LAB 3 – SOCKET PROGRAMMING – UDP

DATE : 16.08.2024

BATCH : N,P,Q

Exercise 1

Design a client-server application using connection less protocol. A typical communication between the client and server in the application is as follows:

- client sends the server the number of commands to be executed
  for as many times as the number sent,
  - client sends a command to the server
  - client sends an input for the command
  - server processes the input based on the command and sends the output to the client

A few sample commands and the corresponding action to be taken are listed below:

Command	Action	Sample Input	Sample Server Response
CASE CONVERSION	Convert lower case to upper case and upper case to lower case	Anna University	aNNA uNIVESITY
		welcome WELCOME	WELCOME welcome
PRIME	Checks if prime or even number	17	prime number
		24	Not prime number
		hi	Invalid Input
STRING ab	Number of a 's and b 's	aabba	3 2
		bbbaabbb	2 6
		cdezhi	Invalid Input
		123	Invalid Input
<Any other>	None	<any>	Invalid Command

As indicated above, the server gives an error message in case it receives an invalid command / input from the client.

Exercise 2

Write a UDP Client-Server socket program for the following scenario:

The UDP server has a lookup table consisting of domain names (Eg., annauniv.edu, google.com, ieee.org, yahoo.co.in,) and their respective IP addresses. The UDP client contacts the UDP server to get the IP address for an application server by specifying its domain name. The server in turn, retrieves the IP address from the lookup table and sends the same to the client.

UDP

Use of the User Datagram Protocol, also known as UDP, is widespread on the Internet for time-sensitive transmissions like DNS lookups and video playback. By not explicitly establishing a connection before data is sent, it speeds up communications. This enables the delivery of data incredibly quickly, but it can also result in packets being lost in transit, opening up the potential for DDoS assaults and other forms of exploitation. **UDP** is a defined procedure for transferring data between two computers connected by a network, like all networking protocols. In contrast to other protocols, UDP completes this task in a straightforward manner: it delivers packets (units of data transmission) straight to the destination computer without first establishing a connection, specifying the order of such packets, or verifying that they arrived as intended. The term "datagram" is used to describe UDP packets.

As a more popular transport protocol, UDP is quicker but less dependable than TCP. Through an automatic procedure known as a "handshake," the two computers in a TCP communication first create a connection. The actual transfer of data packets between the two computers won't happen until after this handshake is complete. This procedure is skipped during UDP communications. Alternatively, two computers can just start communicating by exchanging data.

TCP communications also validate that data packets arrive in the intended order and specify the order in which they should be received. TCP mandates that a packet is resent if it is not received, such as when there is network congestion in the intermediary network. None of these features are present in UDP communications.

Some advantages are produced by these distinctions. Data can be transferred significantly more quickly using UDP than using TCP since it doesn't require a "handshake" or check to see whether the data is correct.

This rapidity, nevertheless, results in compromises. A UDP datagram will not be sent again if it is lost in transit. Therefore, UDP-using applications need to be able to endure errors, loss, and duplication.

Technically speaking, such packet loss is less of a problem with UDP as it is a result of how the Internet is designed. Due to the impractical amount of additional memory required, the majority of network routers do not by default conduct packet sequencing and arrival confirmation. When an application needs it, TCP is a technique to close this gap.)

In connections that must be completed quickly, UDP is frequently employed since it is preferable to wait than to infrequently lose packets. Due to the fact that both voice and video communications are time-sensitive and built to withstand some degree of loss, they are both sent via this protocol. For instance, UDP is utilized by many internet-based telephone services that use VOIP (voice over IP), which is a type of communication. This is due to the fact that staticky phone conversations are better than crystal-clear but choppy ones.

The best protocol for online gaming is UDP because of this. In a similar way, DNS servers function using UDP since they must be both quick and effective.

Implementation of UDP Client Server Program

There are some fundamental differences between TCP and UDP sockets. UDP is a connection-less, unreliable, datagram protocol (TCP is instead connection-oriented, reliable and stream based).

There are some instances when it makes to use UDP instead of TCP. Some popular applications built around UDP are DNS, NFS, SNMP and for example, some Skype services and streaming media.

In UDP, the client does not form a connection with the server like in TCP and instead, It just sends a datagram. Similarly, the server need not to accept a connection and just waits for datagrams to arrive. We can call a function called connect() in UDP but it does not result anything like it does in TCP. There is no 3 way handshake. It just checks for any immediate errors and store the peer's IP address and port number. connect() is storing peers address so no need to pass server address and server address length arguments in sendto().

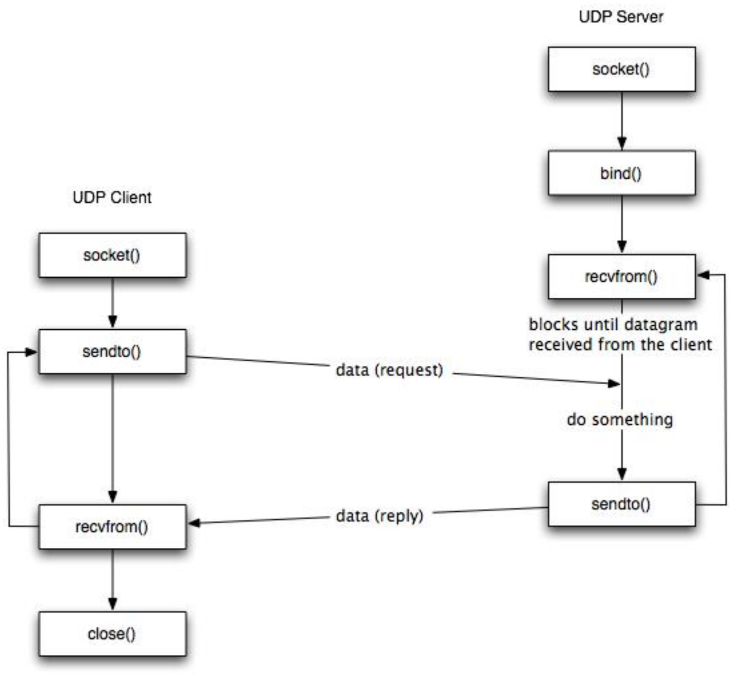
Figure 4 shows the interaction between a UDP client and server. First of all, the client does not establish a connection with the server. Instead, the client just sends a datagram to the server using the sendto function which requires the address of the destination as a parameter. Similarly, the server does not accept a connection from a client. Instead, the server just calls the recvfrom function, which waits until data arrives from some client. recvfrom returns the IP address of the client, along with the datagram, so the server can send a response to the client.

As shown in the Figure, the steps of establishing a UDP socket communication on the client side are as follows:

- Create a socket using the socket() function;
- Send and receive data by means of the recvfrom() and sendto() functions.

The steps of establishing a UDP socket communication on the server side are as follows:

- Create a socket with the socket() function;
- Bind the socket to an address using the bind() function;
- Send and receive data by means of recvfrom() and sendto().



The recvfrom() Function

This function is similar to the read() function, but three additional arguments are required. The recvfrom() function is defined as follows:

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

ssize_t recvfrom(int sockfd, void* buff, size_t nbytes,
                 int flags, struct sockaddr* from,
                 socklen_t *addrlen);
```

The first three arguments sockfd, buff, and nbytes, are identical to the first three arguments of read and write. sockfd is the socket descriptor, buff is the pointer to read into, and nbytes is number of bytes to read. In our examples we will set all the values of the flags argument to 0. The recvfrom function fills in the socket address structure pointed to by from with the protocol address of who sent the datagram. The number of bytes stored in the socket address structure is returned in the integer pointed by addrlen.

The function returns the number of bytes read if it succeeds, -1 on error.

The sendto() Function

This function is similar to the send() function, but three additional arguments are required. The sendto() function is defined as follows:

```
#include <sys/socket.h>
ssize_t sendto(int sockfd, const void *buff, size_t nbytes,
               int flags, const struct sockaddr *to,
               socklen_t addrlen);
```

The first three arguments sockfd, buff, and nbytes, are identical to the first three arguments of recv. sockfd is the socket descriptor, buff is the pointer to write from, and nbytes is number of bytes to write. In our examples we will set all the values of the flags argument to 0. The to argument is a socket address structure containing the protocol address (e.g., IP address and port number) of where the data is sent. addlen specified the size of this socket.

The function returns the number of bytes written if it succeeds, -1 on error.

Necessary Functions

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

Creates an unbound socket in the specified domain.

Returns socket file descriptor.

Arguments:

- **type -** the kind of socket that should be generated (SOCK STREAM for TCP and SOCK DGRAM for UDP).
- **protocol** - The socket will use this protocol.
- **0** indicates to use of the address family's default protocol.

int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen)

Assigns the address to the unbound socket.

Arguments:

- **sockfd**, which is the file descriptor for a socket that will be bonded
- Structure, where the address to be tied to, is supplied (**addr**)
- the size of the addr structure in **addrlen**

ssize_t sendto(int sockfd, const void *buf, size_t len, int flags,

        const struct sockaddr *dest_addr, socklen_t addrlen)

Send a message on the socket

Arguments:

- **Arguments**: sockfd - the socket's file descriptor;
- **buf -** the application buffer containing the data to be transferred
- Size of the application buffer (**len**)
- **flags** - Bitwise OR of socket behavior flags
- Structure called dest **addr** that contains the destination's address
- the dest addr structure's **addrlen** value

ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags, struct sockaddr *src_addr, socklen_t *addrlen)

Receive a message from the socket.

Arguments:

- the socket's file descriptor (**sockfd**),
- the application buffer (**buf**),
- the size of the application buffer (**len**),
- **flags** - Bitwise OR of flags to change the behavior of the socket
- the structure src **addr**, which contains the source address, is returned.
- the variable **addrlen** returns the size of the src addr structure.

int close(int fd)

Close a file descriptor

Arguments:

- File descriptor (**fd**)