מטלה 3 - רשתות תקשורת

Table of Content

Background

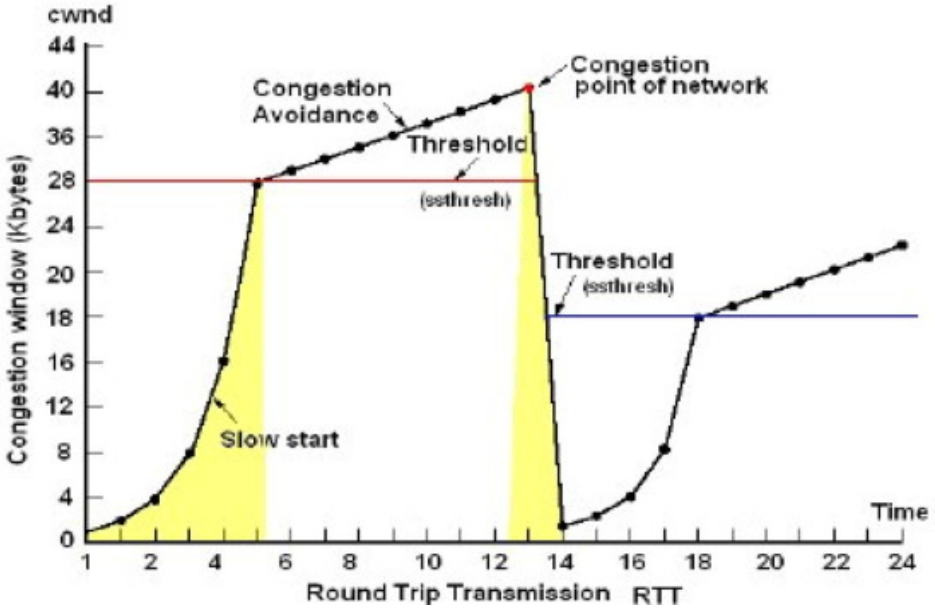
This exercise aims to implement two classes “Sender” and “Receiver” in Python using PyCharm. The main goal of this code is simulating a text file of 2MB transferring between a sender and client. This is achieved using congestion control algorithms: “Cubic” and “Reno” (learnt in class) and using the TCP as the Transport protocol of the packets .

Congestion control

TCP is a classic standard which enables application programs and computing devices to exchange messages over a network. The congestion control ensures efficient packet sending and prevents congestion (traffic) in the process and by that decreases the number of packets lost.

Reno

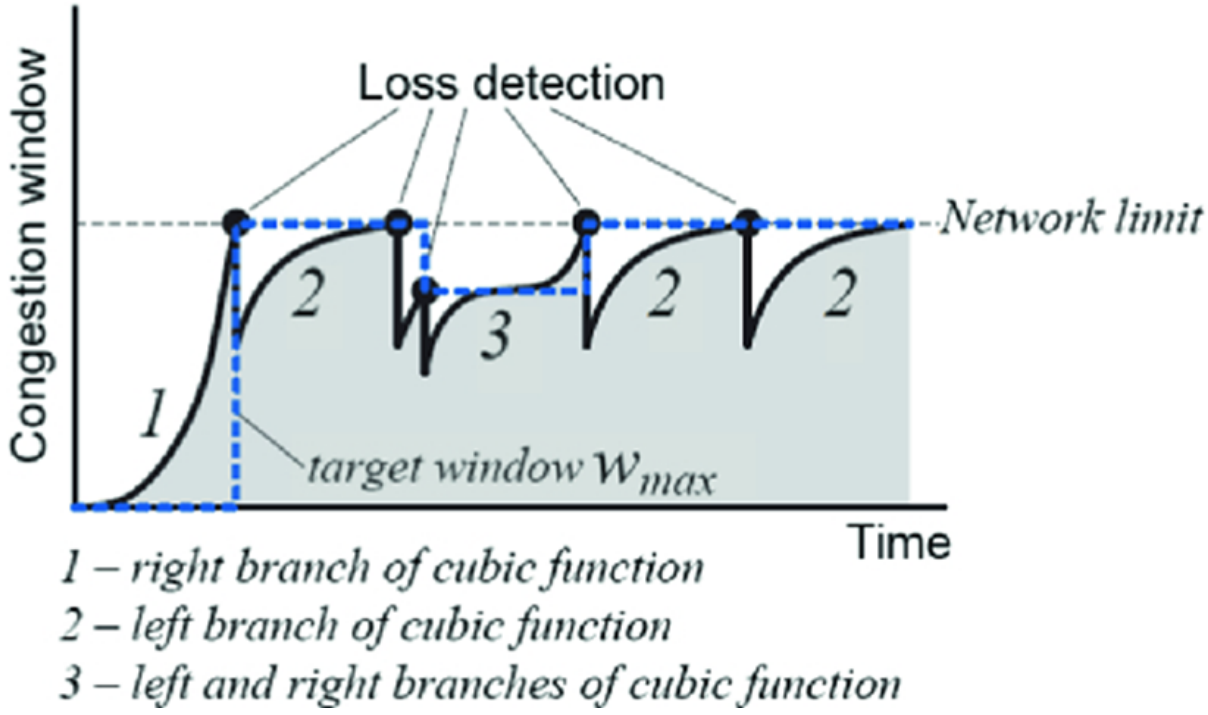
The Reno TCP implements the additive increase multiplicative decrease (AIMD) algorithm for congestion control.[1] This algorithm uses “slow start” which increases the size of the window by 1 at first (so its current size is 2), if it gets ACK which means the transfer of packets was successful it continues and increases the window’s size by 2(window size = 4). This process goes on and doubles the window’s size every time ACK is received (Slope #1 -slow start) in the image below[2] until it faces a threshold in which case it begins to increase the window linearly (Slope #2 - congestion avoidance). Finally when reaching a “problem” we halve the size of the current window.
[3]



[1] https://www.linkedin.com/advice/0/what-pros-cons-using-tcp-reno-vs-cubic
[2] https://media.geeksforgeeks.org/wp-content/uploads/20220202203124/Renocongestionwindow.png
[3] https://media.geeksforgeeks.org/wp-content/uploads/20220202203124/Renocongestionwindow.png

Cubic

TCP Cubic is a newer TCP variant that is designed for high-speed networks with a high bandwidth-delay product.[4] The algorithm also uses ”Fast retransmission” which is a TCP extension that allows saving time when a packet is lost, the main difference is that instead of waiting the timeout (which could be a waste of time) the extension assumes that if there are 3 duplicate ACK, the packet was lost and it resends it. “Fast Recovery” is also used, which helps dealing with lost packets more quickly.
The way Cubic works is to initialize the window size to a small value, the window’s size increases by a cubic function (hence the name of the algorithm) the window increases until reaching a threshold, this is similar to the “slow start” in Reno and is marked in the following image as Slope 1, the only difference here is that instead of multiplying the window size by 2 we increase it by x^3 .
When there is load on the network (which is likely to cause congestion and loss of packets) the sending window needs to be reduced, as explained in Reno the window size halves.



[5]

[4] https://www.cs.princeton.edu/courses/archive/fall16/cos561/papers/Cubic08.pdf
[5]https://www.researchgate.net/publication/341995073/figure/fig1/AS:1019836431863809@1620159361369/Cwnd-behavior-in-CUBIC-TCP-14.png

The process

The sender class begins by opening and reading the text file (in binary) Then we declare “socket.socket(socket.AF_INET, socket.SOCK_STREAM)” which is used to create a TCP/IP socket, which can be used to establish a network connection between the sender and receiver, socket.AF_INET and socket.SOCK_STREAM is used to create the socket using the IPv4 protocol.

```
1 usage
def send_file(filename, host, port):
    file_size = os.path.getsize(filename)
    part_size = file_size // 2 if file_size % 2 == 0 else file_size // 2 + 1
    with open(filename, 'rb') as f:
        data = f.read()
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((host, port))
```

After doing so the code continues with an infinite loop unless the sender wants to stop sending the file.Next we’re setting the cc algorithm using the s.setsockopt() which sets the socket options. Note the encode() function in the sender class which is necessary to send information between the two as when sending data over a network, it has to be in the form of bytes. After setting the congestion control algorithm to “cubic”, the code sends the first half of a file over the TCP connection using the sendall() function. Once the process is over we close the connection using socket.close() method.

```
    while True:
        print("Changing CC algorithm to cubic")
        s.setsockopt(socket.IPPROTO_TCP, socket.TCP_CONGESTION, "cubic".encode())
        # Send first half of file
        print("Sending the first half")
        s.sendall(data[:part_size])  # Changed from s.send() to s.sendall()
        print("First half size", part_size)  # Changed from temp to part_size
        print("Waiting for authentication")
        # Receive the authentication from the server
        auth = s.recv(5)
        if auth.decode() == str(ID1 ^ ID2):
            # Send second half of file
            print("Changing CC algorithm to Reno")
            s.setsockopt(socket.IPPROTO_TCP, socket.TCP_CONGESTION, "reno".encode())
            print("Sending the second half")
            s.sendall(data[part_size:])  # Changed from s.send() to s.sendall()
            print("Second half size", len(data[part_size:]))  # Changed from temp to len(data[part_size:])
        else:
            print("Connection was cut")
            s.close()
            break

        print("File sent.")
```

Next, we move into the receiver’s class where we use the bind() which assigns an IP address and a port number to a socket instance and listen() function in order to listen for incoming connections, once a connection is found it is accepted by the socket and returns a

connection as well as an address. The receiver receives the first part of the text using the recv() function and then writes it into the disk (in binary) and calculates the time it takes to receive that part of the text.

```
def receive_file(filename, host, port):
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        s.bind((host, port))
        s.listen()
        conn, addr = s.accept()
        while True:
            with open(filename, 'wb') as f:
                conn.setsockopt(socket.IPPROTO_TCP, socket.TCP_CONGESTION, "cubic".encode())
                # Receive first half of file
                start_time = time.time()
                data = b""
                while len(data.decode()) < part_size - 4:
                    chunk = conn.recv(2048)
                    if not chunk:
                        break
                    data += chunk
                f.write(data)
                end_time = time.time()
                print(f'Received first half in {end_time - start_time} seconds')
                time_pt1.append(end_time - start_time)
```

Note that the while loop goes on up until the length of the data is part_size-4, this is because we wanted to make sure we're not losing any data and would prefer having another chunk and not lose any bit.

Finally, an authentication (which is a xor between the two IDs) is sent to the sender just before the connection is closed.

```
# Send authentication:
auth_msg = str(ID1 ^ ID2).encode()
conn.sendall(auth_msg)
```

Moving back to the sender class, the authentication is received and being compared with the ID_1^ID_2 (XOR in python) to check if there isn't a match, we terminate the collection and placing a message to the screen, else, a socket is being set up, tries to connect to the receiver and sends the second and last part of the text file and terminates later. **We are then closing the connection to let the server know that the sender stopped sending data**.(The while loop in the receiver will break since)

```
# Receive the authentication from the server
auth = s.recv(5)
if auth.decode() == str(ID1 ^ ID2):
    # Send second half of file
    print("Changing CC algorithm to Reno")
    s.setsockopt(socket.IPPROTO_TCP, socket.TCP_CONGESTION, "reno".encode())
    print("Sending the second half")
    s.sendall(data[part_size:])  # Changed from s.send() to s.sendall()
    print("Second half size", len(data[part_size:]))  # Changed from temp to len(data[part_size:])
else:
    print("Connection was cut")
    s.close()
    break

print("File sent.")
```

Meanwhile, the receiver, just like in the first part tries to make a connection with the sender and receive the rest of the file, write it in binary into the disk and of course records the amount of time taken to receive that information. (Appends it to time_pt2 list)

```
    # Receive second half of file
    conn.setsockopt(socket.IPPROTO_TCP, socket.TCP_CONGESTION, "reno".encode())
    start_time = time.time()
    data = b""
    while len(data.decode()) < part_size - 4:
        chunk = conn.recv(2048)
        if not chunk:
            break
        data += chunk
    f.write(data)
    end_time = time.time()
    print(f'Received second half in {end_time - start_time} seconds')
    time_pt2.append(end_time - start_time)
```

Once the server finished processing both parts, it sends the client a message (“File sent”) letting it know another request can be sent (can send the text again).

Last but not least, a question appears on the screen asking whether we’d like to resend the file, if we do we must press ‘y’ and the file is being resent, once we’ve had enough we press ‘n’ and the socket is closed, the server gets a notification that the process is DONE!

```
    # Ask user if they want to send the file again
    check = s.recv(50).decode()
    if check == "again":
        send_again = input('Send file again? (y/n): ')
        if send_again.lower() != 'y':
            s.send("_done_sending_all_".encode())
            s.close()
            break
        else:
            s.send("_keep_sending_".encode())
```

```
conn.sendall("again".encode())

is_done = conn.recv(18).decode()
if is_done == "_done_sending_all_":
    conn.close()
    break
```

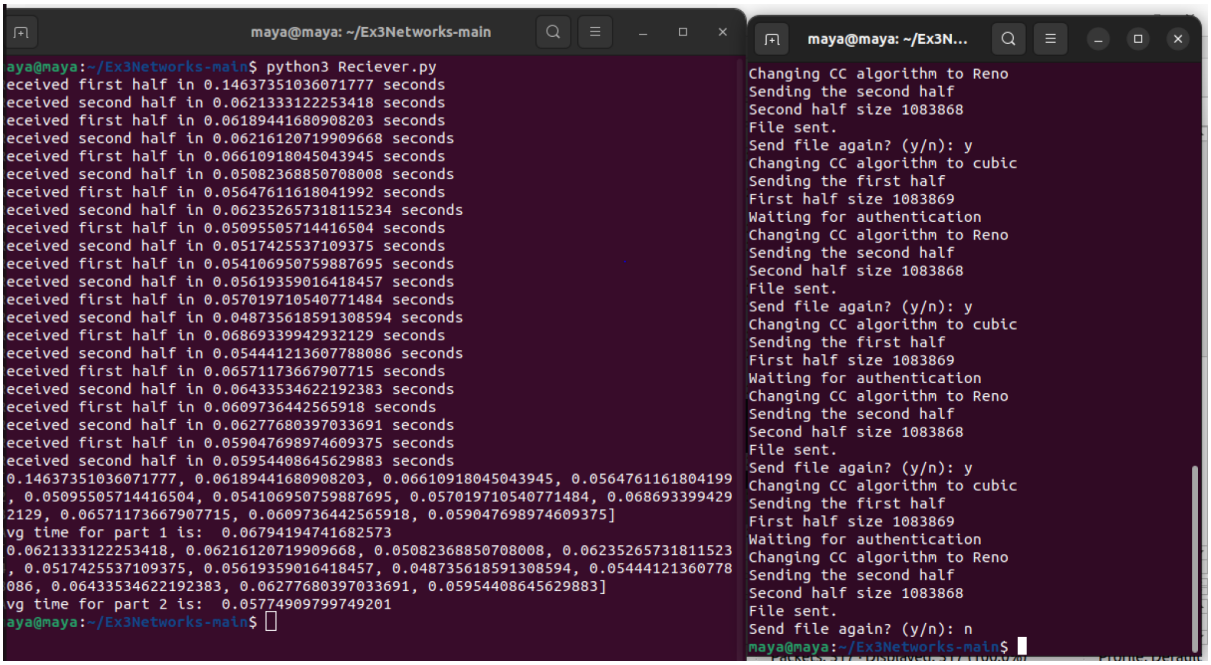
At the end of each run we get two messages printed, the first one is the time taken for the first half of the text to be received (using the Cubic algorithm), and the second one is the time taken for the second half (using Reno). At the end of the whole process (when we finally write ‘n’) we get a tuple with all the times the first half took and the average time as well as a tuple with all the second half’s times and of course the average time.

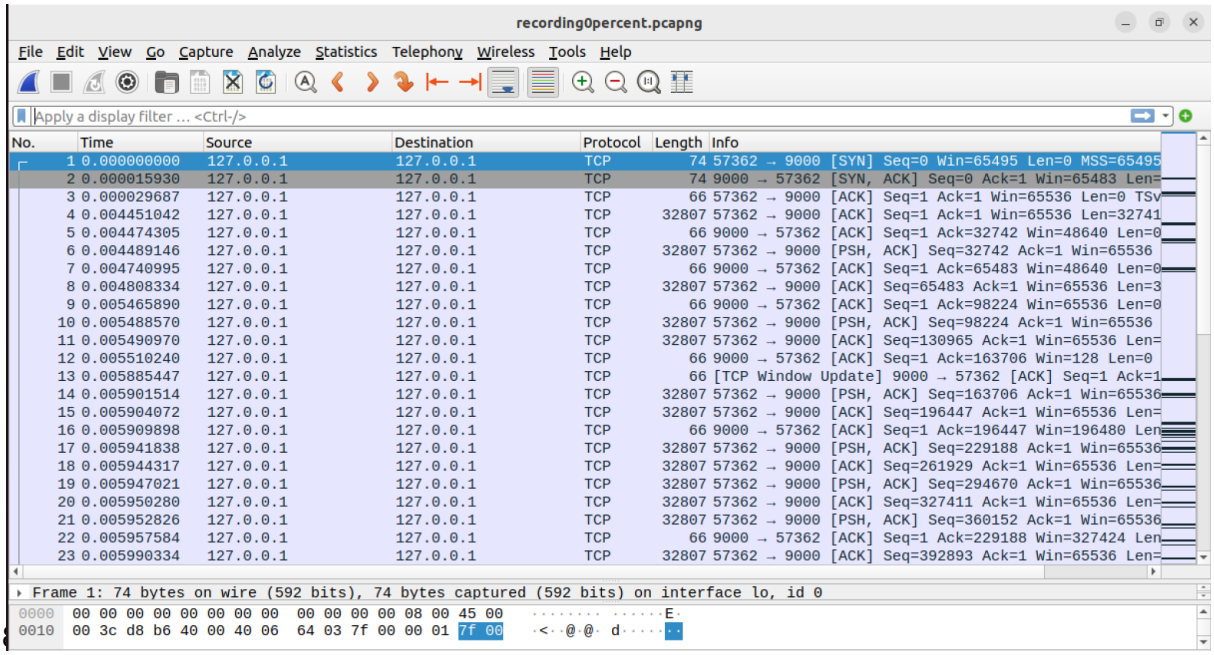
Running the program Packet Loss:

Note that if we get one the following: retransmission, fast transmission, dup ack, out of order messages when “standing” on one of the packets with the mouse, it means that the current packet was lost.

0% Packet Loss:

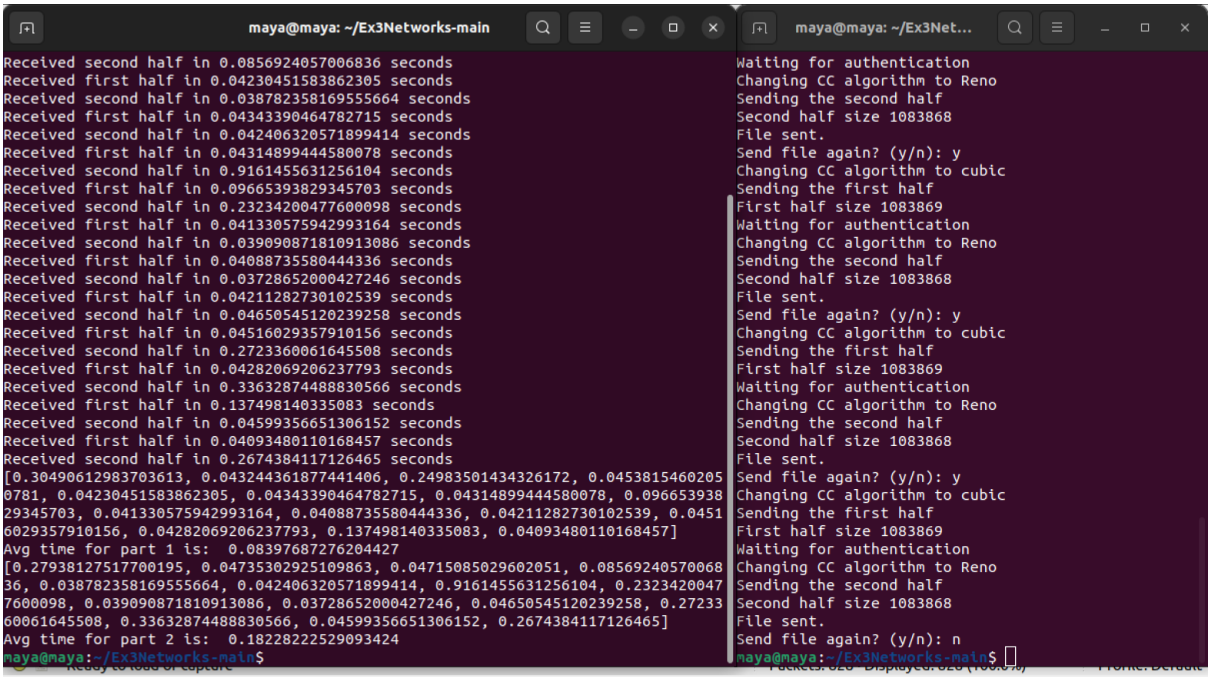
In the Sender.py we sent the file 11 times. The output is on the Reciever.py, with two Time lists, each list contains the times it took for the corresponding half of the file. We also calculated the average of each part’s time,note that the time difference between Reno and Cubic here is almost nonexistent:

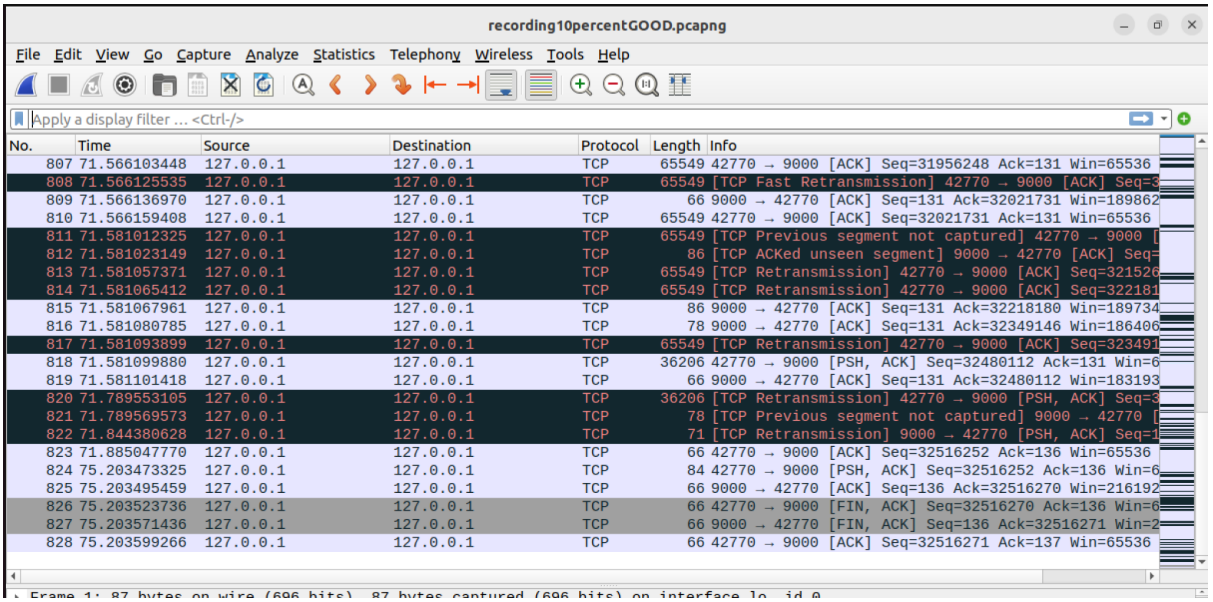




10% Packet Loss:

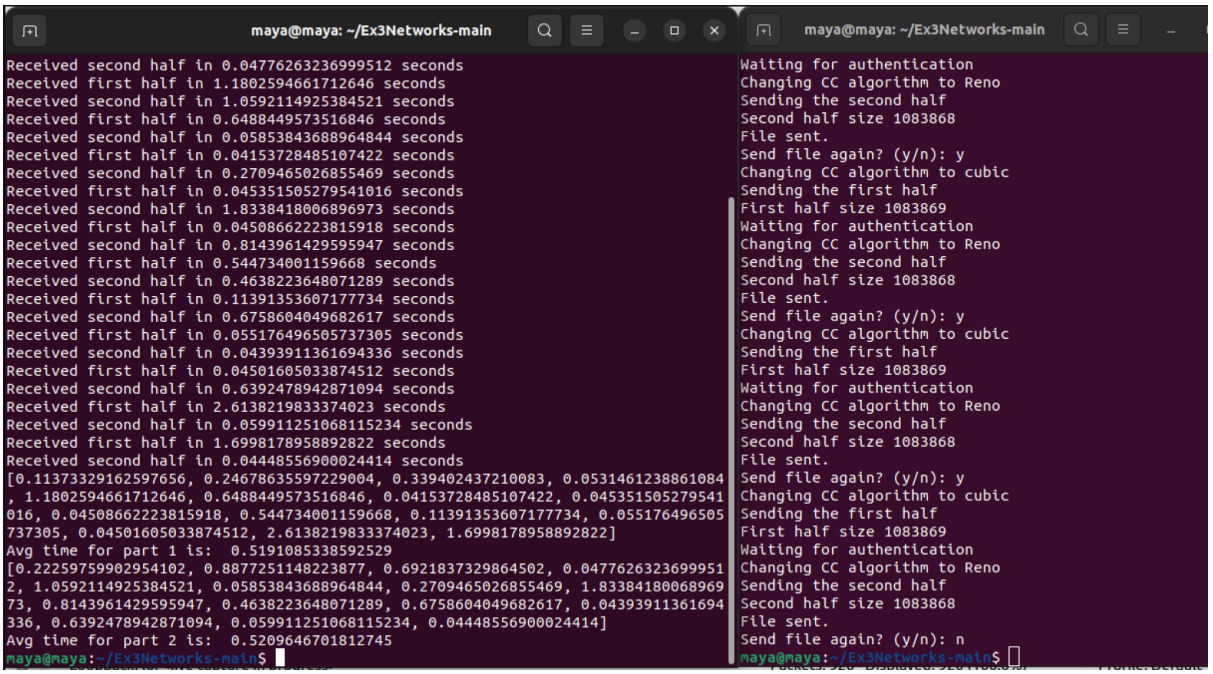
In the Sender.py we sent the file 11 times with 10% packet loss. The output is on the Reciever.py, with two Time lists, each list contains the times it took for the corresponding half of the file. We also calculated the average of each part's time. We can see in the wireshark's screenshot that the packets in black are lost, duplicate acks, out of order packets, etc.

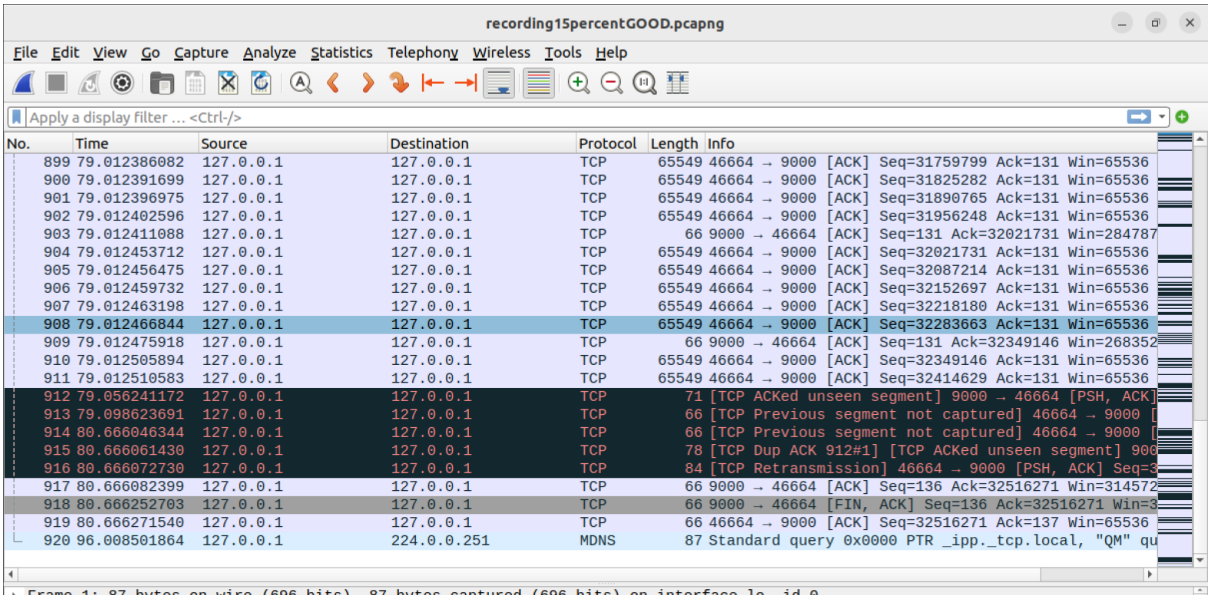




15% Packet Loss:

In the Sender.py we sent the file 11 times with 15% packet loss. The output is on the Reciever.py, with two Time lists, each list contains the times it took for the corresponding half of the file. We also calculated the average of each part's time, note that the average time is fairly similar (although the first part took less time to send) this could've been because internet connection, there were two time where the first part took more than 1 second to send which is an anomaly - increased the average time of the first half).

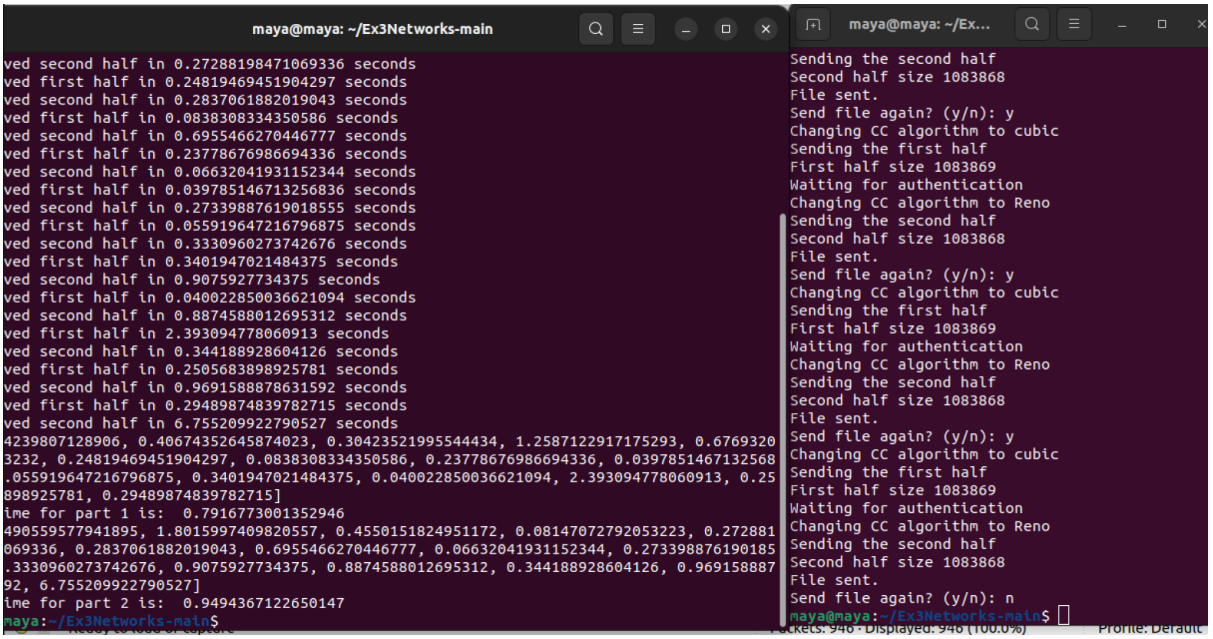


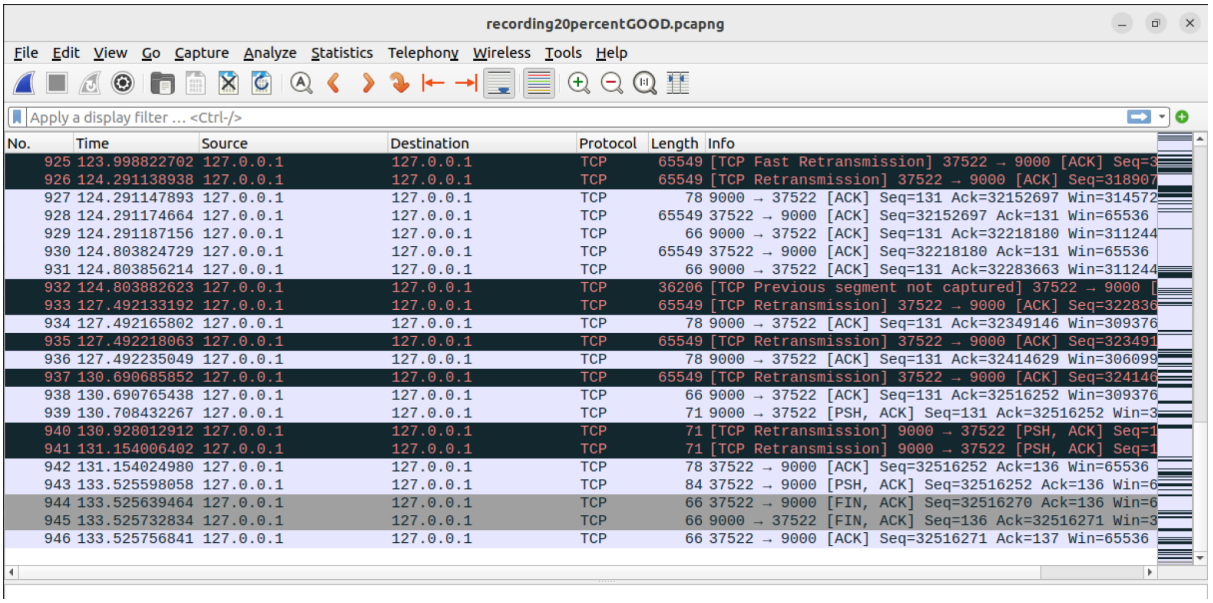


20% Packet Loss:

In the Sender.py we sent the file 11 times with 20% packet loss. The output is on the Reciever.py, with two Time lists, each list contains the times it took for the corresponding half of the file. We also calculated the average of each part's time:

Good:





Conclusion

Packet loss - transferring time

In the last part of the exercise we've tested our code on 4 different packet loss percentages: 0%, 10%, 15% and 20%.
Note that the first half of the text file is sent faster than the second part in all cases, another thing to notice is that the first half is sent using the Cubic algorithm and the second is sent using Reno. This makes sense as Cubic uses the function x^3 and Reno x after congestion, let us look at the difference between the two functions:

x	1	2	3	4	5	6	7
$f(x) = x^3$	1	8	27	64	125	126	343
$f(x) = x$	1	2	3	4	5	6	7

Note how much faster the cubic function grows rather than linear, this helps us make sense out of the results received above. The Cubic algorithm is known to be better for handling packet loss as it recovers faster than Reno as it uses a faster growing function.

Another important thing to notice is how the difference between the two grows as the packet loss percentage increases, in the 0% packet loss we have a tiny difference of 0.015 seconds, as the number of packets lost increased to 20% so did the time difference which became more than 4 seconds.