbers from the clients; performs arithmetic operations and sends the result back to the client. Server applications must handle at least five clients simultaneously. Both the server and client should display input and output numbers as well as the operation. The server and client processes should be run on different machines. During evaluation,students will demonstrate via creating multiple client processes on different machines.

Theory:-

The Client / Server Model It is possible for two network applications to begin simultaneously, but it is impractical to require it. Therefore, it makes sense to design communicating network applications to perform complementary network operations in sequence, rather than simultaneously. The server executes first and waits toreceive; the client executes second and sends the first network packet to the server. After initial contact, either the client or the server is capable of sending and receiving data.



A client initiates communications to a server.

What are Unix Sockets?

Imagine a two-way tunnel, not of dirt but of data, built within the digital terrain of your Linux system. That's essentially what a Unix socket represents. It's a software endpoint facilitating bidirectional communication between processes, regardless of their location within the system or even beyond its borders. Unix sockets offer two distinct flavors:

1. **Network Sockets**

These are the long-distance runners, enabling communication across networks using protocols like TCP/IP. Imagine sending data packets across the internet, each bearing the socket's address as its destination.

2. **Domain Sockets**

These local champions facilitate communication between processes within the same system. Think of them as private pipes connecting programs within the Linux kingdom.

Socket Types and Their Roles

The type of socket dictates its communication style:

1. Stream Sockets (SOCK_STREAM)

Think of them as flowing rivers, ensuring reliable, sequential data transmission. Imagine sending a document, with each byte arriving in the correct order, much like the pages of a book.

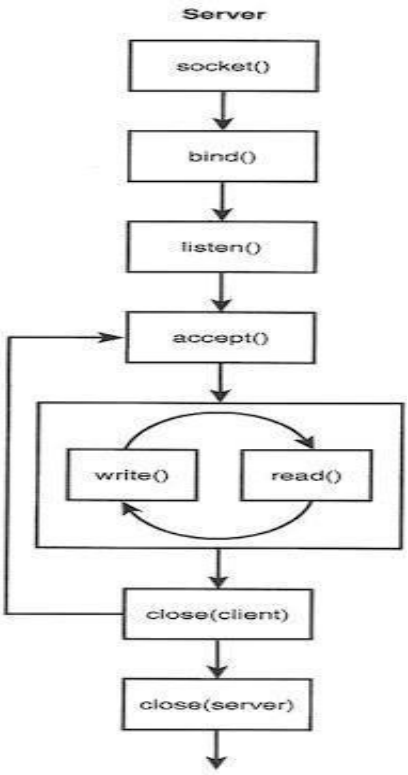
2. Datagram Sockets (SOCK_DGRAM)

Picture them as mail delivery, where individual packets carry the data. While faster, they lack guaranteed delivery and order, like postcards mailed independently.

Creating UNIX Sockets

Sockets are a means to allow communication between two different processes over the same or different machines/networks. To be more precise, it's a way to talk to other computers and exchange data. In Unix, every I/O action is done by writing or reading a file descriptor. Sockets are the end point of communication with each process having its socket and a socket descriptor for all the operations. In this article, we are going to read more about creating a socket in a Unix system and implementing both client and server-side programs.

Creating TCP server:-



Step 1:Creating a socket:

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

Creating a socket is in some ways similar to opening a file. This function creates a file descriptor and returns it from the function call. You later use this file descriptor for reading, writing and using with other socket functions Parameters: family: AF_INET or PF_INET (These are the IP4 family) type: SOCK_STREAM (for TCP) or SOCK_DGRAM (for UDP) protocol: IPPROTO_TCP (for TCP) or IPPROTO_UDP (for UDP) or use 0

Step 2:Binding an address and port number

int bind(int socket_file_descriptor, const struct sockaddr * LocalAddress, socklen_t AddressLength);

We need to associate an IP address and port number to our application. A client that wants to connect to our server needs both of these details in order to connect to our server. Notice the difference between this function and the connect() function of the client. The connect function specifies a remote address that the client wants to connect to, while here, the server is specifying to the bind function a local IP address of one of its Network Interfaces and a local port number. 15 The parameter socket_file_descriptor is the socket file descriptor returned by a call to socket() function. The return value of bind() is 0 for success and –1 for failure. Again make sure that you cast the structure as a generic address structure in this function. You also do not need to find information about the IP addresses associated with the host you are working on. You can specify: INNADDR_ANY to the addressstructure and the bind function will use on of the available (there may be more than one) IPaddresses. This ensures that connections to a specified port will be directed to this socket, regardless of which Internet address they are sent to. This is useful if host has multiple IP addresses, then it enables the user to specify which IP address will be b_nded to which portnumber.

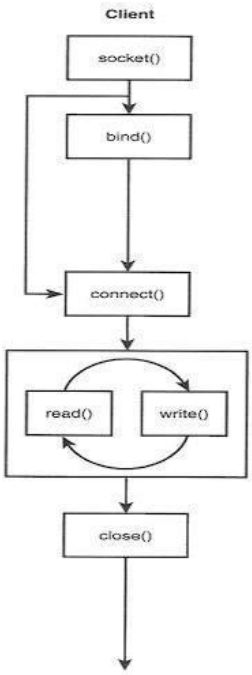
Step 3:Listen for incoming connections Binding is like waiting by a specific phone in your house, and Listening is waiting for it to ring. int listen(int socket_file_descriptor,int backlog);

The backlog parameter can be read in Stevens' book. It is important in determining how many connections the server will connect with. Typical values for backlog are 5 – 10. The parameter socket_file_descriptor is the socket file descriptor returned by a call to socket() function. The return value of listen() is 0 for success and –1 for failure.

Step 4:Accepting a connection. int accept (int socket_file_descriptor, struct sockaddr * ClientAddress, socklen_t *addrlen);

accept() returns a new socket file descriptor for the purpose of reading and writing to the client. The original file descriptor is used usually used for listening for new incoming connections. Servers will be discussed in much more detail in a later lab. It dequeues the next connection request on the queue for this socket of the server. If queue is empty, this function blocks until a connection request arrives. (read the reference book TCP/IP Implementation in C for more details.)

Creating a TCP Client



Step 1:

Create a socket : Same as in the server.

Step 2:

Binding a socket: This is unnecessary for a client, what bind does is (and will be discussed indetail in the server section) is associate a port number to the application. If you skip this step with a TCP client, a temporary port number is automatically assigned, so it is just better to skip this step with the client.

Step 3:

Connecting to a Server: int connect(int socket_file_descriptor, const struct sockaddr *ServerAddress, socklen_t AddressLength); Once you have created a socket and have filled in the address structure of the server you want to connect to, the next thing to do is to connect to that server. This is done with the connect function listed above. 17 **This is one of the socket functions which requires an address structure so remember to type cast it to the generic socket structure when passing it to the second argument ** Connect performs the three-way handshake with the server and returns when the connection is established or an error occurs.

Once the connection is established you can begin reading and writing to the socket.

Step 4:

Writing to a socket:

int write(int file_descriptor, const void * buf, size_t message_length);

The return value is the number of bytes written, and –1 for failure. The number of bytes written may be less than the message_length. What this function does is transfer the data from you application to a buffer in the kernel on your machine, it does not directly transmit the data over the network. This is extremely important to understand otherwise you will end up with many headaches trying to debug your programs.

Reading from a socket:

int read(int file_descriptor, char *buffer, size_t buffer_length);

The value returned is the number of bytes read which may not be buffer_length! It returns –

1 for failure. As with write(), read() only transfers data from a buffer in the kernel to your application , you are not directly reading the byte stream from the remote host, but rather TCP is in control and buffers the data for your application.

Step 5:

After you are finished reading and writing to your socket you most call the close system call on the socket file descriptor just as you do on a normal file descriptor otherwise you waste system resources.

The close function : int close(int filedescriptor);

Conclusion :-

In conclusion, Sockets are one of the best ways to make strong and effective communication. IPC (inter-process communication) inside the same host has made strong Unix socket programming in C. It helps to build reliable and high-performance communication channels for processes operating on the same computer by using Unix domain sockets. The client initiates a TCP socket, configures the server address, and connects for message reception.

Name & Sign of Course Teacher
Mrs.Prajakta D Sawle