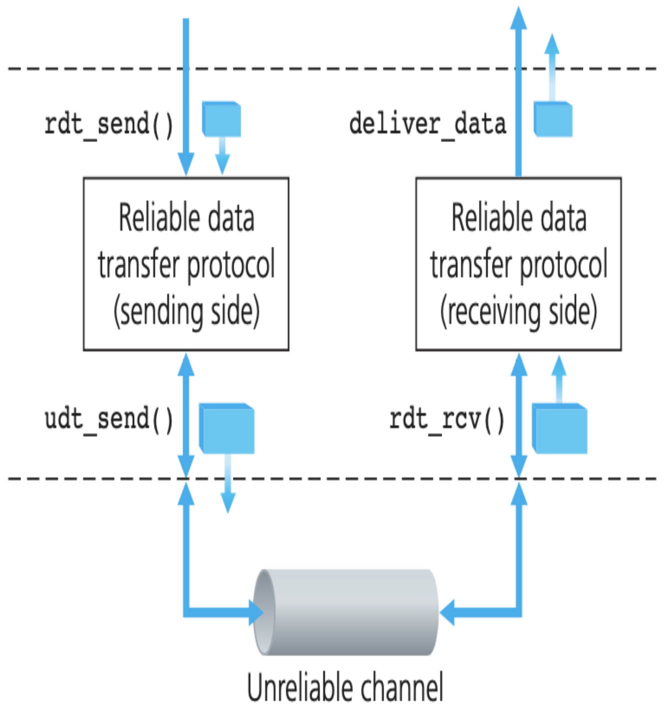
Application layer →

UDP →

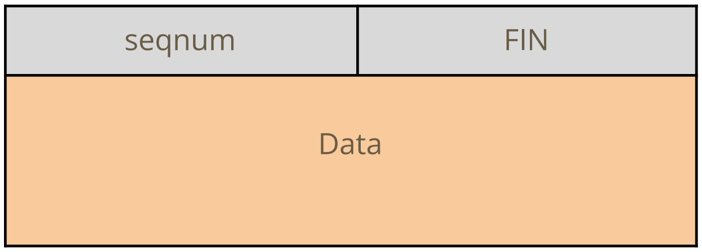

rdt_send()
deliver_data
Reliable data transfer protocol (sending side)
Reliable data transfer protocol (receiving side)
udt_send()
rdt_rcv()
Unreliable channel

OUR UDP-FTP

Gayala Manoj (CS19BTECH11011)
Gubbala Suraj (CS19BTECH11024)
Vinesh (AI19BTECH11023)
Srikar Jilugu (MA19BTECH11005)
Rahul S (AI19BTECH11002)

Our UDP-FTP offers reliable data transfer(RDT) features by implementing some of the basic and important principles of RDT at the application level (i.e FTP) and using UDP as transport layer protocol.

Our Application header consists of two header fields: seqnum and FIN which are combined with the data field to complete the structure of our application-layer packet.



This segment structure is presented as a python class **Packet** with the following attributes:

- seqnum : int
- FIN: bool
- Data: bytes

The features offered by our application as part of RDT are:

- Packet acknowledgment
    - Cumulative ACKS
- Packet loss detection
    - Timeout
    - Duplicate ACKS
- Packet retransmission
    - Fast Retransmit
- Pipelined RDT
- Synchronous jobs at the sender
- Delayed ACKS
- Dynamic Timeout and Round-Trip Time Estimation
- Congestion Control
- Connection Management

Packet acknowledgment

As our client and server operate on different end systems, we use acknowledgments(ACKs) from the server which are sent as response to the packets previously received. This receiver feedback helps the client(sender) to know which set of packets are correctly received by the server. ACKs are packets which have the **seqnum** field filled with the sequence number of the packet that is being acknowledged and are empty in the **data** field

Cumulative ACKS

We use cumulative ACKS as part of our receiver feedback, i.e when the client/sender receives an ACK from the server, client can be assured that all the packets from the last unacknowledged packet to the packet corresponding to this ACK are successfully received.

Packet loss detection

Detection of packet loss by client/sender is done by

- **Timeout** of the socket at the client side which is set with a custom delay.
- Three consecutive **duplicate ACKs** from the server.

Timeout

Whenever the client socket times out, the sender will be indicated that there might be a packet loss of the earliest (ordered by seqnum) unacknowledged packet.

Duplicate ACKs

If the client receives three consecutive same ACKs of a previously acknowledged packet from the server, then the client will be similarly indicated that the earliest unacknowledged packet is lost/not received by the server.

Packet retransmission

Packet retransmissions are done whenever packet loss is detected, only the earliest unacknowledged packet is retransmitted to the server.

Fast Retransmit

Along with the timeout scenario, retransmissions are also done when Duplicate ACKS are detected; this is done because repetitive ACKs for the same packet indicate that the next packet is lost with a high probability.

Pipelined RDT

To improve the performance of the protocol, **pipelining** is used, i.e client sends a set of packets simultaneously before waiting for acknowledgments. This is implemented using a sliding window with a finite(and variable) size which contains all the sent and unacknowledged packets

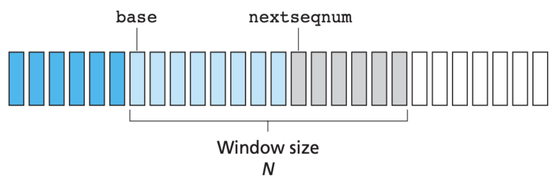


Illustration of sender's window

Receiver also maintains a buffer which stores all the packets with seq number higher than the expected packet and as soon as the expected packet is arrived, it sends an acknowledgment with the seq number of the last contiguous packet.

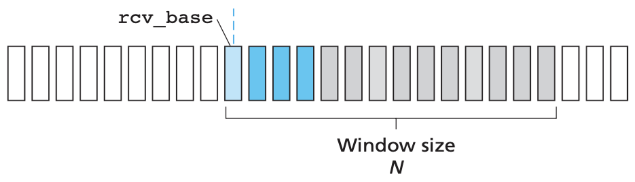


Illustration of receiver's window

Synchronous jobs at the sender

Application at the sender side has two threads which are used to handle received ACKS and pipeline send at the same time. This improves the performance of the protocol as it doesn't have to wait to check the ACKS until the new packets are sent.

Thread 1:

This is the main thread that sends the pipelined data to the receiver as long as the window is not full and when the data(file transfer it sits idle until the other thread finishes its job.

Thread 2:

This thread handles the receiving of ACKS from the server and does the packet-loss detection, retransmission accordingly, this thread works until it knows that the server has received all the packets (discussed in connection management)

Delayed ACKS

When the server is receiving segments from the client, it waits greedily for some time(delay) so that it can expect a set of in-order segments and send a single cumulative acknowledgement with the seq number of the last received packet.

If it receives duplicate packets (i.e. the already received packets) it sends an ACK for the already acknowledged packet.

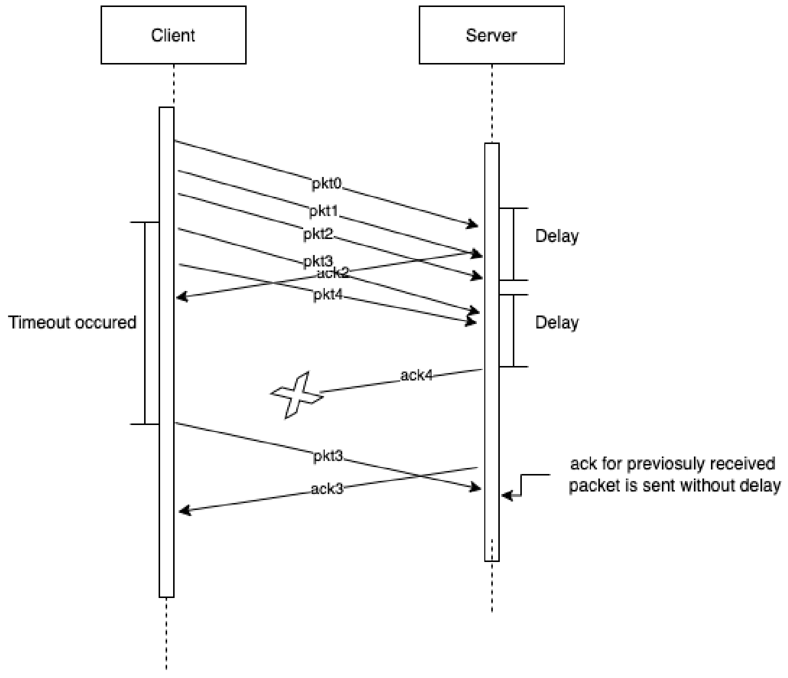


Illustration of Delayed ACKs at the receiver end

Dynamic Timeout and Round-Trip Time Estimation

The timeout for the acknowledgment of a sent packet at the client is dependent on the RTT estimated by our application. We estimate the RTT($EstimatedRTT$) and its deviations recursively using the below formulas.

$$EstimatedRTT = (1 - \alpha) * EstimatedRtt + \alpha * SampleRtt$$

$$DevRtt = (1 - \beta) * DevRtt + \beta * |SampleRtt - EstimatedRtt|$$

$$DevDevRtt = (1 - \gamma) * DevDevRtt + \gamma * ||SampleRtt - EstimatedRtt| - DevRtt|$$

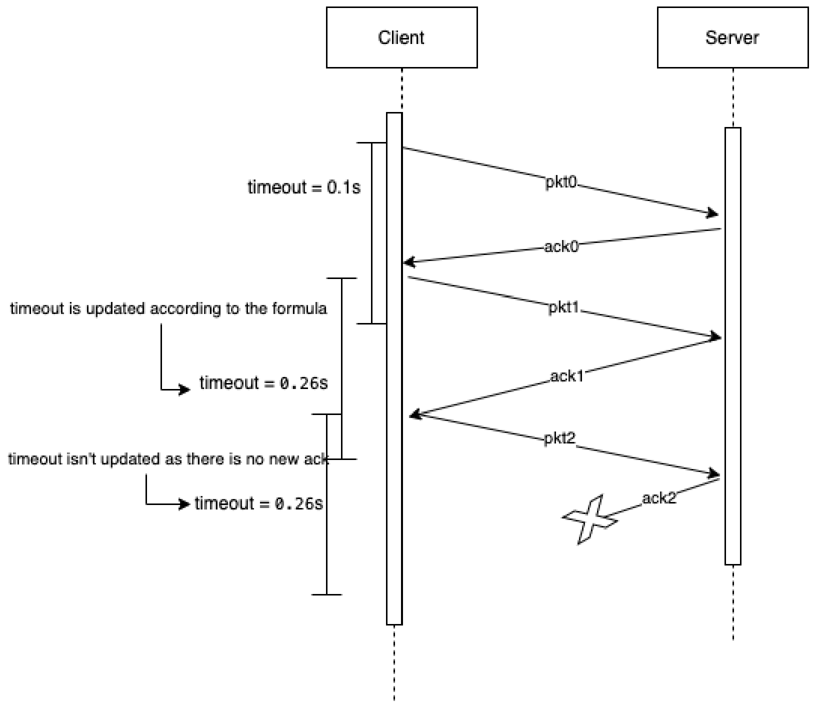
where $SampleRtt$ is the current calculated RTT, $DevRtt$ is first-order deviation of RTT and $DevDevRtt$ is the second-order deviation of RTT

α, β, γ are the parameters which are fixed to 0.125, 0.25, 05 respectively

The timeout is varied according to RTT so that the client can estimate the time taken for the next acknowledgement better. The formula used for finding the current timeout is:

$$Timeout = EstimatedRtt + 4 * DevRtt + 10 * DevDevRtt$$

This timeout update is done every time an acknowledgment for a new packet is arrived.


Client
Server
timeout = 0.1s
pkt0
ack0
timeout is updated according to the formula
pkt1
timeout = 0.26s
ack1
pkt2
timeout isn't updated as there is no new ack
ack2
timeout = 0.26s

Dynamic Estimation of RTT

Congestion Control

Our protocol also contributes to the congestion control by changing the window size according to the network traffic with an *end-end congestion control* mechanism This is done by the following algorithm we proposed:

We have two parameters α, β which are used in this algorithm

- If we receive an ACK for unacknowledged packets indicating that congestion in the network is low, then the window_size is increased as following:

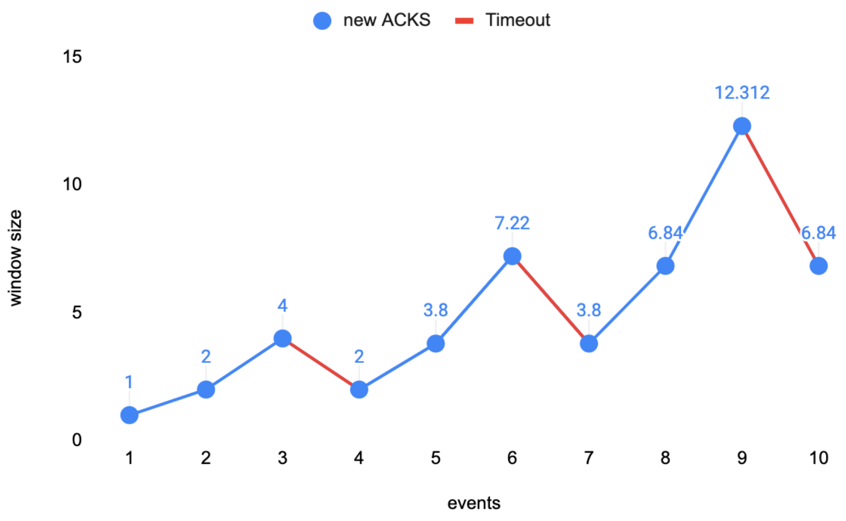
  $$WindowSize = min(WindowSize * \alpha, MaxWindow)$$

- If the socket gets timed out indicating there is some congestion in the network, then the window_size is decreased as following:

  $$WindowSize = WindowSize/\alpha$$

  $$\alpha = max(1, \alpha - \beta)$$

In out protocol, base values for α and β are 2 and 0.1 respectively and the $MaxWindow$ usedis 16. These values are set using observations and experiments.



Simulation of window size changes during the transfer

This algorithm tries to bring the window size to a stable value according to the network traffic. Traffic in the network is not tracked directly but the behavior of the receiving ACKS is used to interpret the congestion.

Connection Management

When the client gets assured that all the data chunks of the file are transferred (i.e all the packets sent are acknowledged) then a final packet with the FIN field set true is sent(reliably) and the client closes the connection when it receives the ACK for the final packet.

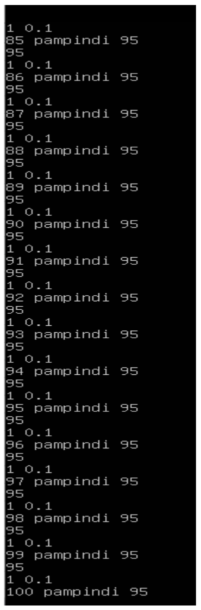
Report

We experimented with “Our UDP-FTP” for different values of delay and loss. We have configured them using the “tc” command.

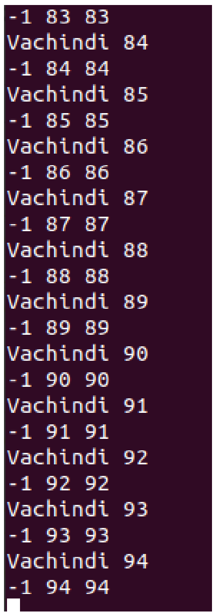
The tc command we used: *sudo tc qdisc change dev enp1s0 root netem loss 5% delay 100ms*

We tried for delay values of 0ms, 10ms, 50ms, 100ms and loss values of 0%, 2% and 5%. So we have taken a total of 12 combinations. The results have been noted in the sheet attached with this report.

The main challenge we faced was with 2% and 5% loss. When we tried transferring a file from one VM to another, we found that there is complete packet loss of a specific packet and as a result the client and server is getting stuck. We are attaching pics to prove our claim.



```
1 0.1
85 pampindi 95
95
1 0.1
86 pampindi 95
95
1 0.1
87 pampindi 95
95
1 0.1
88 pampindi 95
95
1 0.1
89 pampindi 95
95
1 0.1
90 pampindi 95
95
1 0.1
91 pampindi 95
95
1 0.1
92 pampindi 95
95
1 0.1
93 pampindi 95
95
1 0.1
94 pampindi 95
95
1 0.1
95 pampindi 95
95
1 0.1
96 pampindi 95
95
1 0.1
97 pampindi 95
95
1 0.1
98 pampindi 95
95
1 0.1
99 pampindi 95
95
1 0.1
100 pampindi 95
```

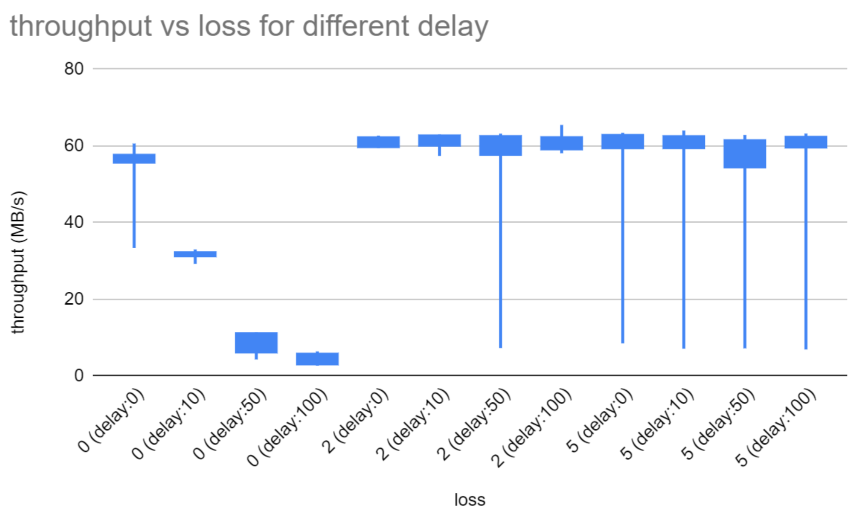


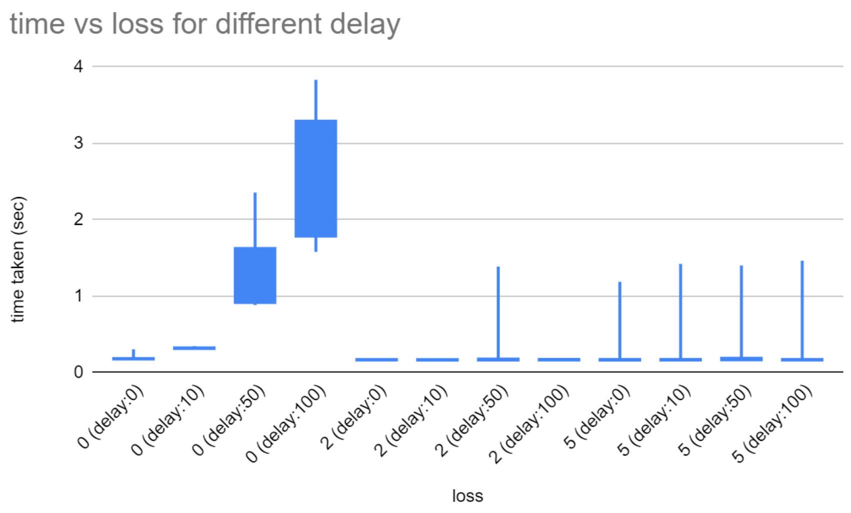
```
-1 83 83
Vachindi 84
-1 84 84
Vachindi 85
-1 85 85
Vachindi 86
-1 86 86
Vachindi 87
-1 87 87
Vachindi 88
-1 88 88
Vachindi 89
-1 89 89
Vachindi 90
-1 90 90
Vachindi 91
-1 91 91
Vachindi 92
-1 92 92
Vachindi 93
-1 93 93
Vachindi 94
-1 94 94
```

In the figures you can observe that the same packet has been retransmitted 100 times but it was not received at the server. So we have come up with a workaround and transferred files between the same VM using its IP address in the network. We know that this is a quick fix but we have come to this conclusion only after multiple hours of rigorous testing.

We experimented with a 10Mb file for various combinations of loss and delay.

The graphical representation of the throughputs are as follows:





The above charts can be viewed in the following *google sheet link* (charts page) and have filter options to have an explicit view of some particular combination we have run