IT304 Lab3
Introduction to Socket Programming

1 TCP Vs. UDP

TCP is a transport layer protocol used by applications that require guaranteed delivery. It is a sliding window protocol that provides handling for both timeouts and retransmissions. TCP establishes a full duplex virtual connection between two endpoints. Each endpoint is defined by an IP address and a TCP port number. The operation of TCP is implemented as a finite state machine. The byte stream is transferred in segments. The window size determines the number of bytes of data that can be sent before an acknowledgement from the receiver is necessary.

The User Datagram Protocol offers only a minimal transport service – non-guaranteed datagram delivery and gives applications direct access to the datagram service of the IP layer. UDP is used by applications that do not require the level of service of TCP or that wish to use communications services (e.g., multicast or broadcast delivery) not available from TCP. UDP is almost a null protocol; the only services it provides over IP are checksumming of data and multiplexing by port number. Therefore, an application program running over UDP must deal directly with end-to-end communication problems that a connection-oriented protocol would have handled – e.g., retransmission for reliable delivery, packetization and reassembly, flow control, congestion avoidance, etc., when these are required.

2 Socket programming introduction

In computer networking, an Internet socket or network socket is an endpoint of a bidirectional inter process communication flow across an internet protocol based computer network such as the internet. The term internet socket is also used as a name for an application programming interface (API) for the TCP/IP protocol stack, usually provided by the operating system. Internet sockets constitute a mechanism for delivering incoming data packets to the appropriate application process or thread, based on a combination of local and remote IP address and port numbers. Each socket is mapped by the operating system to a communicating application process or thread. A socket address is the combination of an IP address (the location of the computer) and a port (which is mapped to the application program process) into a single identity, much like one end of a telephone connection is the combination of a phone number and a particular extension.

Two types of internet sockets:

1. Stream sockets: They are connection oriented reliable sockets also called TCP sockets.
2. Datagram sockets: They are connectionless unreliable sockets also called UDP sockets

3 TCP client server model

Most interprocess communication uses the client server model. These terms refer to the two processes which will be communicating with each other. One of the two processes, the client, connects to the other process, the server, typically to make a request for information. Notice that the client needs to know of the existence of and the address of the server, but the server does not need to know the address of (or even the existence of) the client prior to the connection being established.

Notice also that once a connection is established, both sides can send and receive information. The system calls for establishing a connection are somewhat different for the client and the server, but both involve the basic construct of a socket. A socket is one end of an interprocess communication channel. The two processes each establish their own socket.

The steps involved in establishing a socket on the client side are as follows:

3.1 Server side commands

1. **Socket creation:**
   int sockfd = socket(domain, type, protocol)
   - sockfd: socket descriptor, an integer (like a file-handle)
   - domain: integer, communication domain e.g., AF_INET (IPv4 protocol) , AF_INET6 (IPv6 protocol)
   - type: communication type
     SOCK_STREAM: TCP(reliable, connection oriented)
     SOCK_DGRAM: UDP(unreliable, connectionless)
   - protocol: Protocol value for Internet Protocol(IP), which is 0.This is the same number which appears on protocol field in the IP header of a packet.
2. **Setsockopt:set the socket options**
   *int setsockopt(int sockfd, int level, int optname, const void *optval, socklen_t optlen);*

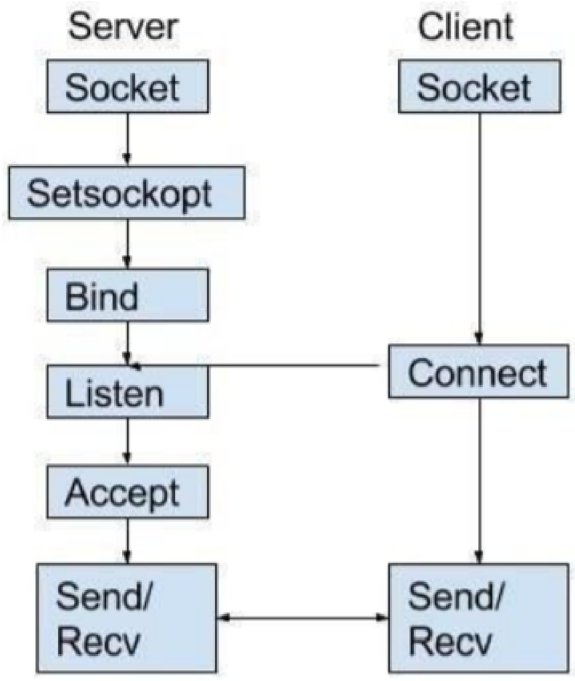

Figure 1: Caption

This helps in manipulating options for the socket referred by the file descriptor sockfd. An application program can use setsockopt() to allocate buffer space, control timeouts, or permit socket data broadcasts.

3. **Bind:**
*int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);*
After creation of the socket, bind function binds the socket to the address and port number specified in addr(custom data structure). In the example code, we bind the server to the localhost, hence we use INADDR_ANY to specify the IP address.

4. **Listen:**
int listen(int sockfd, int backlog);
It puts the server socket in a passive mode, where it waits for the client to approach the server to make a connection. The backlog, defines the maximum length to which the queue of pending connections for sockfd may grow. If a connection request arrives when the queue is full, the client may receive an error with an indication of ECONNREFUSED.

5. **Accept:**
*int new_socket= accept(int sockfd, struct sockaddr *addr, socklen_t *ad-*

drlen);
It extracts the first connection request on the queue of pending connections for the listening socket, sockfd, creates a new connected socket, and returns a new file descriptor referring to that socket. At this point, connection is established between client and server, and they are ready to transfer data

3.2 Client side commands

1. Socket connection: Exactly same as that of server's socket creation

2. **Connect:**
   *int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);*
   The connect() system call connects the socket referred to by the file descriptor sockfd to the address specified by addr. Server's address and port is specified in addr

3. For sending and receiving data *read()* and *write()* system calls are used.

4 UDP client server model

Commands for UDP client server model is shown in following figure:

All the other commands have property same as TCP but there are two new commands here

1. **Sendto():**Send a message on the socket *ssize_t sendto(int sockfd, const void *buf, size_t len, int flags, const struct sockaddr *dest_addr, socklen_t addrlen)*

   Arguments :

   - sockfd – File descriptor of socket
   - buf – Application buffer containing the data to be sent
   - len – Size of buf application buffer
   - flags – Bitwise OR of flags to modify socket behaviour
   - dest_addr – Structure containing address of destination
   - addrlen – Size of dest_addr structure

2. **rcvfrom():** Receive a message from the socket. *ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags, struct sockaddr *src_addr, socklen_t *addrlen)*

   Arguments :

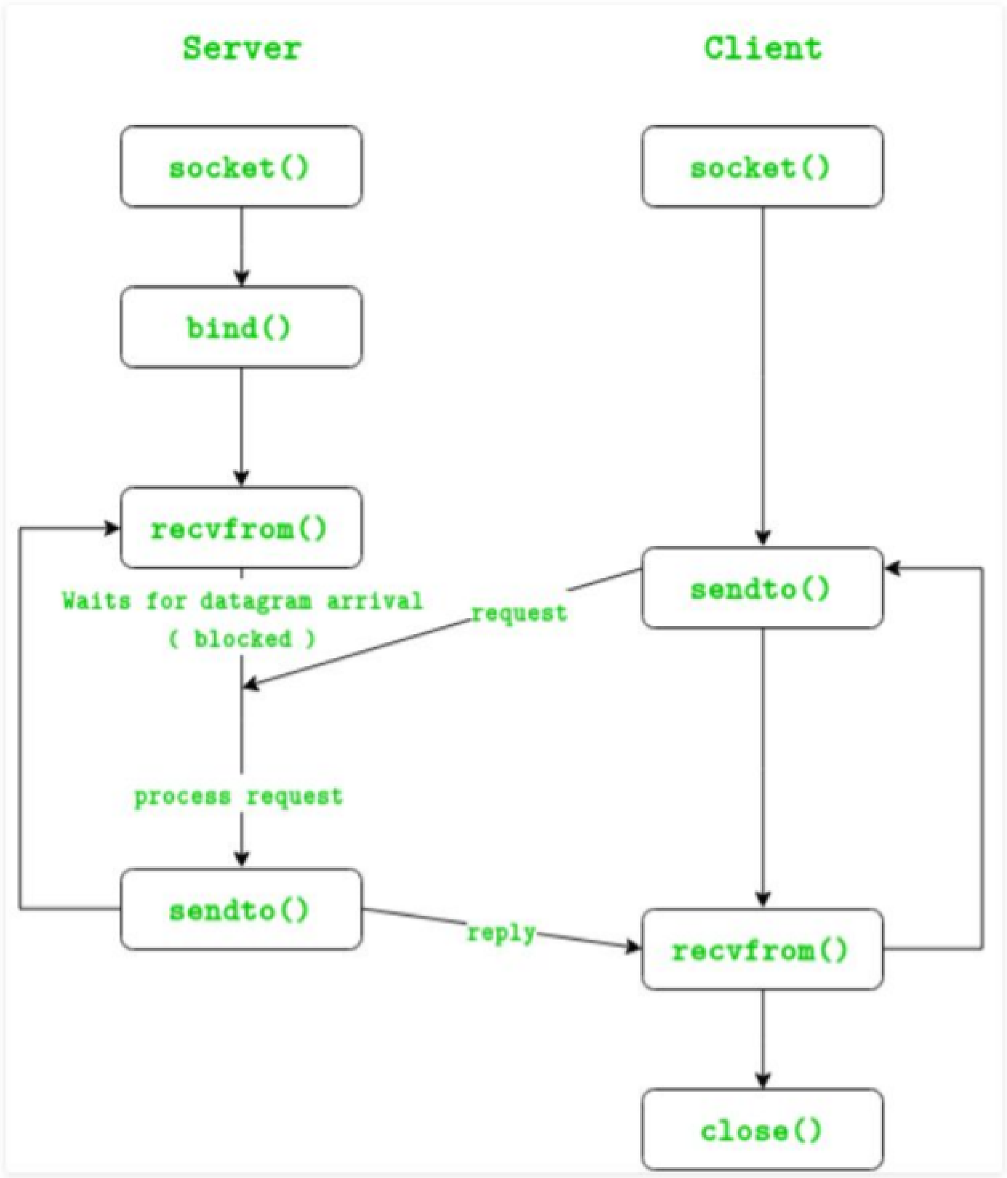

Figure 2: Caption

- sockfd – File descriptor of socket
- buf – Application buffer in which to receive data
- len – Size of buf application buffer

- flags – Bitwise OR of flags to modify socket behaviour
- src_addr – Structure containing source address is returned
- addrlen – Variable in which size of src_addr structure is returned

3. **Close():** Close a file descriptor
   int close(int fd)
   Arguments :
   fd – File descriptor

5 Multithreading in c

5.1 What is a Thread?

A thread is a single sequence stream within in a process. Because threads have some of the properties of processes, they are sometimes called lightweight processes .

5.2 Why Multithreading?

Threads are popular way to improve application through parallelism. For example, in a browser, multiple tabs can be different threads. MS word uses multiple threads, one thread to format the text, other thread to process inputs, etc.

Threads operate faster than processes due to following reasons:

1. Thread creation is much faster.
2. Context switching between threads is much faster.
3. Threads can be terminated easily
4. Communication between threads is faster.

Pthread library is used to perform multithreading in C.

Functions in Pthread;

- pthread_create: creates new thread
- pthread_join: Joins two threads

NOTE: To run pthread in gcc compiler
CMD: gcc -pthread -o term [FILENAME]

6 Exercise

6.1 Exercise 1

6.1.1 Exercise 1a

Study simple client-server hello message program of TCP and UDP. Compile and run the program and understand output

6.1.2 Exercise 1b

Study multithread sample code. Compile and run the program and understand output.

6.2 Exercise 2

6.2.1 Exercise 2a:FTP using socket programming

Design an application of FTP using TCP. GET method is given to you using that Implement PUT and LIST methods in FTP. Design your own message structure. Use appropriate data structure.
Hint:[TCP client server code is available in folder]

6.2.2 Exercise 2b:Multithreaded server

Extend the application you have designed in Exercise 2a such that server can handle multiple clients in parallel. Use multithreading. *[Hint:`https://dzone.com/articles/parallel-tcpip-socket-server-with-multi-threading`]*

6.2.3 Exercise 2c:

Implement a scenario in which there is one server (PC1) which can handle multiple clients (using multithreading implemented in exercise 2b) and one client (PC2). Server is supposed to send a file of 50Mb for each request. Client (PC2) can send multiple requests (by generating multiple threads) for the same file.

- Find the time required to transfer file for each request of client. (Hint: Use time function used in linux shell `https://www.faqforge.com/linux/determine-execution-time-command-linux/`)
- Change the number of requests generated from 1to 10. Create a table for time required to transfer file to any one client(fixed) for different number of requests.
- Plot a graph of time required to transfer file for any one client(fixed) vs. different number of requests.

6.2.4 Exercise 2d

Implement a scenario in which two clients are supposed to access the same file at the server. Provide a locking mechanism which prevents them to access the file at the same time.(Hint: Use code with mutex and locking mechanism from reading material)

6.3 Exercise 3

Goal of this exercise is to implement a scenario where all the clients of the server receives the file at the same time. (Such as online game or stock market etc)
Implement a scenario in which there is one server and multiple clients.
Periodic measurement of RTT

1. Server sends echo message to all clients
2. for each client i
   (a) measure the timestamp when echo message is sent (T_{si})
   (b) Measure the timestamp when reply is received (T_{ri})
   (c) $RTT_i = T_{ri}T_{si}$
3. $\Delta T = max(RTT_i)$

Clients send request to get a file(50Mb) from server. Server sends the file with timestamp(Tms) and ΔT . When client receives file(at time Tmr) it holds file till (ΔT -(Tms - Tmr)) time. After that it displays file. If the client receives file after ΔT then it sends request to server to update value of ΔT .

7 Submission guidelines

Submit a zip file which contains

- code of exercise 2 and 3 with naming convention exercisename_client/server i.e. 2a_client, 2a_server etc.
- Graph of exercise 2c