INTRODUCTION

SOCKET PROGRAMMING

A socket is a communications connection point (end point) that you can name and address in a network. The process that use a socket can reside on the same system or on different systems on different networks. Sockets are useful for both standalone and network applications. A socket allow you to exchange information between processes on the same machine or across a network, distributes work to most efficient machine and allows access to centralized data easily.

The connection that a socket provides can be connection-oriented or connectionless. a dialogue between the programs will follow. A program that provides the service (the server program establishes the available socket which is enabled to accept incoming connection requests. Optionally, the server can assign a name to the service that it supplies which allows clients to identify where to obtain and how to connect that service. The client of the service (the client program) must request the service of the server program. The client does this

by making connection to the distinct name or to the attributes associated with the distinct name that the server program has designated. It is similar to dialing a telephone number and making a connection with another party that is offering a service. When the reciever of the call (the server, in this example, a plumber) answeres the telephone, a connection is established. The plumber verifies that you have reached the correct party, and the connection remains active as long as both parties requires it. Connectionless communication implies that no connection is established over which a dialog or data transfer can take place. Instead, the server program designates a name that identifies where to reach it (much like a post office box).

SOCKET TYPE

Stream(SOCK_STREAM);

This type of socket is connection-oriented. Established an end-to-end connection by using the Bind(), listen() and accept(), and connect() functions. SOCK_STREAM sends data without errors or duplication, and receives the data in the sending order. SOCK_STREAM builds flows control to avoid

data overruns. It does not impose record boundaries on that data. SOCK_STREAM considers the data to be a stream of bytes

Datagram (SOCK_DGRAM):

In Internet Protocol terminology, the basic unit of data transfer is a datagram. The datagram socket is connectionless. It establishes no end-to-end connection with the transport provider (protocol). The socket sends datagrams as independent packets with no guarantee of delivery. Datagrams can arrive out of order. For some transport providers, each datagram can use a different route through the network.

Creating a connection-oriented socket:

A connection-oriented server uses the following sequence of function calls socket(), bind(), connect(), listen() accept(), send(), recv() close().

A connection-client uses the following sequence of function calls socket(), gethostbyname(), connect(), send(), recv(), close().

Creating a connectionless socket:

A connectionless client illustrates the socket API's that are written for User Datagram.

Protocol (UDP)

The server uses the following sequence of functions calls

socket(), bind(), sendto(), recvfrom(), close()

The client example uses the following sequence of function calls.

socket(), gethostbyname(), sendto(), recvfrom(), close().

Functions and their parameters:

- socket()

This function gives a socket descriptor that can be used in later system calls.

syntax: int socket (int domain, int type, int protocol)

domain → AF_INET or AF_UNIX

type → the type of socket needed (Stream or Datagram)

SOCK_STREAM for stream socket

SOCK_DGRAM for Datagram Socket

protocol → 0

- bind()

This function associates a socket with a port

syntax: int bind(int fd, struct sockaddr* my_addr, int addrlen)

fd → socket descriptor

my_addr → ptr to structure sockaddr

addrlen → size of (struct sockaddr)

- connect()

This function is used to connect to an IP address on a defined port.

syntax: int connect(int fd, struct sockaddr* serv_addr, int addrlen)

fd → socket file descriptor

addrlen → size of (struct sockaddr)

- listen()

This function is used to wait for incoming connection. Before calling listen(), bind() is to be called. After calling listen(), accept() is to be called inorder to accept incoming connection.

syntax: int listen(int fd, int backlog)

fd → socket file descriptor

backlog → number of allowed connections.

- accept()

This function is used to accept an incoming

connections

syntax: int accept (int fd, void * addr, int * addrlen)
fd → socket file descriptor
addr → ptr to struct sockaddr

□ send ():
This function is used to send data over stream sockets. It returns the number of bytes sent out.

syntax: int send (int fd, const void * msg, int len, int flags)

fd → socket descriptor
msg → ptr to the data to be send.
flags → 0

□ recv ()
This function receives the data send over the stream sockets. It returns the number of bytes read into the buffer.

syntax: int recv (int fd, void * buf, int len, unsigned int flags)

fd → socket file descriptor
buf → buffer to read into the buffer

len → maximum length of the buffer
flag → set to 0

□ send to ()

This function serves the same purpose as send() function except that it is used for datagram sockets. It also returns the number of bytes send out.

syntax: int sendto (int fd, void *, int len, unsigned int flags, const struct sockaddr *to, int tolen)

fd → socket file descriptor
msg → ptr to the data to be send
len → length of the data to be send
flags → 0
to → ptr to a struct sockaddr
tolen → size of (struct sockaddr)

□ recv from ()

This function serves the same purpose as recv() function except that it is used for datagram sockets. It also returns the number of bytes recieved

syntax: int recv from (int fd, const void * buf, int len, unsigned int flags, const struct sockaddr * from, int fromlen)

fd → socket file descriptor
buf → buffer to read information into
len → maximum length of the buffer
flags → set to 0
from → ptr to struct sockaddr
fromlen → size of (ptr to an int that should be intialized to struct sockaddr)

1. close()

This function is used to close that socket on your socket descriptor.

syntax: close(fd);

fd :⇒ socket file descriptor

Client-Server Communication Using TCP

Problem Description

Develop a program to implement interprocess communication using stream sockets with the help of interfaces provided by the standard library.

Data Structures and Functions:
Data Strutures used here are:-

struct sockaddr : This structure holds socket address information for many types of sockets.

```
struct sockaddr
{
  unsigned short sa_family; // address family AF_xxx
  char sa_data[14]; // 14 bytes of protocol address.
};
```

sa_family can be a variety of things. But it will be AF_INET for everything we do in this document. sa_data contains a destination address and port number for the socket. To deal with struct sockaddr, programmers created a parallel structure.

struct sockaddr_in ("in" for "internet")

```
struct sockaddr_in
{
  short int sin_family; // Address family
  unsigned short int sin_port; // Port number
  struct in_addr sin_addr; // Internet address
  unsigned char sin_zero[8]; // Same size as struct
                             sockaddr
};
```

This structure makes it easy to reference elements of the socket address. Note that sin_zero (which is included to pad the structure to the length of a

struct sockaddr) should be set to all zeroes with the function memset. Finally, the sin_port and sin_addr must be in Network Byte Order.

Functions used here are:

System calls that allow to access the network functionality of a Unix box are as given below. When you call one of these functions, the kernel takes over and does all the work for you automatically.

Server side:
socket(), bind(), connect(), listen(), accept(), send(), recv(), close().

Client side
socket(), gethostbyname(), connect(), send(), recv(), close()

Shown 2-7-18