CS3543: Computer Networks 2
Assignment 2 - Report

Team Members
1. Govind Balaji S (CS18BTECH11015)
2. S. Nishant Reddy (ES18BTECH11013)
3. Tatipelly Vamshi (ES18BTECH11021)
4. Sumadhva Sridhar (ES18BTECH11018)

Task 1: Preparation and Preliminary Study

1. Transferring 100MB of data using FTP without any externally applied packet loss or delay

Attempt_no:	Throughput (MB/s)	Time taken (100MB data)
1.	8.5376	11.71 secs
2.	10.6856	9.36 secs
3.	10.4614	9.56 secs
4.	10.5187	9.51 secs
5.	10.5897	9.44 secs
6.	10.2131	9.79 secs
7.	10.6420	9.40 secs
8.	10.1600	9.84 secs
9.	10.2661	9.74 secs
10.	9.5543	10.47 secs

Average Throughput: 10.16285 MB/s

2. Transferring 100MB of data using FTP with a 50ms delay and 5% packet loss in each direction

Attempt_no:	Throughput (kB/s)	Time taken (s)
1.	77.1150	1327.89 (~22min)
2.	75.5638	1355.15
3.	75.0108	1365.14
4.	75.4251	1357.64
5.	74.1796	1380.43
6.	76.7300	1334.55
7.	78.0555	1311.89
8.	76.0386	1346.68
9.	74.4985	1374.52
10.	77.9931	1312.94

Average Throughput: 76.061 kB/s

We see that upon adding delay and packet loss, the throughput of FTP went down by around 1/134th of what it was initially.

Task 2: Implementing "our-UDP-FTP"

Pictorial Representations of all Packet Formats

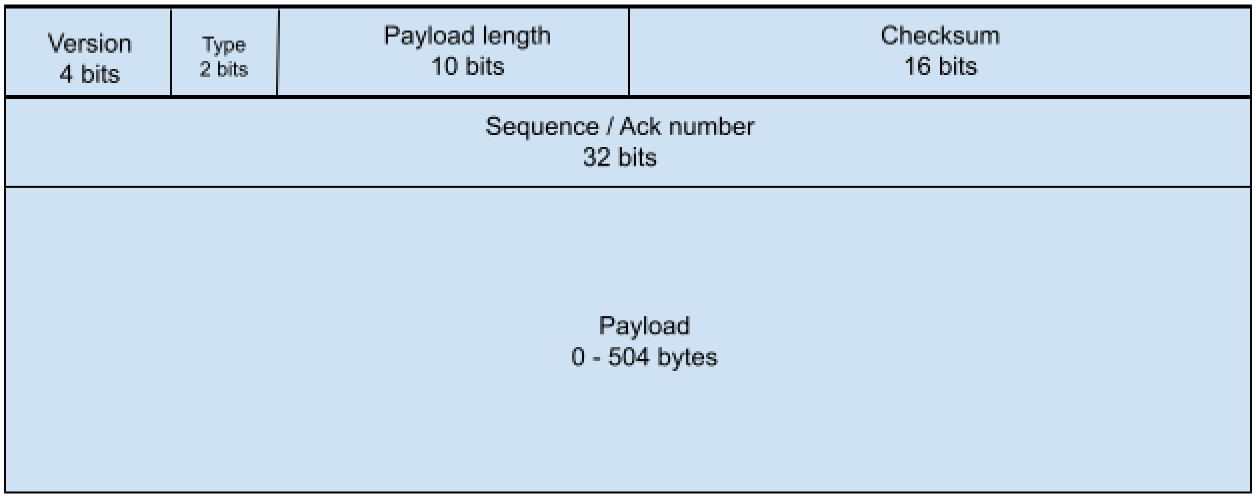
We use **sequence numbers instead of byte numbers** in the packets, since we need **fewer bits to encode sequence numbers**. This is possible because both sender and receiver will divide data into packets the same way. Thus, given a sequence number, each of them can calculate the byte number individually.

We **reuse the same field for sequence numbers that are being acknowledged by the receiver**. This is possible since the logical flow of data is unidirectional during a file transfer.

The 16bit checksum is the standard TCP checksum. We chose our packet length to be a maximum of 512 bytes so that it is deliverable by any router across the internet.

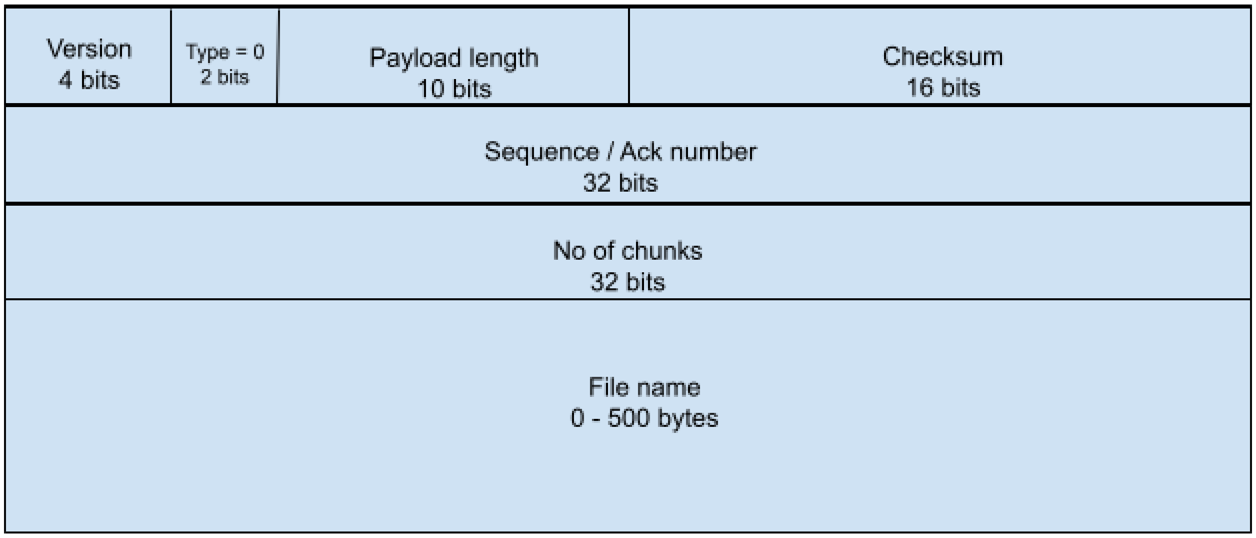
1. Data Packet / Data_Ack Packet:

This is the general format of data chunks sent from the sender to the receiver and data acknowledgment packets sent from the receiver to the sender.



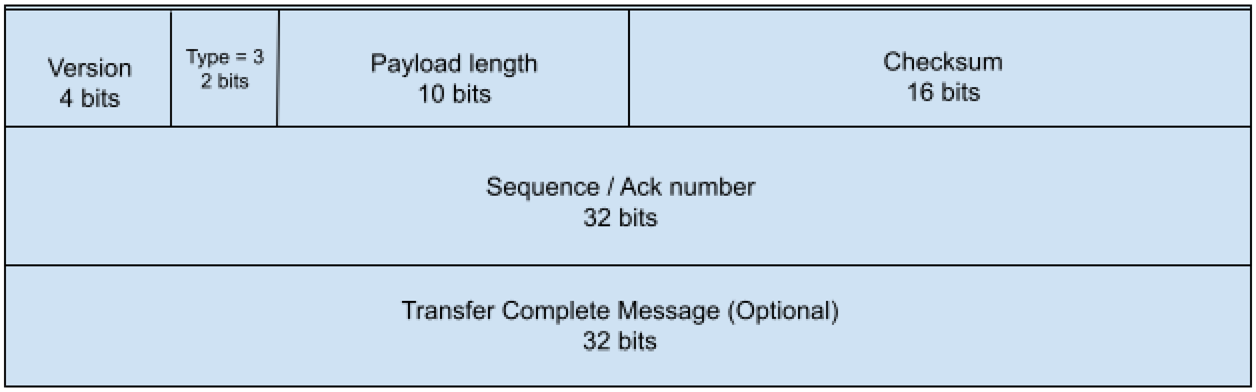
2. Metadata / Metadata_Ack Packets:

This is the general format of metadata packets sent from the sender to the receiver to perform and metadata acknowledgement packets sent from the receiver to the sender while performing the initial handshake (before sending any data).



3. Transfer Complete (Bye) Packets:

This is the general format of transfer complete packets that contain the “Bye” message. These packets are sent from the receiver to the sender once all the data chunks have been successfully received. These packets determine when the sender terminates.

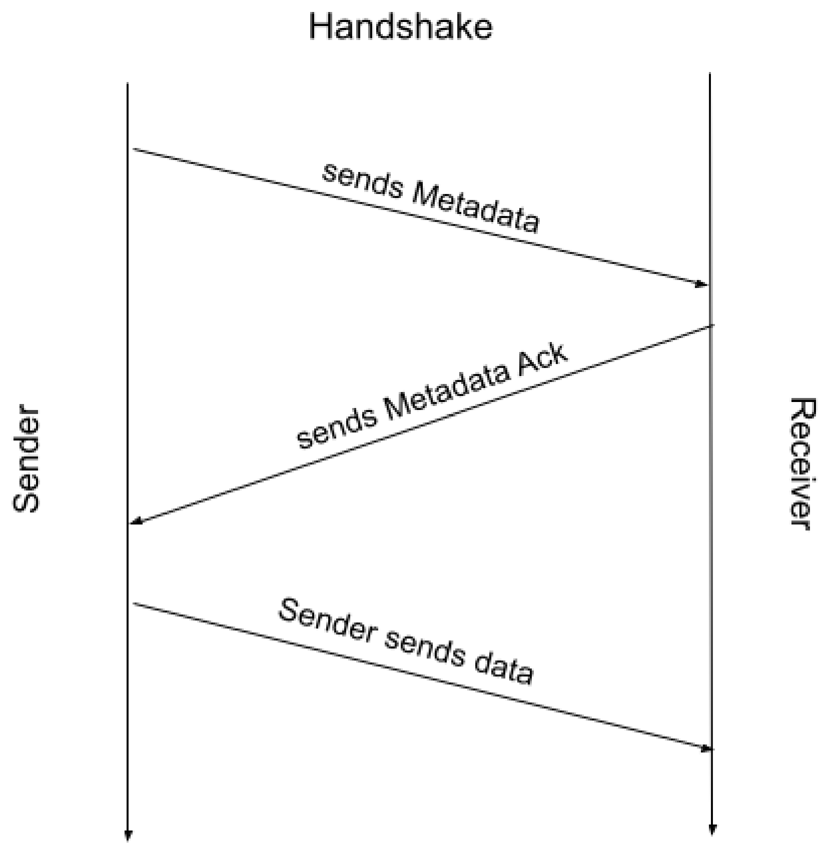


Features of “our-UDP-FTP”:

Handshake:

The **three-way handshake** between the sender and receiver is initiated by Host A (the sender). It sends a **Metadata packet that contains information such as the name of the file being sent, the number of data chunks** that the file is split into, and a sequence number (used for acknowledgement) to the receiver. After receiving this Metadata packet, Host B (the receiver) performs checksum verification on the packet and then sends a **Metadata_Ack packet** with the same sequence number back to the sender. Once the sender receives the Metadata_Ack packet from the receiver, **it starts sending the data chunks to the receiver.** Note that this is 3-way since, only the sender starts sending data after the first two-way handshake.

The pictorial representation of the handshake is shown below.



Reliable Data Transfer (ACKs & Checksum):

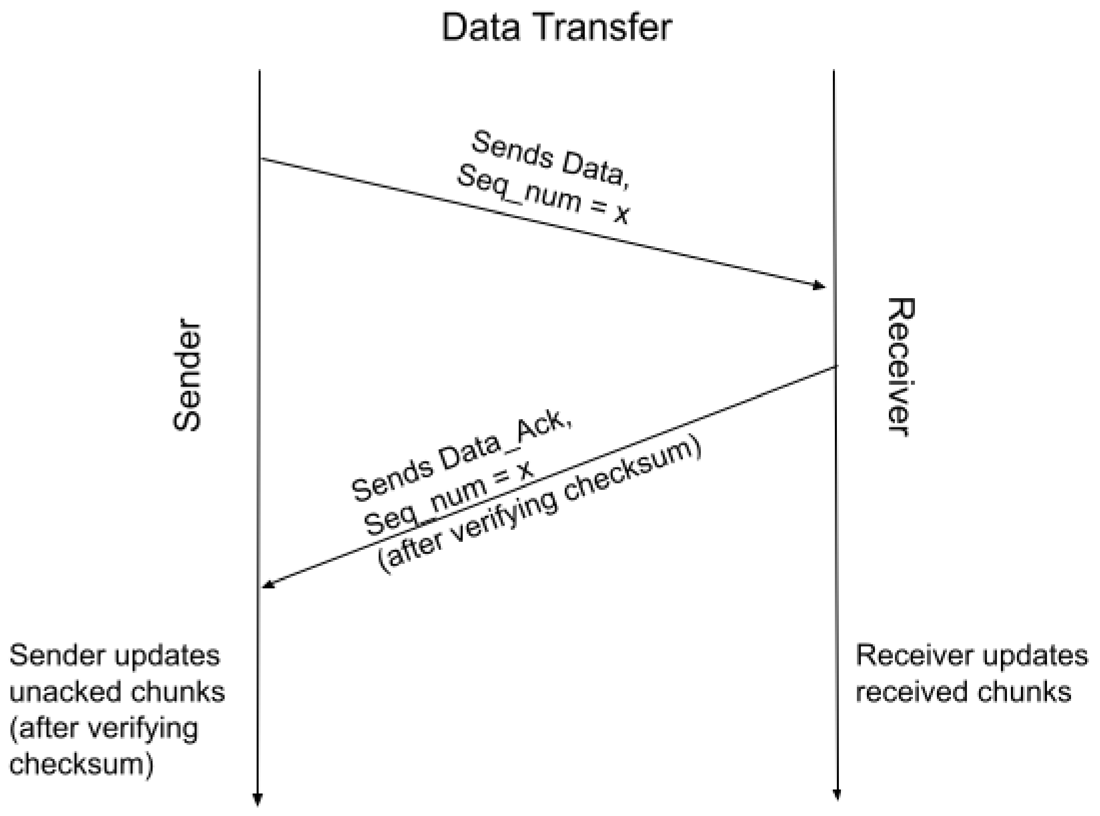
We implement Reliable Data Transfer in our-UDP-FTP primarily using **acknowledgement packets and checksum verifications** at both the sender's end and the receiver's end.

Checksum verifications are performed on all messages sent through the network. All packets have checksum fields that contain the checksums of their payloads as calculated at the sending host. The checksums of the payloads of packets are also calculated at the receiving host. Data packets and Ack packets are considered to be valid only if their calculated checksums are the same at both ends of the network.

The receiver sends acknowledgement (Data_Ack) packets to the sender corresponding to all data chunks that are successfully received without any checksum problems. The sender maintains a list of all packets that have been acknowledged by the receiver, and continues to send all unacknowledged packets to the receiver until all data chunks have been successfully received.

Since **there is no order in which the receiver prefers its chunks, it does not send the sequence number it wants in the ACK packet**. Instead, **only the sequence number which it received**, is sent.

The pictorial representation of data transfer is shown below.


Data Transfer
Sender
Receiver
Sends Data,
Seq_num = x
Sends Data_Ack,
Seq_num = x
(after verifying checksum)
Sender updates unacked chunks (after verifying checksum)
Receiver updates received chunks

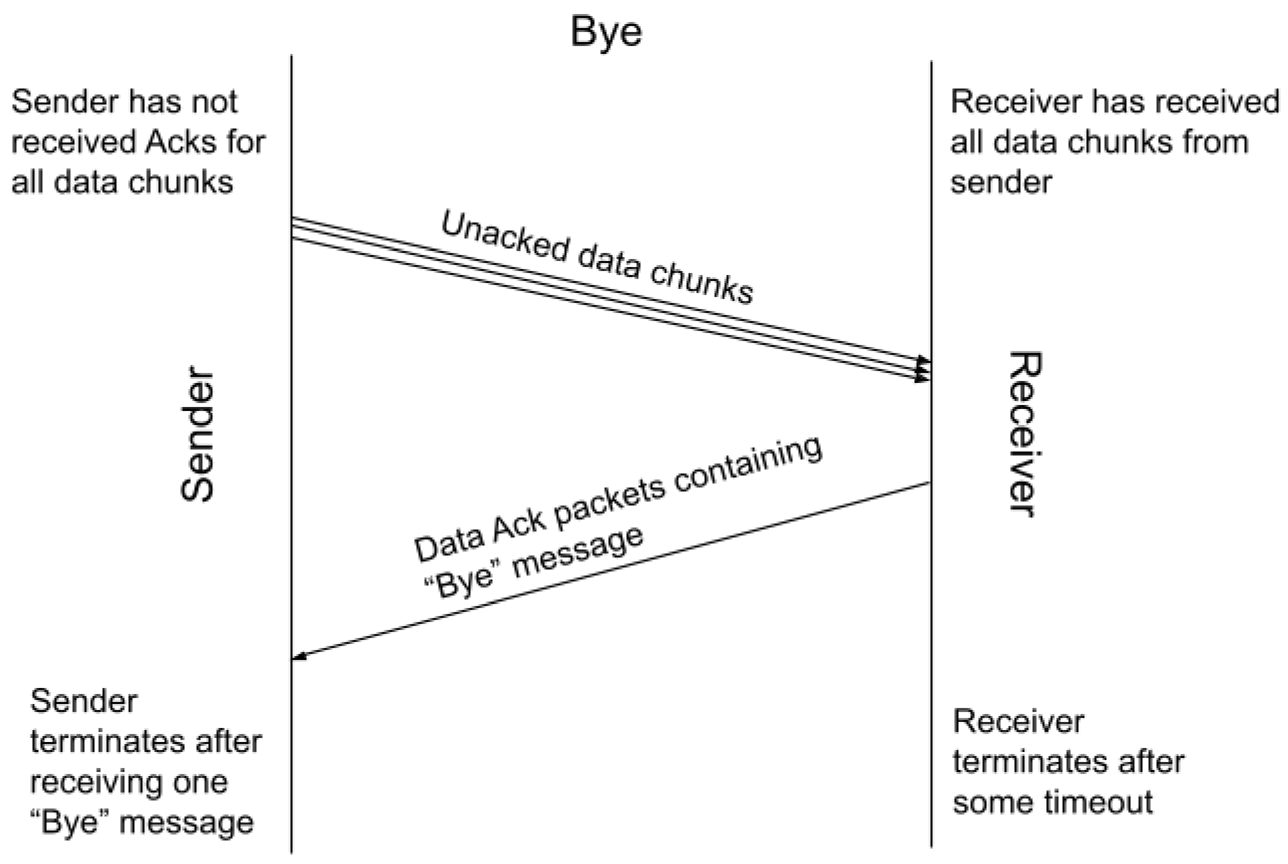
Closing the Connection:

The connection between the sender and receiver is closed after all the data chunks have been received at the receiver. This is done with the help of special **“Bye” messages** sent through the network.

The receiver sends Data_Ack packets to the sender corresponding to every single Data packet that it successfully receives. **Once the receiver has received all the data chunks from the sender, all Data_Ack packets corresponding to future Data packets from the sender are sent along with a special “Bye” message** stored in the payload of the Data_Ack packet. The receiver continues to send these special “Bye” packets for a specified amount of time and then terminates.

In addition to updating the unacked chunks, **the sender also checks each Data_Ack packet for this special “Bye” message**, As soon as it receives a verified Data_Ack packet that contains the “Bye” message in its payload, it knows that all the data chunks have been successfully received at the receiver, and immediately terminates.
In this way, **both the sender and receiver terminate after carefully verifying that the data has been sent successfully**.

The pictorial representation of the “Bye” procedure is shown below.


Bye
Sender has not received Acks for all data chunks
Receiver has received all data chunks from sender
Unacked data chunks
Sender
Receiver
Data Ack packets containing “Bye” message
Sender terminates after receiving one “Bye” message
Receiver terminates after some timeout

Flow Control:

For our application logic, the order of receiving chunks from UDP does not matter. We are good as long as we put together all chunks in the right order while saving the file. Also there is no point in delivering any partial file before the entire file contents. With these observations, our objective is as follows:

No matter in what order, or when the chunks reach our-UDP-FTP, wait till every chunk reaches and then put them together.

Therefore, we maintain a buffer as large as the entire file's size itself. It's equivalent to an infinite buffer. Thus **flow control is not applicable to our application**, under a good assumption that network delay is the bottleneck rather than the time to process received chunks.

There is a caveat however. Even though our receiver has infinite buffer at the application layer, the UDP socket by the OS will have a finite buffer. However starting from Linux 2.4, there is an auto tuning mechanism, that scales the buffers depending on the load automatically. With that, the infinite buffer assumption of our application makes sense.

Congestion Control:

We adapt the DAIMD (Decreasing Additive Increase and Multiplicative Decrease) - UDT congestion control algorithm as described by Yunhong Gu, Xinwei Hong, and Robert L. Grossman (2004 https://papers.rgrossman.com/proc-086.pdf).
We also combine this with the slow start technique as described by Yunhong Gu and Robert L. Grossman (2007 https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.113.5963&rep=rep1&type=pdf)

The idea is to follow **AIMD similar to TCP**. However, the additive increase is not constant. **When the sending rate is low, the increase rate is very high, and it decays down to 0 as the sending rate tends to infinity**.

Slow start is followed from the start of data transfer till we encounter either the first timeout or the maximum flow window size. Since flow control does not apply to our application, this size is taken as the total number of chunks in the file to be sent itself.

Difference in our implementation:

While the papers above describe a timer that fires a “sending event” for every time interval that is the inverse of the sending rate, we instead **relate the “sending rate” to the size of the “congestion window”**.

However, **the order in which the chunks within a file are sent is not important** as far as our application is concerned. We exploit this by **NOT requiring the “congestion window” to be contiguous**. Thus, the congestion window is **more like a congestion set**. However, we will refer to this as the congestion window only throughout this report.

Single Threaded Timeouts:

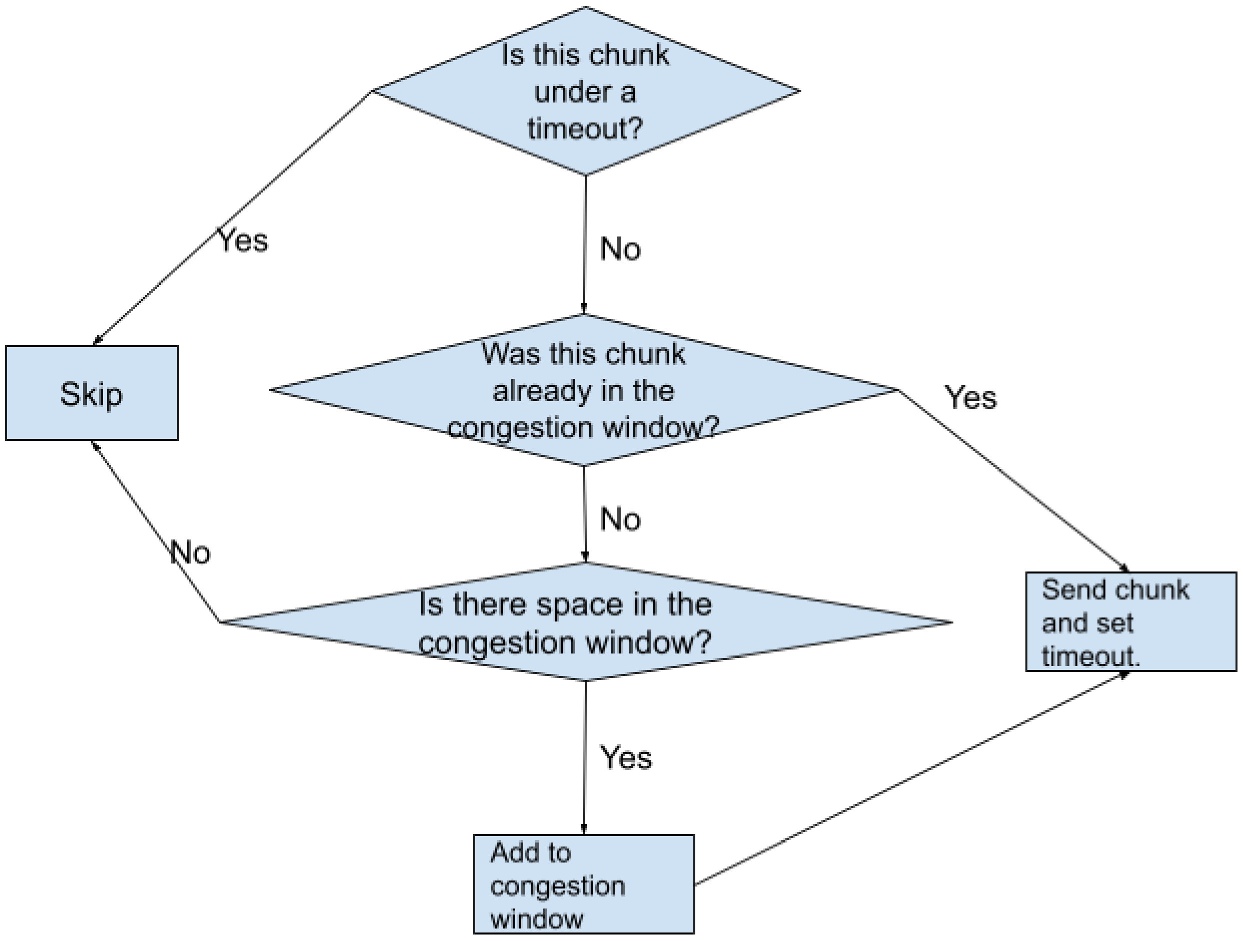
Our protocol does not involve any NAKs. Instead, the negative feedback for congestion control are timeouts. Similar to TCP, whenever a packet is sent, we wish to start a timer for that sequence number. We wish to retry sending only after the timeout ends. However, **when we attempted to create a callback-based-timeout for every chunk, the number of threads to be spawned was not scalable**. Our laptop began to slow down.

As a workaround, whenever we send a packet, we calculate the time at which it can be sent again. We know a packet’s timeout is over when we see current timestamp > sent timestamp + timeout. Thus it is enough to **have a single threaded sender looking at every unacked packet’s timestamps all the time. This greatly improved the performance**, compared to no timeouts at all as well as multithreaded timeouts.

The duration for timeouts is calculated based on RTT estimates with exponential moving averages, similar to TCP.

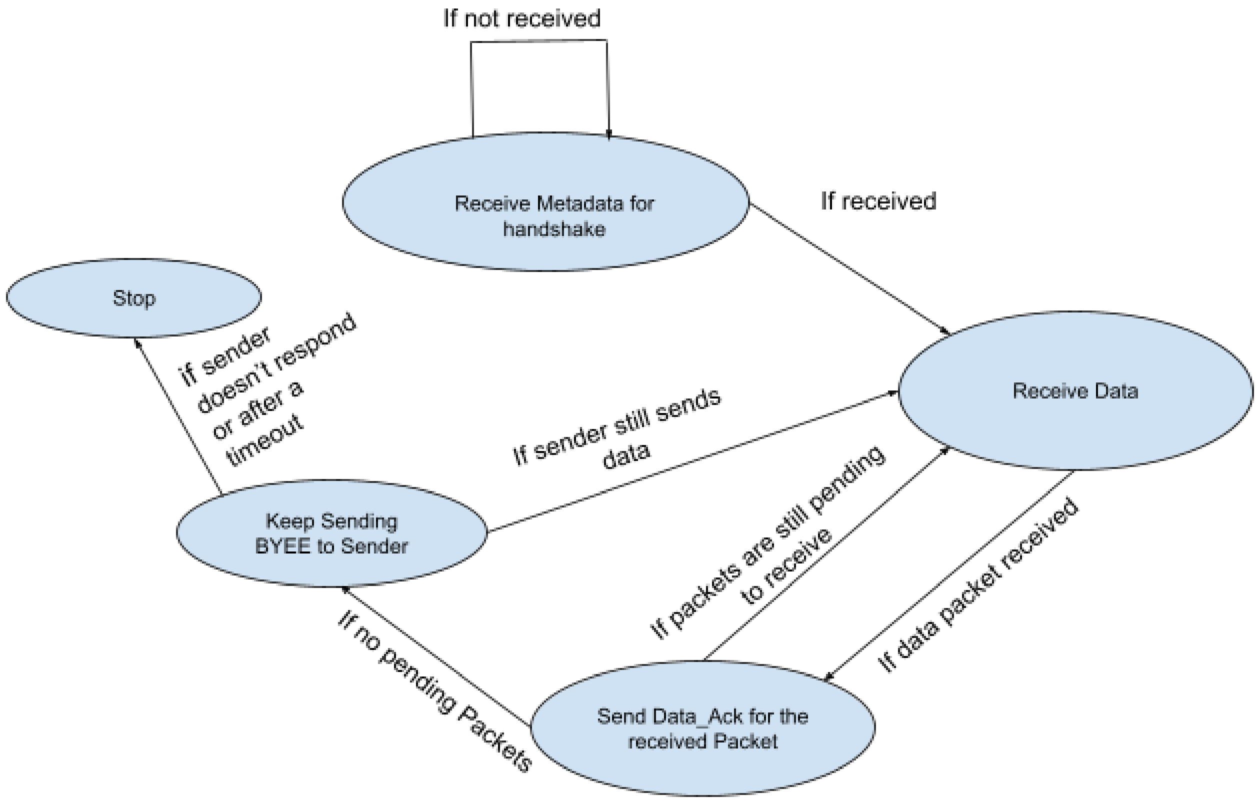
Putting together the Sender’s workflow:

The sender’s working can be described at a high level as follows:
While there are unacked chunks, the sender repeatedly finds some unacked chunk and does the following:


Is this chunk under a timeout?
Yes
No
Skip
Was this chunk already in the congestion window?
Yes
No
No
Is there space in the congestion window?
Send chunk and set timeout.
Yes
Add to congestion window

Receiver’s Workflow:

A high level description of the receiver’s workflow is given below.


If not received
Receive Metadata for handshake
If received
Stop
Receive Data
if sender doesn’t respond or after a timeout
If sender still sends data
Keep Sending BYEE to Sender
If packets are still pending to receive
If data packet received
If no pending Packets
Send Data_Ack for the received Packet

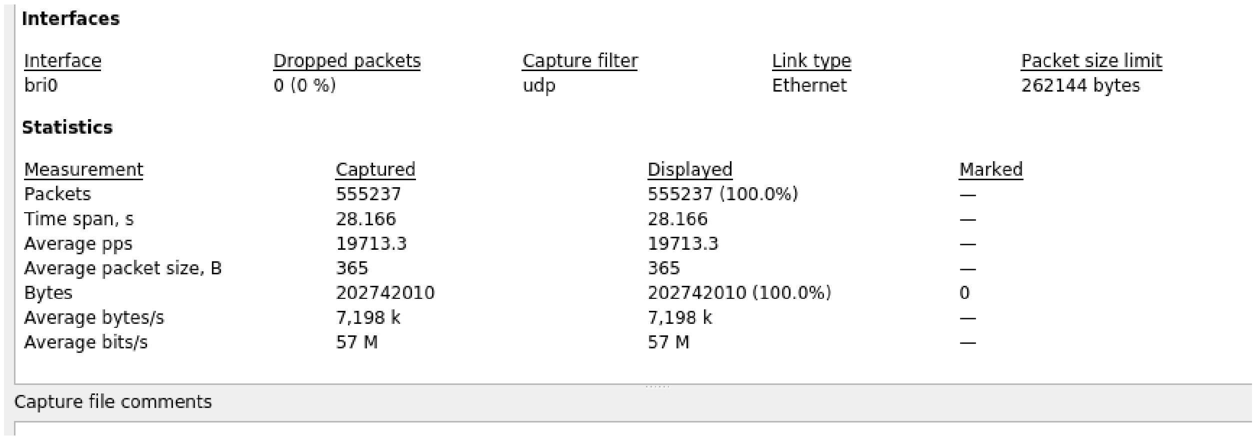
Measuring the Performance of the algorithm

(a) Without loss or delay on 100Mbps link:

Attempt_no:	Throughput (B/s)	Time taken (s)
1.	3.75443e+06	27.929
2.	3.83209e+06	27.363
3.	4.01046e+06	26.146
4.	3.55775e+06	29.473
5.	3.4766e+06	30.161
6.	3.75457e+06	27.928
7.	3.84347e+06	27.282
8.	3.8446e+06	27.274
9.	3.50882e+06	29.884
10.	3.38218e+06	31.003

Average throughput = 3,696,497 B/s

Wireshark capture of attempt #1:



```
Interfaces

Interface      Dropped packets    Capture filter    Link type    Packet size limit
bri0           0 (0 %)            udp               Ethernet     262144 bytes

Statistics

Measurement               Captured        Displayed                 Marked
Packets                   555237          555237 (100.0%)           —
Time span, s              28.166          28.166                    —
Average pps               19713.3         19713.3                   —
Average packet size, B    365             365                       —
Bytes                     202742010       202742010 (100.0%)        0
Average bytes/s           7,198 k         7,198 k                   —
Average bits/s            57 M            57 M                      —

Capture file comments
```

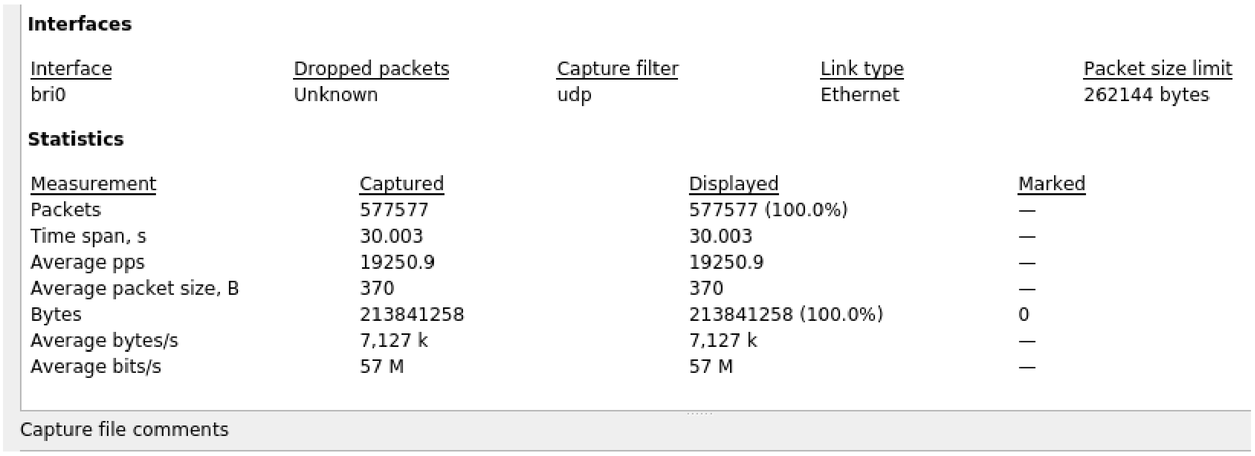
Note that the time taken is almost the same. (Raw bytes throughput in both directions is shown, and hence does not correspond to our application throughput)

(b) With 50ms delay and 5% Packet loss on 100Mbps link

Attempt_no:	Throughput (B/s)	Time taken (s)
1.	3.53663e+06	29.649
2.	3.37445e+06	31.074
3.	3.41934e+06	30.666
4.	3.67509e+06	28.532
5.	3.59619e+06	29.158
6.	3.60707e+06	29.07
7.	3.47544e+06	30.171
8.	3.11409e+06	33.672
9.	3.55884e+06	29.464
10.	3.37706e+06	31.05

Average throughput = 3,473,420 B/s

Wireshark capture of attempt #1:



Interfaces

Interface	Dropped packets	Capture filter	Link type	Packet size limit
bri0	Unknown	udp	Ethernet	262144 bytes

Statistics

Measurement	Captured	Displayed	Marked
Packets	577577	577577 (100.0%)	—
Time span, s	30.003	30.003	—
Average pps	19250.9	19250.9	—
Average packet size, B	370	370	—
Bytes	213841258	213841258 (100.0%)	0
Average bytes/s	7,127 k	7,127 k	—
Average bits/s	57 M	57 M	—

Capture file comments

Note that the time taken is almost the same. (Raw bytes throughput in both directions is shown, and hence will be much higher than our actual application throughput)

Analysis:

First, to confirm that ourUDPFTP transferred files reliably, we compare the “md5sum” of sent and received files. They were the same.

Next, we see that in case of no delay or packet FTP performs better than ours. This will be due to better implementation efficiency. However, when there is packet loss and delay, **ourUDPFTP beats FTP’s throughput by ~45 times**. This is due to better RDT features, that are more suitable for file transfer.

Also ourUDPFTP’s throughput blows up by only ~6% in case of delay + packet loss, much much lower than the disastrous blowup of FTP.