Problem 1:

Ping:

Time : 0.695068 ms
ip address: 128.10.3.61 port: 59921

Time : 2.700928 ms
ip address: 128.10.25.101 port: 57343

Traffic generator:

No VPN:
Completion time 0.109000,reliable bps 902055.045872, pps 862.385321.

VPN:

Completion time 0.407000,reliable bps 257002.457002, pps 245.700246.

Mutlliple router:

Completion time 0.817100,reliable bps 128013.479378 , pps 122.3839.

We can see from the above data that, compared with no routing, routing would significantly reduce the throughput of the app.
And when under multiple router, the throughput of the server will also reduce.
The overhead of the overlay router are to be blamed for the decrease of throughput performance.

As the overlay server will have to read and retransfer the data to the real server, the more overlay server will have, the higher the overhead would be.

Problem 2:

Design:

For the server, once got file transfer request and confirm that the path exist. It will fork a child process to bind a socket for file transfer.

A sliding window is implemented to assure reliable file transfer.
Server will read the file with size of 1000 and create an array of WINSZ to store the packet. At the beginning, the server will read the array in sliding window manner, and send the packet to client.
However, it will not send any packet until it got the ACK from client. For negative ACK, server will just send the missing packet again if it is still in the sliding window array.
For positive ACK, it will check whether the current position of sliding window match the content of ACK. Thus, it can adjust the window according. Since the positive ACK means that the client already got the acknowledged packet n. The sliding window will move forward to send the packet n+1 in the sliding window.

On client end, it will send a request for file and once confirmed it will start accept packet from server.
Client also follow a sliding window manner to make sure reliable transfer.
The packets received from server are stored in an array of WINSZ. The sequence is strictly followed to make sure that packet n is stored at n%WINSZ position of the sliding window.
Once the client received packet n, it will send a postive ACK back to server to inform this.
Thus the server will knows to send the n+1 packet.

Once the window is filled, client will start to write the packet from the sliding window to local file. And move forward.

A timer which alarms at constant time is set, once alarmed, an alarm handler will go through the sliding window to check for lost packet. If one is lost, client will send the negative ACK to server, so server can resend the lost packet.

Result;

Pp.au
50% lost

completion time 243416 usec, bytes transferred 920316, reliable throughput 3 mbps

Kline-jarrett.au
completion time 744572 usec, bytes transferred 2898266, reliable throughput 3 mbps

10% lost

Pp.au
completion time 199232 usec, bytes transferred 920316, reliable throughput 4 mbps

Kline-jarrett.au
completion time 583033 usec, bytes transferred 2898266, reliable throughput 4 mbps

We can see that the higher the packet lost rate, the lower the reliable throughput.

Tcp
Pp.au
completion time 20094 usec, bytes transferred 920316, reliable throughput 45 mbps

Kline-jarrett.au
completion time 83484 usec, bytes transferred 2898266, reliable throughput 34 mbps

As it is not easy to make TCP packet lost in our environment. It is clear that the TCP throughput is still bigger than our UDP.

BONUS

TCP-greedy utilize the fact that upon detecting possible packet loss, other TCP flows will back off and give away their share of bandwidth. Instead of back off as well, our TCP-greedy algorithm will not back off and keep increase exponentially.

Meanwhile in order not to shoot oneself in the shoot, it will keep track of packet loss by detecting the ACK. If got three consecutive ne

This algorithm is developed by modifying Slow Start:

Reset CongestionWindow to 1

Step 1. Perform exponential increase
CongestionWindow ← CongestionWindow + 1

Step 2. Until timeout at start of connection: Instead of cutting the CongestionWindow. We would adjust the timeout to try to get the ACK:
Timeout ← 2*Timeout
Step 3. If doubling the timeout does help to receive ACK. Then increase the congestionWindow.
CongestionWindow←CongestionWindow + 1

Repeat 2 and 3 until increasing the Timeout does not help.

If increasing Timeout does not help. Instead of cut CongestionWindow by half or reset to 1. We decrease the CongestionWindow linearly.
CongestionWindow ← CongestionWindow - 1/CongestionWindow

We should keep track of the last CongestionWindow and Timeout that could get ACK.

Then this combination is the one we need.