Reliable UDP

Vishal Singh, 114708875
Deept Tripathi(113070705)
Harsha Vardhan(114394620)

Problem formulation and approach

In this project, we have implemented **reliability and ordering** features of the TCP on the application layer over UDP. In recent times, there have been renewed interests among enterprises in using UDP as their transport layer protocol. After the relatively less success of SPDY, Google came up with an out-of-the-box solution of QUIC in 2014. Google has claimed performance improvements like a reduction in YouTube video rebuffer by 15-18% and Google search latency by 3.6-8% using QUIC. We have taken inspiration from such application layer protocols and implemented our own Reliable-UDP protocol.

Formulation ::

Bursting of UDP packets

We understand that UDP is mainly used in multiplayer gaming and live-streaming because of its **lightweight** nature as compared to TCP. In such network systems, the sender just bursts packets of UDP and the receiver receives and evaluates those packets without sending any acknowledgement. However in **case of reliable UDP, we need to have acknowledgement** in order for the sender to retransmit any lost packet. Without taking flooding into account, It would be best practice for the sender to send as much data as possible and hope that most of it reaches the receiver. However to provide a guarantee of delivery, a response from the receiver containing information about the missing packet is necessary to create a reliable protocol.

feedBack from the receiver

The content of the feedback from the receiver will help the server know the missing packets and also decide what other packets it should send next. Apart from the information, it becomes extremely critical to decide what should be the frequency of this feedback from the receiver. **The frequency of the feedback should be just about right in order to efficiently manage packet** loss without flooding the network environment.

Based on the parameter of UDP packets bursting and feedback, we formulate 2 of the next simple algorithm below. In simple terms, The sender first segments the data into multiple packets like it is done in TCP. Each packet is assigned a relative sequence number to ensure the ordering when received by the server. The receiver collects these files in a buffer and continuously writes to a file the continuously available packets in the buffer.

Algorithm 1 ::

In this algorithm, the sender limitlessly bursts UDPs packets in the form of a sliding window. The receiver processes those packets and stores them in its buffer and then sends ACK packets with sequenceNo containing the next expected/missing packets that should be sent by the sender. Based on this received sequenceNo, the sender shifts its sliding window to begin from this acknowledged sequenceNo. The intention behind this logic is that even if some packets are lost, due to regular burst of packets, some known packets will eventually reach the hosts,

However a **drawback** of this approach is that the unnecessary bursting of packets will cause flooding on both the receivers side and sender side. This is evidently seen in our host programs running on python. The receival of a large number of packets which are not useful causes **duplicate processing of packets** on both sides. This is especially true in the case of the receiver.

Think of the scenario below as shown in figure 2, here just 1 packet loss of one middle sequence(packetNo 2) in a sliding window causes the sender to send multiple packets again and this causes duplicate packets to be received.

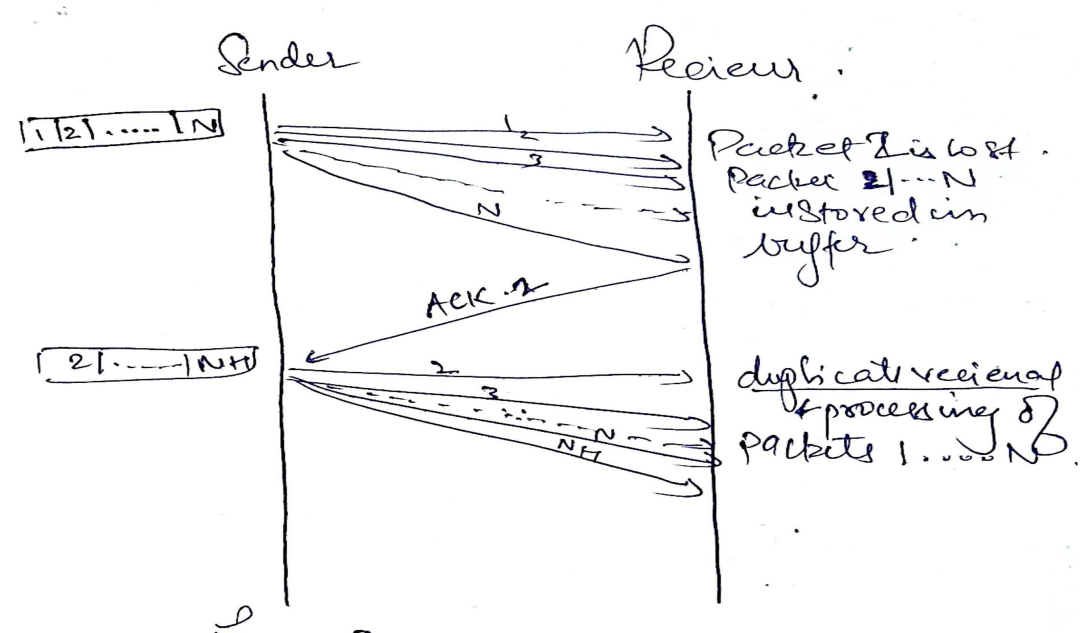

Sender
Reciever
1|2|.....|N
1
2
3
N
Packet 2 is lost.
Packet 2|...N is stored in buffer.
ACK 2
2|.....|N+1
2
3
N
N+1
duplicate recieving & processing of packets 1.....N.

Figure 2:-
Overhead of duplicate processing on reciever's side in Algorithm 1 :-

Algorithm 2 ::

This algorithm is very similar to the above algorithm with one major difference. In this algorithm on receiving a NACK packet , instead of shifting the sliding window completely backwards, The sender and the receiver only focus on transmitting the missing packets to avoid flooding on both sides.

Think of the scenario below as shown in figure 4, here after a packet loss of one middle sequence(packetNo 2) in a sliding window The sender only focuses on reliably sending the lost packet. After an acknowledgement is received for the lost packet which contains the next sequence expected, The sender again restarts the congestion window from this sequence Number

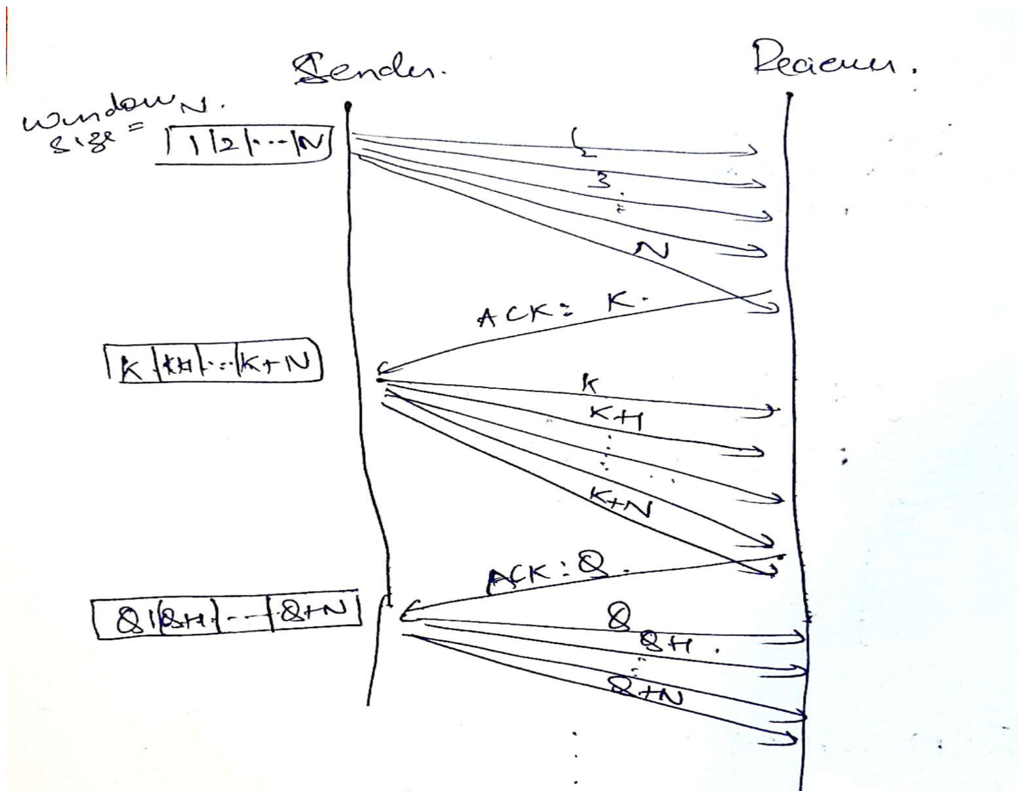


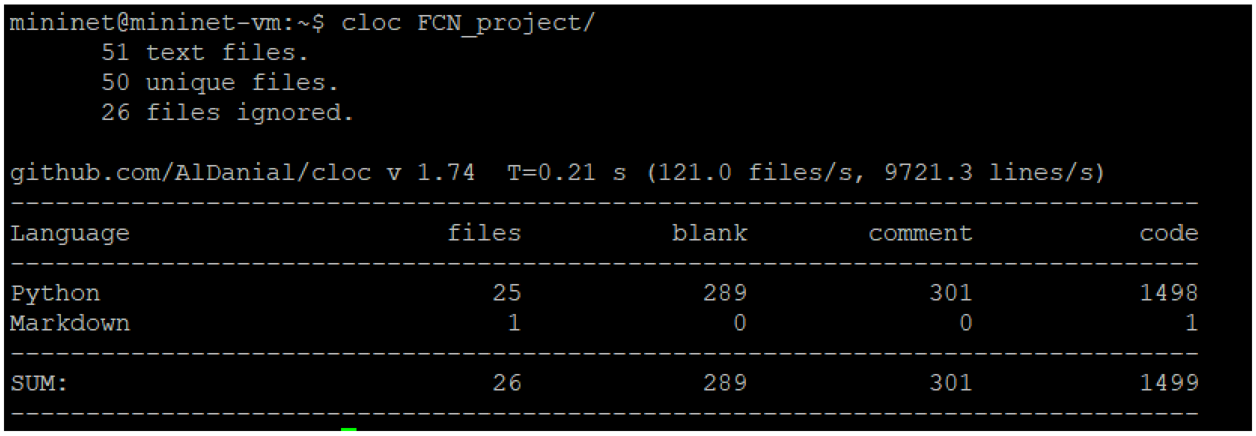
FIGURE: 1
Algorithm 1:
Very similar to GOBACKN

Note that we are not taking into account congestion control for our algorithm.

The algorithm performs better than algorithm 1 since this algorithm is not unnecessarily sending duplicate packets which prevents delay on the receiver side. We can imagine a scenario in which only one or 2 packets from the initial portion of a sliding window are lost. Due to this focused approached of sending packets , the receiver quickly recipes the missing packet and the sliding window can be moved forward quickly

Systems Contribution (coding, setting up environment etc)

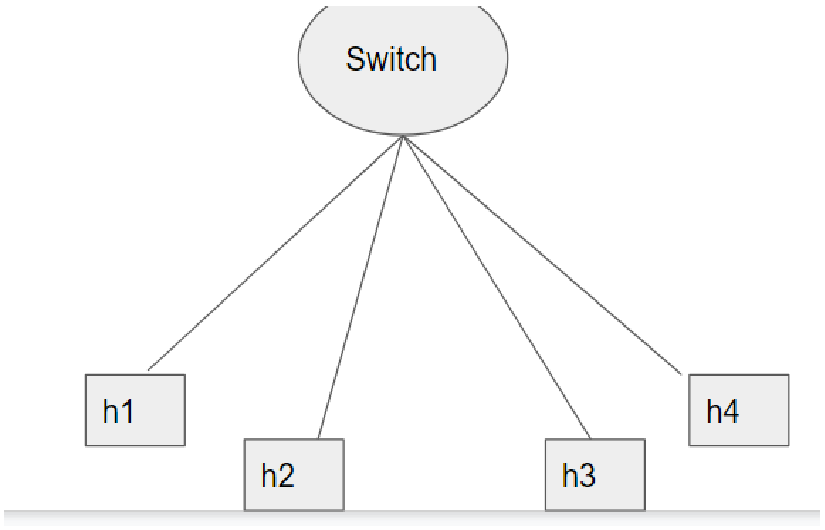
Coding :: our code for reliable UDP which includes the testing framework and graphs is present on github github link. Below are the details of the code,



```
mininet@mininet-vm:~$ cloc FCN_project/
      51 text files.
      50 unique files.
      26 files ignored.

github.com/AlDanial/cloc v 1.74  T=0.21 s (121.0 files/s, 9721.3 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
Python                          25            289            301           1498
Markdown                         1              0              0              1
-------------------------------------------------------------------------------
SUM:                            26            289            301           1499
-------------------------------------------------------------------------------
```

Setting up the environment :: We have used **mininet** to create our network topology and test our network. We test our RUDP in an environment of packet loss 1% (Since this is the generally accepted packet loss as per Wikipedia). We have kept our delay to be 5ms on the ethernet interfaces. **We are not much concerned about the Bandwidth or delay in the environment since decreasing delay and increasing delay will only slow down our experimentation**. Below is the Topology



Since we are mainly concerned with the reliability aspect of UDP hence mutual connections is what we focus on. We run the sender and receiver sockets/programs in pairs of (sender-receiver) h1-h2 and h3-h4.

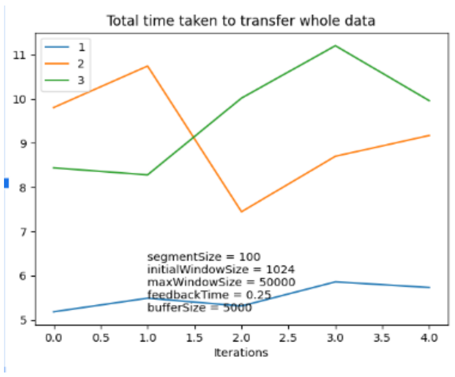
We run our mininet and run the sender receiver codes in xterm on the hosts. We tried to automate this process in the mininet using background processes. But we were facing some issues with consistency and parallel running of these socket processes

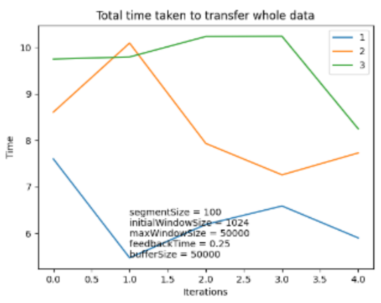
Detailed evaluation

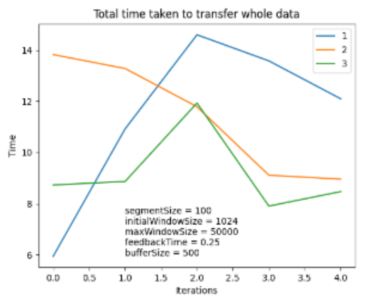
Transmission time::

Time Taken to transmit the whole file

When considering the transfer of files , it is important to consider files of various sizes. (In our evaluation, the type of file doesn't matter since we are reading and writing data in binary format). However we observed that we are getting **inconsistent results for small file sizes** of order 10^5 bytes. This is because of the fact that we have kept the segment size to be 100, which results in only 1000 packets for such files. This is explained in examples below







To obtain consistent results, we focus our analysis towards larger files of size 1000000 bytes. Considering the segments size of 100, this results into 10^4 segments which gives us good enough transmissions to measure behaviour

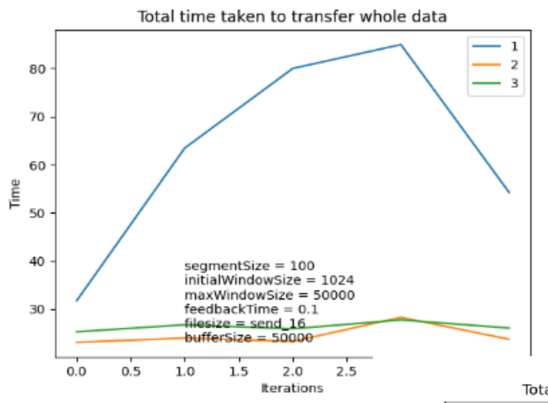
Thorough analysis of the evaluation and additional 30% (30 points)

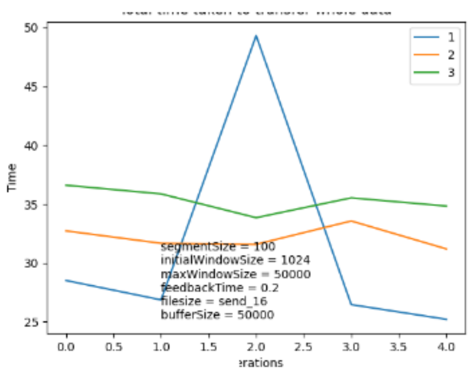
Consequences of the feedback timer::

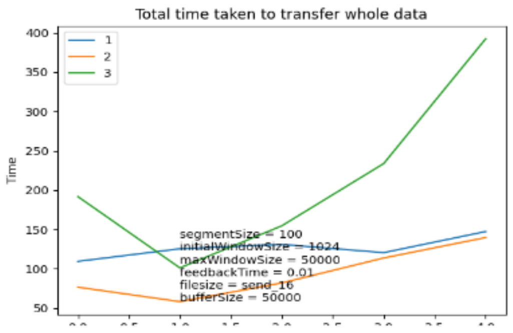
It is imperative to measure effects of different frequency of feedback from the receiver to the sender. The feedback mechanism ensures that the sender is constantly aware of the state of the system. (basically what is the next packet number the receiver is expecting).

However it should be noted that if the frequency of the timer is too large, the sender will not be updated regularly with the state of the receiver. And if the timer is too low, the receiver will be flooded with NACK/ACK packets from the sender which will in turn cause duplicate packet processing on the sender's end. Thus it is important to find a middleground

Note :: please check the parameters in the graph that are being used for evaluation





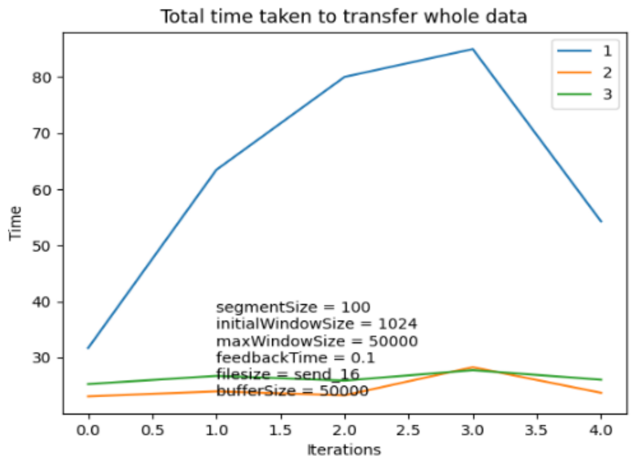


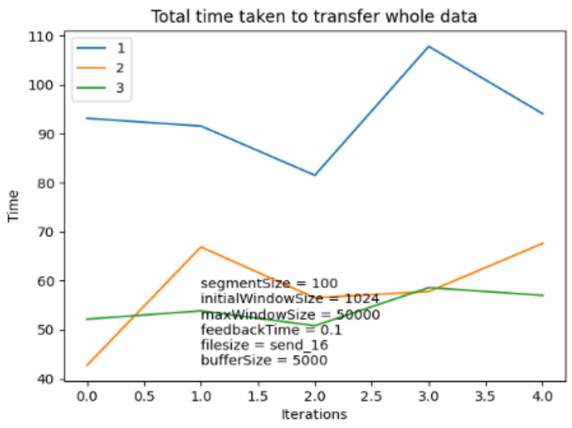
As seen in the image above the algorithm 2 best performs best i.e takes least transmission time when feedback timer is set to 0.1 seconds i.e 100 ms. Since we have set the delay of our system to be 5ms, and considering the processing on the host side, we can say that the RTT in our connection is 20ms. Thus we would need the feedback frequency to be greater than this to avoid flooding. Moreover , if we take the feedback timer to be 200 ms, The server is not able to update itself and preemptively send appropriate packets to the receiver

Consequence of the Buffer sizes::

As expected the increase in buffer sizes on the server sizes leads to better or decreased retransmission time because the buffer could simply store the achieved packets that are ahead of the current required packet. Once the current required packet is received, the receiver can club together the packets and write to the files.

However the increase in buffer size by a certain factor doesn't decrease the retransmission time by the same factor. This result leads to an inference that our hosts which are running on python programs are too slow to process the received data as compared to the rate of burst of the sent data.





Comparison ::

As seen in the graphs above, **Algorithm 2 performs considerably better from Algorithm 1** in terms of retransmission time. This is due to the fact that prevention of duplicate transmission prevents the flooding of the socket buffers on the receiver side. As per the inference drawn from the buffer sizes section, the processing of duplicate packets is the major overhead in the built system

Limitations ::

- Since we run our protocol on DATAGram sockets and process our data in python , our data processing is very very slow as compared to the TCP modules running in the kernel as it happens in the real world.
- Because of constant flooding of UDP packets in our networks , we ae not able to prove if this is a practical algorithm since in real world, the routers might detect constant flooding from a host and block the hosts from aggressively sending the data