Lab 6

1) In the case of *mypingd* application, the values obtained for the completion time in both of the models is shown below.

Type	Completion Time(msec)
Overlay-Router (10 Routers)	6.141
Server-Client	0.512

The completion time for the case of server-client architecture is much faster than that of overlay-router application because of the overhead caused by the extra steps taken (forwarding to different routers, when compared to direct client-server) in this case. The values also demonstrate the same.

When using the traffic receiver app, the following are the throughput values obtained using the tunnel server and the normal client server app.

Type	Sender(Mbps)	Receiver(Mbps)
Overlay-Router (10 Routers)	136.4	142.3
Server-Client	135.8	141.5

As observed, there is no difference in the throughput values of the application in both the cases. Though there is an overlay path in the middle in the former case, the throughput values that are calculated in both of the given applications (*traffic_snd, traffic_rcv*) depend only on the transit to the next hop.

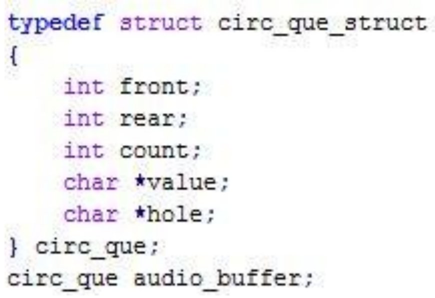
In case of the client-tunnel-server case, throughput for *traffic_snd* is calculated in between client-end_router, and for *traffic_rcv* it is in between end_router-server. Whereas in the latter scenario of server-client architecture, *traffic_snd* is calculated in between client-server, and for *traffic_rcv* it is in between client-server. Thus in effect it is always in between two nodes. Hence there is no difference in their values observed.

2) File server application using UDP is implemented by using an approach similar to the audiostream application developed in Lab 5.

Before sending any data, turboServer application appends the sequence number of the packet to the payload data. It then keeps sending packets at a regular interval of 100 nano-seconds (tested with various values at both server and receiver and found the mentioned value to be optimal) till the end of file is reached. In addition to sending the data, the server also maintains a circular buffer where it holds the information of the last 100 packets that are sent. Client is designed in such a way that it drops every *lossnum'th* packet that is received by it from the server. Client upon dropping the packet sends a request for a retransmission packet with the sequence number of the missing packet(negative-ack). Server checks if the negative-ack'ed packet is present in the buffer it maintains (via the sequence number) and if present, retransmits the packet to the client.

Client upon receiving the data from the server puts the data into a circular buffer which is later read continuously from the buffer and written to a file as done with the audiolisten application. This gives the client some buffer time to handle the loss of packets. As mentioned above, client drops every *lossnum'*th packet and when it receives a retransmitted packet checks if the current written value of the buffer is less than the sequence number of the incoming retransmitted packet. If that's the case, it adds the incoming packet into the circular buffer at its correct location. Else it drops the packet. Shown below is circular buffer data structure that is used at the client side for implementation.

The data structure used is a circular buffer. There is a front position which checks the position from where to write to /filename. The rear position checks where to write the value received from sigpoll to circular buffer. The value pointer stores the value of the buffer (preceded with the sequence number). Hole pointer stores whether the value pointed at its position is a hole or not.



```
typedef struct circ_que_struct
{
    int front;
    int rear;
    int count;
    char *value;
    char *hole;
} circ_que;
circ_que audio_buffer;
```

In my program, I have sequence numbers starting from 10000. This number is initially extracted from the packet and checked with the front value of the circular buffer. If it is less than the sequence number pointed by the value at front (implying that it's successor part is already written to /filename) the packet gets discarded. Else, it checks if it's value is more than the previous sequence number by 1. If not, requests a retransmit packet and sets the value of the hole flag at that position to be 1(Initially, all of the hole values are initialized to 0).

Now, when the retransmitted packet occurs(flag=1), the packet's sequence number is tokenized from the first 5 bytes of the string and if it's sequence value is greater than front position's sequence number, it's data value gets stored at the positon of it's sequence number.

The performance of the application turned out to be decent enough. I was getting a throughput value of about 120Mbps on average for large and small files. The packet losses are minimal on average and on some of the occasions the drops are absolute zero.

Running the Program:

a) **Server Arguments**: *%turboserver portnumber secretkey configfile.dat lossnum*

   The server looks for the file in the directory ./filedeposit. If the specified file does not exist, the server closes the connection and ignores the request. If the file exists, the server transmits the content else it throws an error.

b) **Client Arguments**: *%turboclient hostname portnumber secretkey filename configfile.dat lossnum*

3) Self-congestion occurs in the case of greedy variant of the TCP when it starts getting too aggressive and tries to occupy the entire bandwidth. In such a scenario, the problem arises when the sender sends more bytes than the receiver window could handle. This leads to a loss of huge number of packets, which need to be retransmitted again, thus effectively decreasing the throughput of the sender.

   It seems obvious from the above discussion to not just send packets at a faster rate but also to consider the window size available at the receiver side. Thus, in a greedy environment it makes sense to increase the sender rate up until the point where the receiver window is not completely filled up and once that's done, it should keep sending from that point until the receiver window size expands, at that same constant rate. Whenever, the receiver window size expands, it could again follow a similar approach described above.