**TCP protocol**

• TCP (Transmission Control Protocol) is a standard that defines how to establish and maintain a network conversation through which application programs can exchange data

• TCP works with the Internet Protocol (IP), which defines how computers send packets of data to each other

• together, TCP and IP are the basic rules defining the Internet

. all major Internet applications such as the World Wide Web, email, remote administration, and file transfer rely on TCP

• TCP is a connection-oriented protocol

• means a connection is established and maintained until the application programs at each end have finished exchanging messages

• TCP creates a connection between the source and destination node before transmitting the data and keeps the connection alive until the communication is no longer active

**Features of a TCP connection**

Connection Oriented

• Reliability

Handles lost packets

• Handles packet sequencing

• Handles duplicated packets

Full Duplex

• Flow Control

Congestion Control

**Client/Server model**

the Client-Server model is a relationship in which one program (the client) requests a service or resource from another program (the server)

two processes or two applications that communicate with each other to exchange some information one of the two processes acts as a client process, and another process acts as a server

• there can be multiple clients that talk to one server

• clients typically communicate with servers by using the TCP/IP protocol suite

computer transactions in which the server fulfils a request made by a client are very common the client-server model has become one of the central ideas of network computing.

• the client establishes a connection to the server over a local area network (LAN) or wide-area network (WAN), such as the Internet

• clients need to know the address of the server, but the server does not need to know the address or even the existence of the client prior to the connection being established

- once the server has fulfilled the client's request, the connection is terminated

• because multiple client programs share the services of the same server program, a special server called a daemon may be activated just to await client requests

as mentioned, the client process makes a request for information and after getting the response, this process may terminate or may do some other processing

an example of a client program would be an Internet Browser • sends a request to the Web Server to get one HTML webpage

the server process is takes a request from one or more clients

after getting a request from the client, this process will perform the required processing, gather the

requested information, and send it to the requestor client

once done it becomes ready to serve another client

server processes are always alert and ready to serve incoming requests

an example of a server process would be a Web Server

keeps waiting for requests from Internet Browsers and as soon as it gets any request from a browser, picks up a requested HTML page and sends it back to that Browser

**Types of Servers**

there are two types of servers you can implement

1. **Iterative Server**

• the simplest form of a server where the server process serves one client at a time

• after completing the first request, it takes request from another client

• other client wait until it is their turn

1. **Concurrent Servers**

• this type of server runs multiple concurrent processes to serve many requests at a time

• one process may take longer and another client does not need to wait too long the simplest way to write a concurrent server under Unix is to fork a child

process to handle each client separately

**Sockets**

sockets are the "virtual" endpoints of any kind of network communications done between two computers

• socket programming is a way of connecting two nodes on a network to communicate with each other

. one socket(node) listens on a particular port at an address another socket reaches out to the other to form a connection

server forms the listener socket while client reaches out to the server

when you type www.google.com in your web browser

it opens a socket and connects to google.com to fetch the page and show it to you

• same with any chat client like gtalk or skype

• all network communication goes through a socket

sockets are supported by Unix, Windows, Mac, and many other operating systems

**Steps in using sockets to communicate**

• create a new socket for network communication

attach a local address to a socket (bind)

announce willingness to accept connections (listen)

• block caller until a connection request arrives (accept)

• actively attempt to establish a connection (connect)

send some data over connection (send)

• receive some data over connection (receive)

• release the connection (close)

**Include files**

the following header files need to be included to work with sockets for network communication include system calls for sockets and socket data)

sys/types.h>

contains definitions of a number of data types used in socket calls

<sys/socket.h>

the main socket header file

Includes a number of definitions of structures needed for sockets (socket creation, accept, listen, bin, send, recv, etc)

<netinet/in.h>

contains constants and structures needed for Internet domain addresses

<netdb.h>

defines the structure hostent

carpaſineth

defines in addr structure

**ports**

• a computer might need to run several server programs at once

• might be sending out web pages, posting email, and running a chat server all at the same time

• to prevent the different conversations from getting confused, each server uses a different port

• a port is just like a channel on a TV

• different ports are used for different network services, just like different channels are used for different content

. a port will be defined as an integer number between 1024 and 65535

• port numbers smaller than 1024 are considered well-known system ports (telnet uses port 23, http uses 80, ftp uses 21) (do not use these)

• the port assignments to network services can be found in the file /etc/services

**Port and Service Functions**

• the following functions are provided to fetch service name from the /etc/services file

• struct servent 'getservbyname(char \*name, char \*proto) • takes a service name and a protocol name and returns the corresponding port number for that service

• struct servent "getservbyport(int port, char \*proto)

• takes a port number and a protocol name and returns the corresponding service name

• the return value for each function is a pointer to a structure

struct servent {

char \*s\_name;

char \*\*s\_aliases;

int s port;

char 's\_proto;

}

**IP Address Functions**

• the below functions are for IP Address functionality

. int inet\_aton (const char \*strptr, struct in\_addr \*addrptr)

• converts the specified string, in the Internet standard dot notation, to a network address, and stores the address in the structure provided

• in\_addr\_t inet\_addr (const char \*strptr)

• converts the specified string, in the Internet standard dot notation, to an integer value suitable for use as an Internet address

• char \*inet\_ntoa (struct in\_addr inaddr)

• converts the specified Internet host address to a string in the Internet standard dot notation

**Structures used for support of the main functions**

• various structures are used in socket programming to hold information about the address and port, and other information

• socket address structures are an integral part of every network program, most socket functions require

a pointer to a socket address structure as an argument • we allocate them, fill them in, and pass pointers to them to various socket functions

\* sometimes we pass a pointer to one of these structures to a socket function and it fills in the contents

. we always pass these structures by reference (i.e., we pass a pointer to the structure, not the structure itself), and we always pass the size of the structure as another argument

• when a socket function fills in a structure, the length is also passed by reference, so that its value can be updated by the function

• always set the structure variables to NULL, otherwise it may get unexpected junk values in your structure

**Structures used for support of the main functions**

• sockaddr is a structure that holds the socket information • this is a generic socket address structure, which will be passed in most of the socket function calls

struct sockaddr {

unsigned short sa\_family;

char sa\_data[14];};

sa\_family can be the following, AF\_INET, AF\_UNIX, AF\_NS, AF IMPLINK

• represents an address family • in most of the Internet-based applications, we use AF\_INET.

sa\_data is a protocol-specific address

• we will use port number IP address, which is represented by sockaddr\_in structure

**Structures used for support of the main functions**

• sockaddr\_in is a structure that helps you to reference to the socket's elements

struct sockaddr\_in {

short int sin\_family:

unsigned short int sin\_port;

struct in\_addr sin\_addr;

unsigned char sin\_zero[8];

};

• in addr is used only in the above structure as a structure field and holds the 32 bit netid/hostid

struct in\_addr {

unsigned long s\_addr;};

• hostent is a structure is used to keep information related to host

struct hostent { char \*h\_name;

char \*\*h\_aliases; int h\_addrtype; int h\_length; char \*\*h\_addr\_list

#define h\_addr h\_addr\_list[0]

};

**Sockets versus File I/O**

working with sockets is very similar to working with files

• the socket() and accept() functions both return handles (file descriptor) • reads and writes to the sockets requires the use of these handles (file descriptors)

. in Linux, sockets and file descriptors also share the same file descriptor table

• sockets have addresses associated with them whereas files do not

you cannot randomly access a socket like you can a file with Iseek()

sockets must be in the correct state to perform input or output

**Socket Functions**

• the most common/important functions are the following:

socket()

connect()

• bind()

listen() and accept()

read(), recv(), recvfrom(), or recvmsg(). write(), send(), sendto(), or sendmsg()

close()

• most of the functions are used by both the client and the server with the exception of

• bind() is used particularly by server programs

• connect() by client programs

• all of the above functions need to include <sys/types.h> and <sys/socket.h> except for read/write/close which are defined in <unistd.h>

**socket function**

• to perform network 1/0, the first thing a process must do is call the socket function • specifying the type of communication protocol desired and

protocol family

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

family specifies the protocol family and is usually the constant AF\_INET for IPv4 protocols and AF\_INETc for IPv6 protocols

type specifies the kind of socket you want, and this is usually set to SOCK\_STREAM for a stream socket or SOCK\_DGRAM for a datagram socket

**socket function**

• protocol is the third and should be set to the specific protocol type you are using

IPPROTO\_TCP - TCP transport protocol

• IPPROTO\_UDP - UDP transport protocol

IPPROTO\_SCTP-SCTP transport protocol

• returns a socket descriptor that you can use in later system calls or -1 on error

• the setsockopt function helps in manipulating options for the socket referred to by the file descriptor sockfd

completely optional, but it helps in reuse of address and port • prevents error such as: "address already in use"

int setsockopt(int sockfd, int level, int optname, const void \*optval, socklen\_t optlen);

**connect function**

the connect function is used by a TCP client to establish a connection with a TCP server

int connect(int sockfd, struct sockaddr \*serv\_addr, int addrlen);

• sockfd is a socket descriptor returned by the socket function

• serv addr is a pointer to struct sockaddr that contains destination IP address and port

• addrlen is set to sizeof(struct sockaddr)

• returns 0 if it successfully connects to the server, otherwise it returns -1 on error

**bind function**

when a server starts up, it needs to tell the operating system which port it is going to use • called binding the port

once a server program has created a socket and named it with bind giving it an IP address and port number, should any program anywhere on the network give that same name to the connect() function, that program will find the server program and they will link up

int bind(int sockfd, struct sockaddr \*my\_addr,int addrlen);

sockfd is a socket descriptor returned by the socket function

• my\_addr is a pointer to struct sockaddr that contains the local IP address and port a value for port number means that the system will choose a random port, and INADDR\_ANY value for IP address means the server's IP address will be assigned automatically

server sin port = 0;

server.sin\_addr.s\_addr = INADDR\_ANY,

• addrien is set to sizeof(struct sockaddr)

returns 0 if it successfully binds to the address, otherwise it returns -1 on error

**listen function**

if your server becomes popular, you will probably get lots of clients connecting to it at once • would you like the clients to wait in a queue for a connection?

the listen() system call tells the operating system how long you want the queue to be

• calling listen() with a queue length of 10 means that up to 10 clients can try to connect to the server at once they will not all be immediately answered, but they will be able to wait

the 11th client will be told the server is too busy

. the listen function is called only by a server

int listen(int sockfd, int backlog);

sockfd is a socket descriptor returned by the socket function

backlog is the max number of allowed connections

returns 0 on success, otherwise it returns -1 on error

**accept function**

. once you have bound a port and set up a listen queue, you then just have to...wait

servers spend most of their lives waiting for clients to contact them

• the accept() system call waits until a client contacts the server, and then it returns a second socket descriptor that you can use to hold a conversation on

int accept (int sockfd, struct sockaddr \*cliaddr, socklen\_t \*aderlen);

• sockfd is a socket descriptor returned by the socket function

cliaddr is a pointer to struct sockaddr that contains client IP address and port

addrlen should be set to sizeof(struct sockaddr)

. returns a non-negative descriptor on success, otherwise it returns -1 on error • all read write operations will be done on this descriptor to communicate with the client

**recv function**

• the recv function is used to receive data over stream sockets or connected datagram sockets • if you want to receive data over unconnected datagram sockets you must use recvfrom()

int recv(int sockfd, void \*buf, int len, unsigned int flags);

• sockfd is a socket descriptor returned by the socket function

• buf is the buffer to read the information into

• len is the maximum length of the buffer

flags is set to 0

• returns the number of bytes read into the buffer, otherwise it will return -1 on error

**recvfrom function**

the recvfrom function is used to receive data from unconnected datagram sockets

int recvfrom(int sockfd, void \*buf, int len, unsigned int flags struct sockaddr \*from, int \*fromlen);

• sockfd is a socket descriptor returned by the socket function

buf is the buffer to read the information inco

len is the maximum length of the buffer

flags is set to 0

• from is a pointer to struct sockaddr for the host where data has to be read fromlen should be set it to sizeof(struct sockaddr)

• returns the number of bytes read into the buffer, otherwise it returns -1 on error

**write function**

• the write function attempts to write nbyte bytes from the buffer pointed by buf to the file associated with the open file descriptor, fildes

int write(int fildes, const void \*buf, int nbyte);

• fildes is a socket descriptor returned by the socket function

• buf is a pointer to the data you want to send

• nbyte is the number of bytes to be written

• If nbyte is 0, write() will return 0 and have no other results if the file is a regular file • otherwise, the results are unspecified

• returns the number of bytes actually written to the file associated with fildes if successful otherwise, -1 is returned

**Send function**

• the send function is used to send data over stream sockets or connected datagram sockets •if you want to send data over unconnected datagram sockets, you must use sendto() function

int send(int sockfd, const void \*msg, int len, int flags);

• sockfd is a socket descriptor returned by the socket function

msg is a pointer to the data you want to send

• len is the length of the data you want to send (in bytes)

• flags should be set to 0

\* returns the number of bytes sent out, otherwise it will return -1 on error

**sendto function**

• the sendto function is used to send data over unconnected datagram sockets

int sendto(int sockfd, const void \*msg, int len, unsigned int flags, const struct sockaddr \*to, int tolen);

• sockfd is a socket descriptor returned by the socket function

• msg is a pointer to the data you want to se id

len is the length of the data you want to send (in bytes) flags should be set to 0

• to is a pointer to struct sockaddr for the host where data has to be sent • tolen should be set it to sizeof(struct sockaddr)

• returns the number of bytes sent, otherwise it returns -1 on error

**read function**

• the read function attempts to read nbyte bytes from the file associated with the buffer, fildes, into the buffer pointed to by buf

int read(int fildes, const void \*buf, int nbyte);

• fildes is a socket descriptor returned by the socket function

• buf is the buffer to read the information into

nbyte is the number of bytes to read

• returns the number of bytes actually written to the file associated with fildes, if successful

• otherwise, -1 is returned

**close function**

the close function is used to close the communication between the client and the server

int close( int sockfd );

sockfd is a socket descriptor returned by the socket function

• returns 0 on success, otherwise it returns -1 on error

**shutdown function**

• the shutdown function is used to gracefully close the communication between the client and the server

gives more control in comparison to the close function

int shutdown(int sockfd, int how);

• sockfd is a socket descriptor returned by the socket function

• How

• 0- indicates that receiving is not allowed • 1- indicates that sending is not allowed

• 2 - indicates that both sending and receiving are not allowed • when how is set to 2, it's the same thing as close()

• returns 0 on success, otherwise it returns -1 on error

**Steps specifically for a server socket**

the steps involved in establishing a socket on the server side are as follows

\* create a socket with the socket() system call

•bind the socket to an address using the bind() system call

• for a server socket on the Internet, an ddress consists of a port number on the ha machine

listen for connections with the listen() system call

\* accept a connection with the accept() system call

typically blocks the connection until a client connects with the server

\* send and receive data using the read()/recv and write()/send system calls

**Illustration**

Server Client

Socket Socket

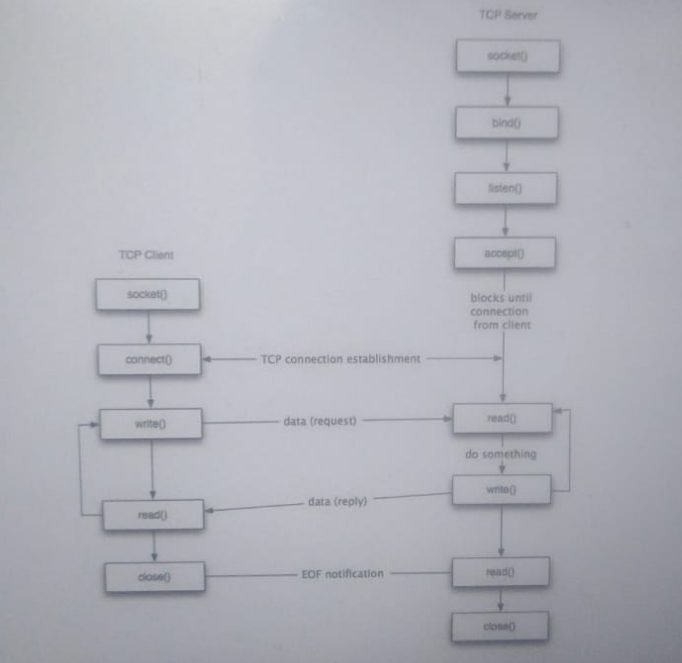
Setsockopt

Bind

Listen Connect

Accept

Send/ Recv Send/ Recv



**simple server steps (pseudocode)**

my sd = socket()

bind( my\_sd, <local address, mainly a port number>)

listen( my side )

start loop

his\_sd = accept( my sd, <empty address to be filled in with his incoming info>)

recv( his\_sd, <where to put what you receive> )

send( his\_sd, <the stuff you want sent> )

close( my\_ sd )

end loop

**Steps specifically for a client socket**

• create a socket with the socket() system call

• connect the socket to the address of the server using the connect() system call

send and receive data

there are a number of ways to do this, but the simplest way is to use the read()/recv and write()/send system calls

**simple client steps (pseudocode)**

my\_sd = socket()

his\_sd = connect( my\_sd, <presumed adderss of some server> )

send( his\_sd, <the stuff you want sent> )

recv( his\_sd, <where to put what you receive> )

close( my\_sd)