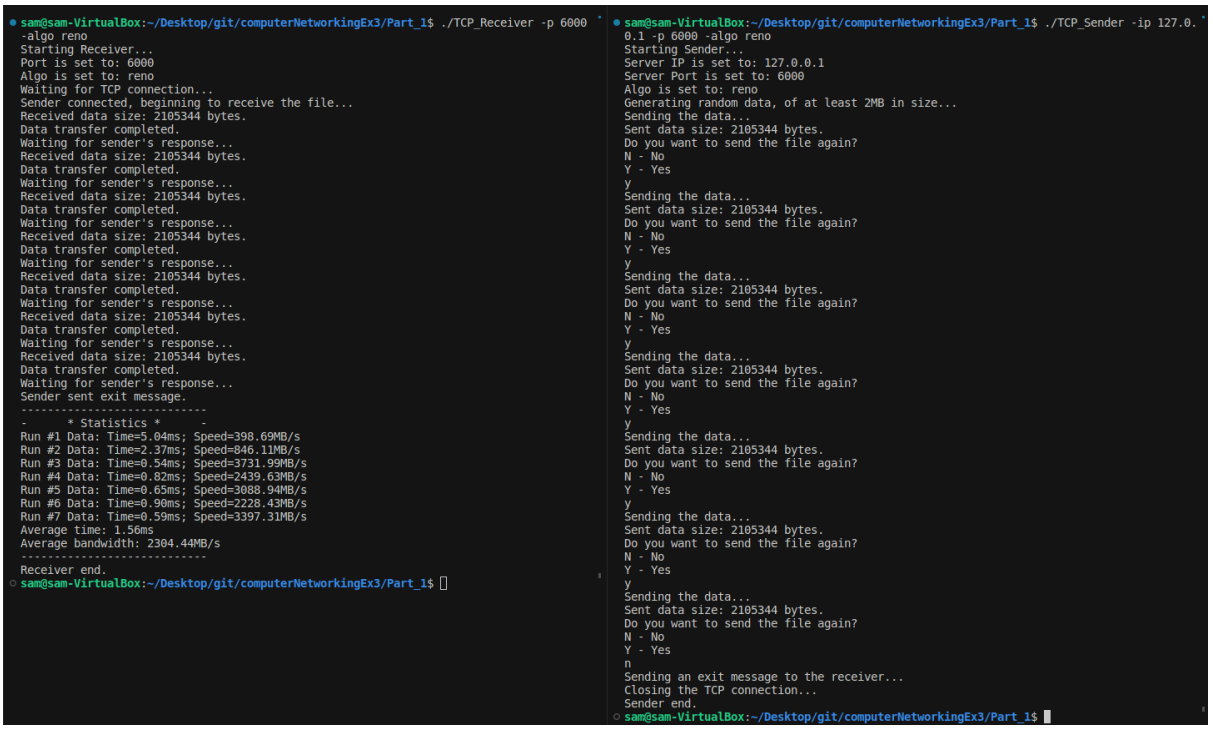
ID's: 5303621, 211791041

Ex3 - Communication Networks Course

*Our assignment support IPv4.

PART A

Example runs:



https://youtu.be/u_4_VYNsGDc
https://youtu.be/uD_roVgl8Z0

AND MANY MORE IN PART C, checking with packet loss.

Explanation:

Client side:

After creating the socket, we pull the server address and port, and the algo requested from the argv (`-ip <server_ip> -p <server_port> -algo <algo>`), we:

1. set the algo (reno or cubic) using
   `setsockopt(sock, IPPROTO_TCP, TCP_CONGESTION, algo, strlen(algo))`
2. set the server port using
   `server.sin_port = htons(atoi(argv[i]));`
3. set the server address using
   `inet_pton(AF_INET, argv[i], &(server.sin_addr))`

Then we `connect(...)` to the server, after the handshake, we establish a TCP connection.

We start by reading a file or generating random data using the given function
`char *data = util_generate_random_data(MIN_FILE_SIZE+BUFSIZ);`

Then, we send the number of total bytes we want to send, to prepare the receiver for all the bytes (and tell him to start the timer), that way he will keep receiving until the said number of bytes is reached.
The `send()` function gets all of the data, and is in charge of splitting into packets and sending everything, we don't need to split it by ourselves, like in PART B.
After sending all the data, we ask the sender if he wants to send again or not, and so on.

To close the connection, we send the receiver an EXIT MESSAGE by "preparing him" to receive 0 bytes instead of the data size.

Server side:
After creating the socket, we pull the server port, and the algo requested from the argv (`-p <server_port> -algo <algo>`), we:

4. set the algo (reno or cubic) using
   `setsockopt(sock, IPPROTO_TCP, TCP_CONGESTION, algo, strlen(algo))`
5. set the server port using
   `server.sin_port = htons(atoi(argv[i]));`
6. set the server address to INADDR_ANY to accept connections from any ip
   `server.sin_addr.s_addr = INADDR_ANY;`

Then we bind the server address and port to the socket using
`bind(sock, (struct sockaddr *)&server, sizeof(server))`

Listen to CLIENT connection at the same time (Allow waiting queue for connection to be CLIENTS)
`listen(sock, CLIENTS)`

Accept received connection and store client's socket id
`accept(sock, (struct sockaddr *)&client, (socklen_t*)&client_len);`

After the handshake, we establish a TCP connection.

We wait to receive the number of total bytes the sender wants to send.
After receiving it, we make sure it's not 0 (telling us he wants to close the connection) and we start receiving until the said number of bytes is reached.
And so on until we get the EXIT_MESSAGE.

To calculate the bandwidth, speed and time, we start a timer right after we get the number of bytes the sender wants to send, and stop it right after we get that amount of bytes.

EXIT_MESSAGE triggers printing the stats and closing the connection.

ID's: 5303621, 211791041

Part B

*Example runs can be found in PART C.

Compared to TCP, UDP is a faster protocol, with smaller overhead data, but it's not reliable.

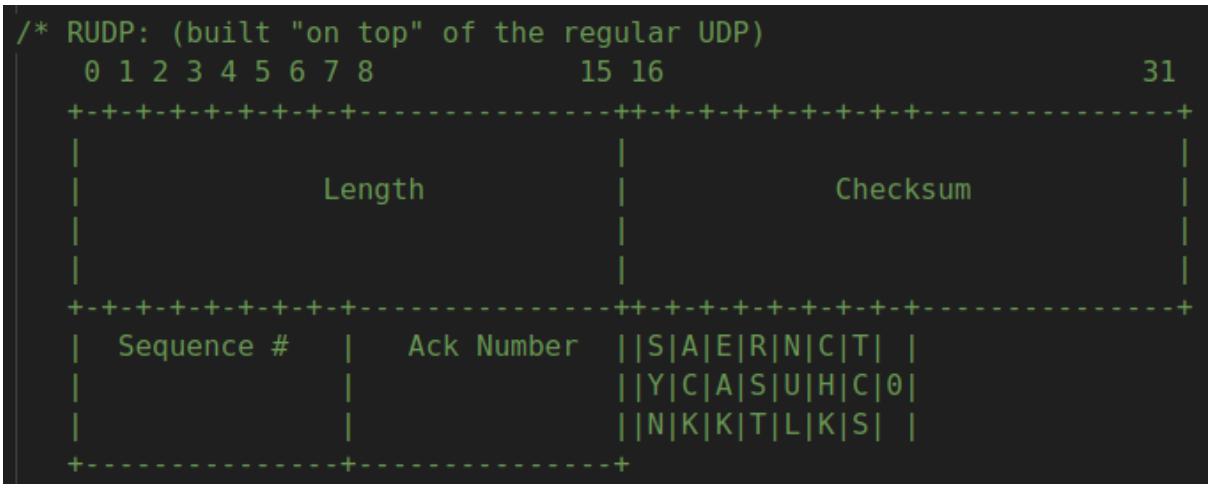
We implemented a Reliable UDP, by creating a new packet structure and header, built on top of the UDP header, with added features to improve reliability.

RUDP Implementation:

We started by designing a new packet and header. The header is built on top of the UDP header which contains the source port, destination port, length, and checksum.

Packet overhead: 7 octet (8*7 bits), could be reduced to 6 octets by reducing the `Length` to 1 octet (explained below).

Our RUDP header:



- **Length** - length of the data itself, without the RUDP header. Used to get the size of the data before receiving, to prepare the buffer, or when sending, to let the receiver know the size. (Technically, could be reduced to 8 bits because data is limited to RUDP_MAX_PACKET_SIZE minus sizeof header)
- **Checksum** - used to validate the correctness of the data. We decided to add a checksum field to our RUDP packet as well (even though UDP has it), to have control over its calculations and add another reliability layer. We can enable or disable the checksum that already exists in the UDP header, but we didn't touch it in this project.
- **Sequence & Ack #** - Both are used as one variable in the header. Used to organize the packets correctly when sending and receiving. When receiving a packet of seq x, the ack returns x+1, letting the sender know we are ready for seq x+1. This generally lets the receiver and sender know where they are in the order of packets.

- **Flags** - 1 byte unsigned int - used to classify the packet (SYN, ACK, etc.)

```
typedef struct _flags {
    unsigned int syn : 1;        // indicates a syn segment in present
    unsigned int ack : 1;        // indicates the ack num in the header is valid
    unsigned int eak : 1;        // not used
    unsigned int rst : 1;        // not used
    unsigned int nul : 1;        // indicates a null segment packet
    unsigned int chk : 1;        // 0 - @$ contains header. 1 - @$ contains header and data.
    unsigned int tcs : 1;        // not used
    unsigned int  : 1;     // not used
    } flags_bitfield;
```

Our RUDP packet:

```
rudp_packet_header header;
char data[RUDP_MAX_PACKET_SIZE - sizeof(rudp_packet_header)];
```

when RUDP_MAX_PACKET_SIZE is 576 (according to RFC 791, RFC 1122, RFC 2460)

When data is sent or received using this new protocol, the packet is built or disassembled using our RUDP API functions, while checking for the accuracy of the data and making sure it's received correctly.

Sending and receiving packets + how we **deal with packet loss**:

When sending a packet, we first build the packet according to our RUDP structure in 2 ways:

1. using the create_packet(...) function and manually setting the flags to syn/ack if needed.
2. using the helper private functions rudp_send_ack(...) and rudp_send_syn(...), that create a packet,flag it by itself and send it

Then we send the packet using the private helper function rudp_send_packet(...). This function deals with packet loss by waiting for an ack packet in return. If any of the following happens, we resend the packet, until limited `tries` is reached:

- Timeout is reached and we didn't receive any ack packet
- recv() function failed
- A packet was received but the syn/ack number doesn't match

That way, the receiver can just receive and send acks, and if something was wrong along the way, it won't send an ack, or the wrong ack will be received, or the ack might go missing, so our checks on the senders side, deal with all those problems.

ID's: 5303621, 211791041

How do we detect timeouts?
Using the select(sock_id + 1, &read_fds, NULL, NULL, &timeout) function inside rudp_send_packet(...), it basically gives us a file descriptor to read/write from/to the "socket" like a file (non-blocking, different from recv() or send()). If nothing was changed in the socket until the timeout, we resend the packet.

API Functions:

int **rudp_socket**(struct sockaddr_in *server_address, int peer_type, uint16_t * seq_number)

This function creates an RUDP socket and a handshake between two peers.
How it works:

1. It gets the peer type (CLIENT or SERVER) as an argument - telling us if the created socket is used as a server or a client
2. if it's for a server - we need to bind the server's address to the socket (received also as an argument - struct sockaddr_in*)

* Now socket is created (for client or server, depending on the peer_type)

3. Handshake:
    a. If server:
        i. Wait for SYN request from client
        ii. Receive it (with client's address)
        iii. Send ACK-SYN to that address to "establish connection"
    b. If client:
        i. Send SYN request to server
        ii. Wait for ACK-SYN

That way we have some sort of handshake between the peers.

rudp_send(...)
This function deals with a large file, so it "splits" the file into the required number of packets (according to the packet data size limit), creates each packet and sends them one by one (only after each receives an ack). The private function used here rudp_send_packet deals with the sending process itself (with the ack and timeouts).

rudp_recv(...)
The function receives the data, using the basic recv() func in UDP.
We keep receiving and sending ack for each packet, until we get a normal packet (not ack or syn), with correct checksum and matching seq number.
Then we stop receiving. (If the ack goes missing, the sender's timeout will end and he will just send us the packet again so we will send the ack again)

The other functions are straightforward, there are more explanations in the code.

ID’s: 5303621, 211791041

PACKET LOSS & SPECIAL FUNCTION

When testing packet loss in C, we noticed a very bad and non-consistent performance for different loss percentages.
After running some tests manually, we found those recommended values for best performance: (Big loss: the timeout should be as low as possible and max_retries as big as possible)

* 0% loss: 5 retries, 1 timeout_sec
* 2% loss: 100 retries, 1000 timeout_usec
* 5% loss: 400 retries, 100 timeout_usec
* 10% loss: 500 retries, 15 timeout_usec
* 50% loss: 1000 retries, 15 timeout_usec
* 75% loss: 10000 retries, 15 timeout_usec

We wanted to make it consistent, so we couldn’t use a specific value right from the beginning because we don’t know the packet loss rate at the beginning.
Introducing you: **loss_optimization(...)**
Right from the beginning of the run, we calculate the number of packets sent and acks received. For each packet sent, we call this function, that way we can calculate a rough loss rate and update the timeout and max_retries if needed.
The algo might be a bit slower in the first second, but the more packets sent, the better the calculation is and the better performance we get.
This function uses these variables to store info (top of RUDP_API.c)

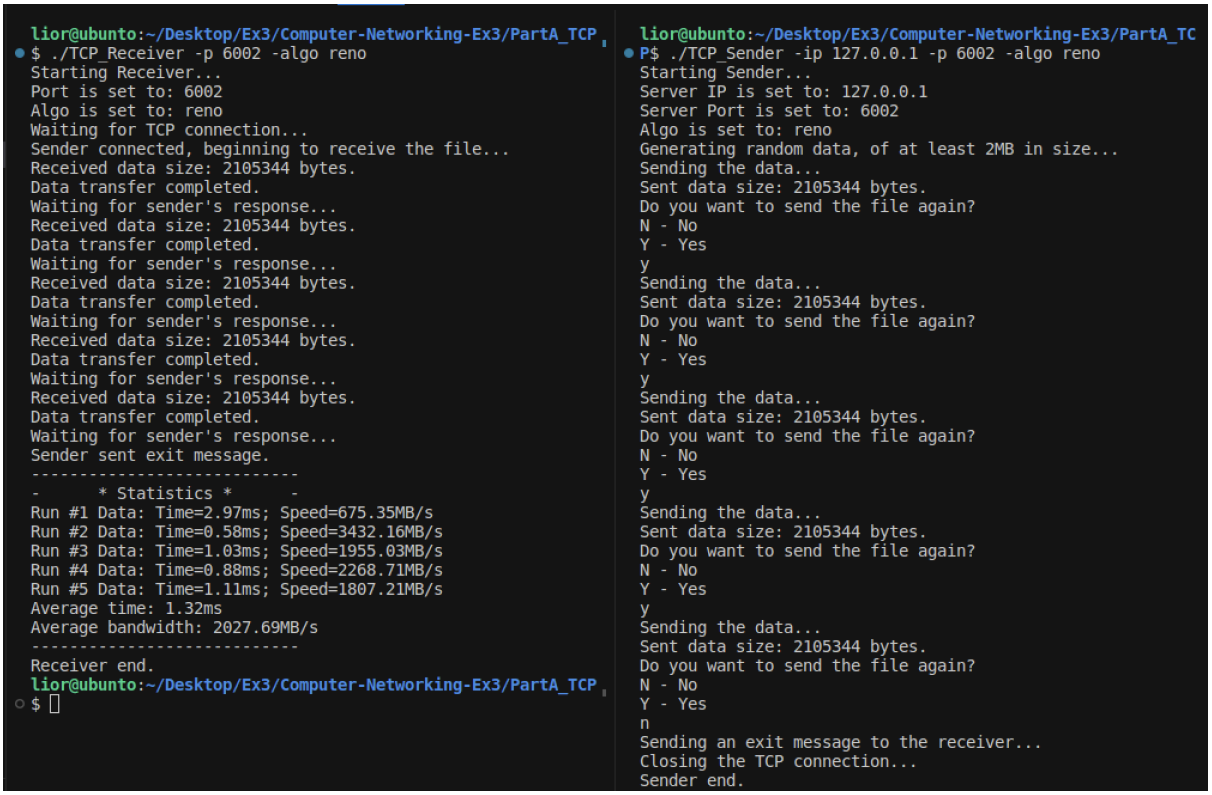
```
static int max_tries = 0;
int *b = &max_tries; /* global int pointer, pointing to global static, for RUDP_Sender.c */

// These are close to the best settings for 0% packet loss. if there is packet loss, the program will change those values to perform the best
static int MAX_RETRIES = 10000;
static int TIMEOUT_SEC = 0;
static int TIMEOUT_USEC = 1000;

// Used to calculate packet loss during the run and adjust timeout and retries
static int packets_sent = 0;
static int ack_received = 0;
```

ID’s: 5303621, 211791041

PART C

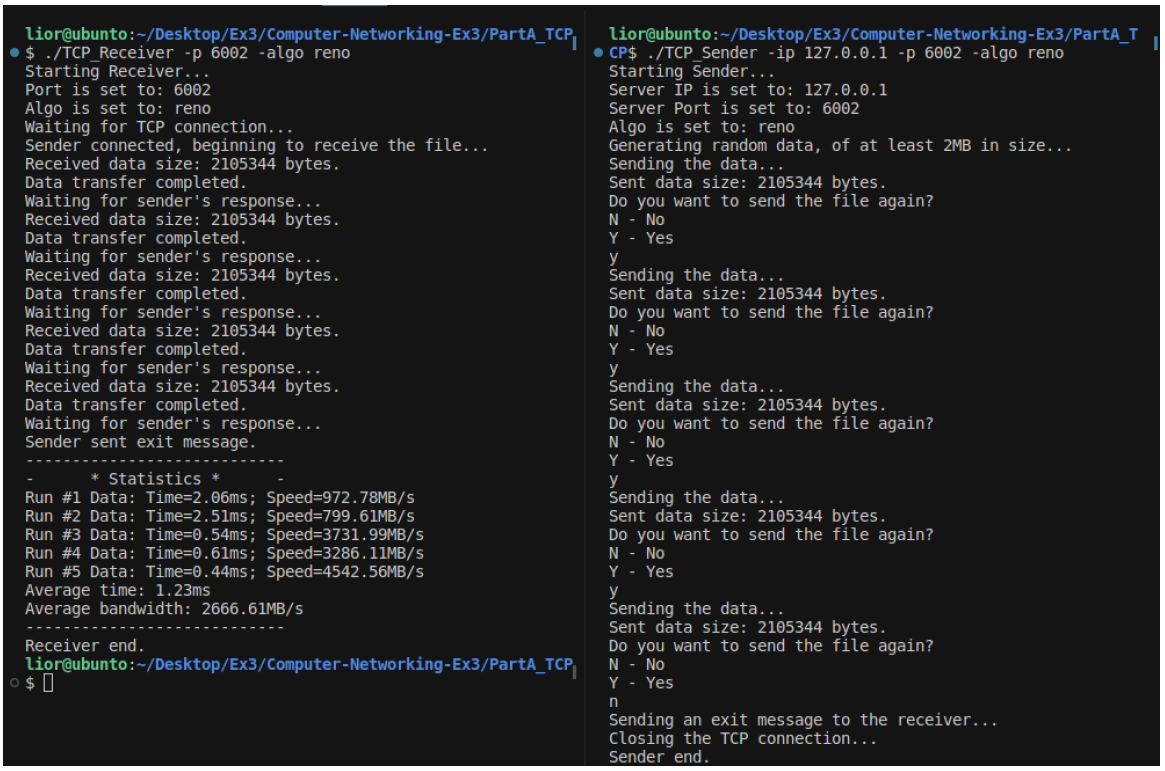
receiver: reno
sender: reno

0% packet loss:

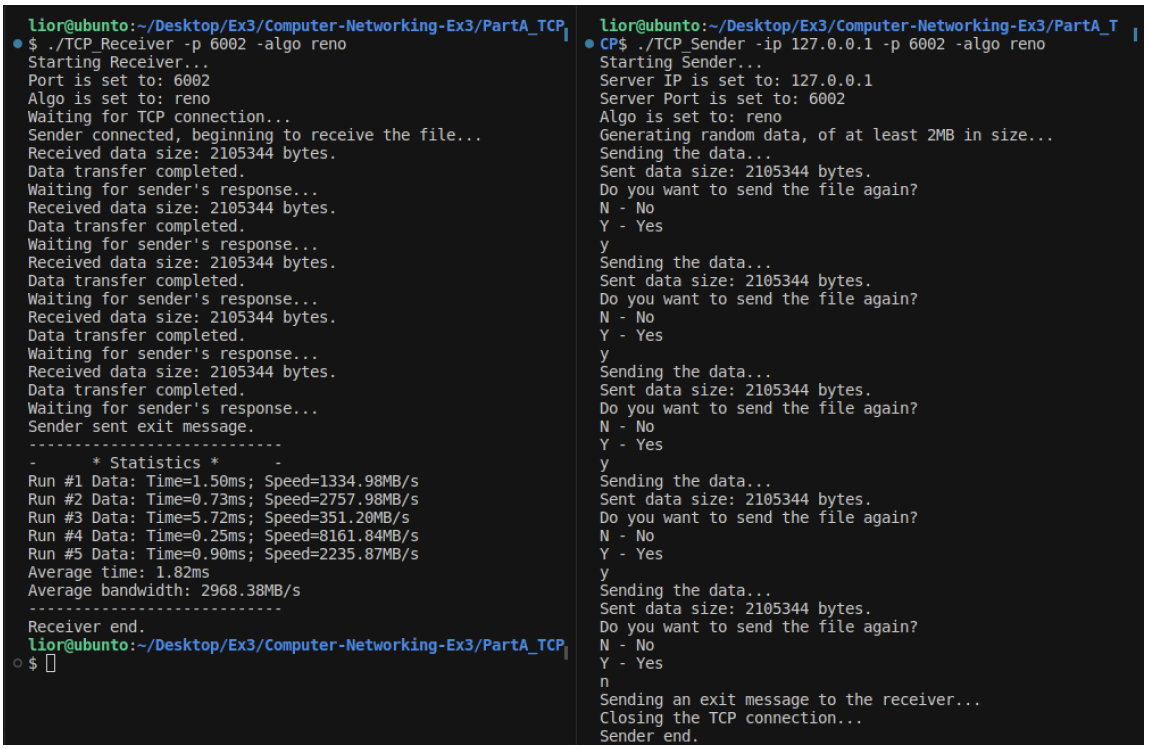


ID's: 5303621, 211791041

2% packet loss:

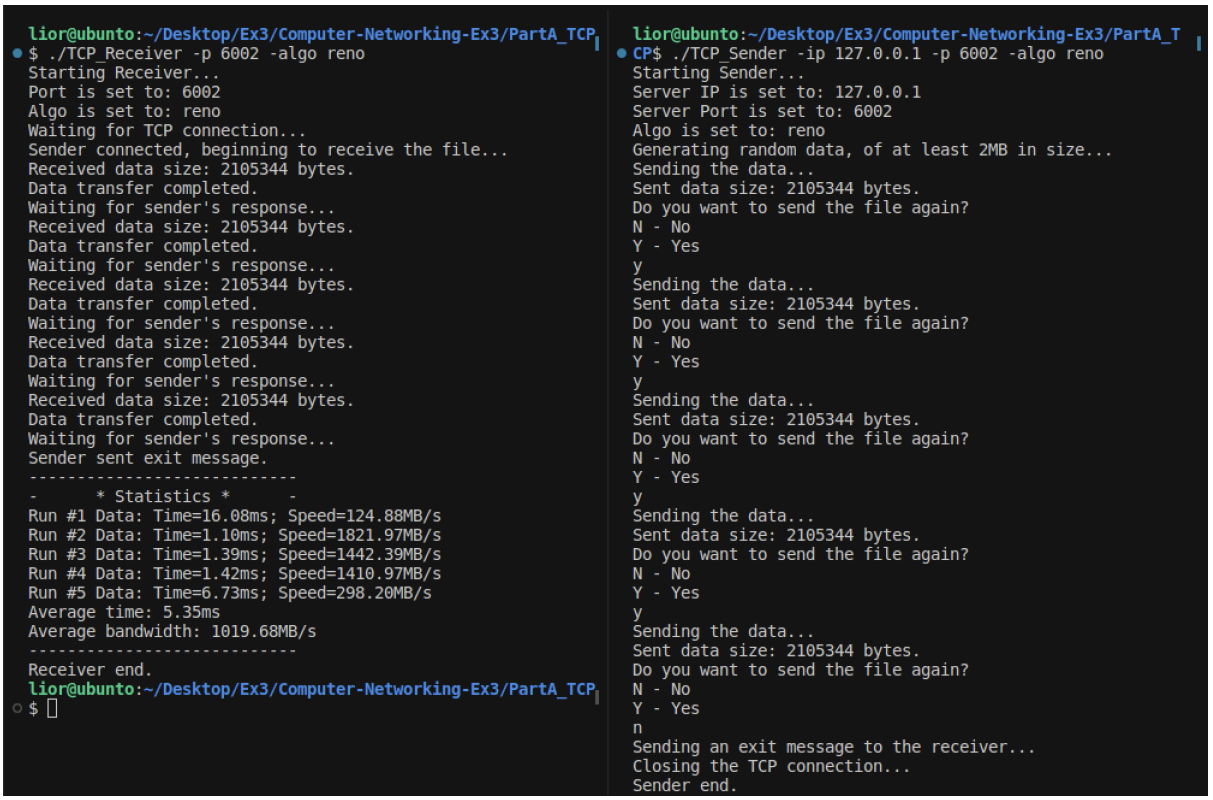


5% packet loss:



ID’s: 5303621, 211791041

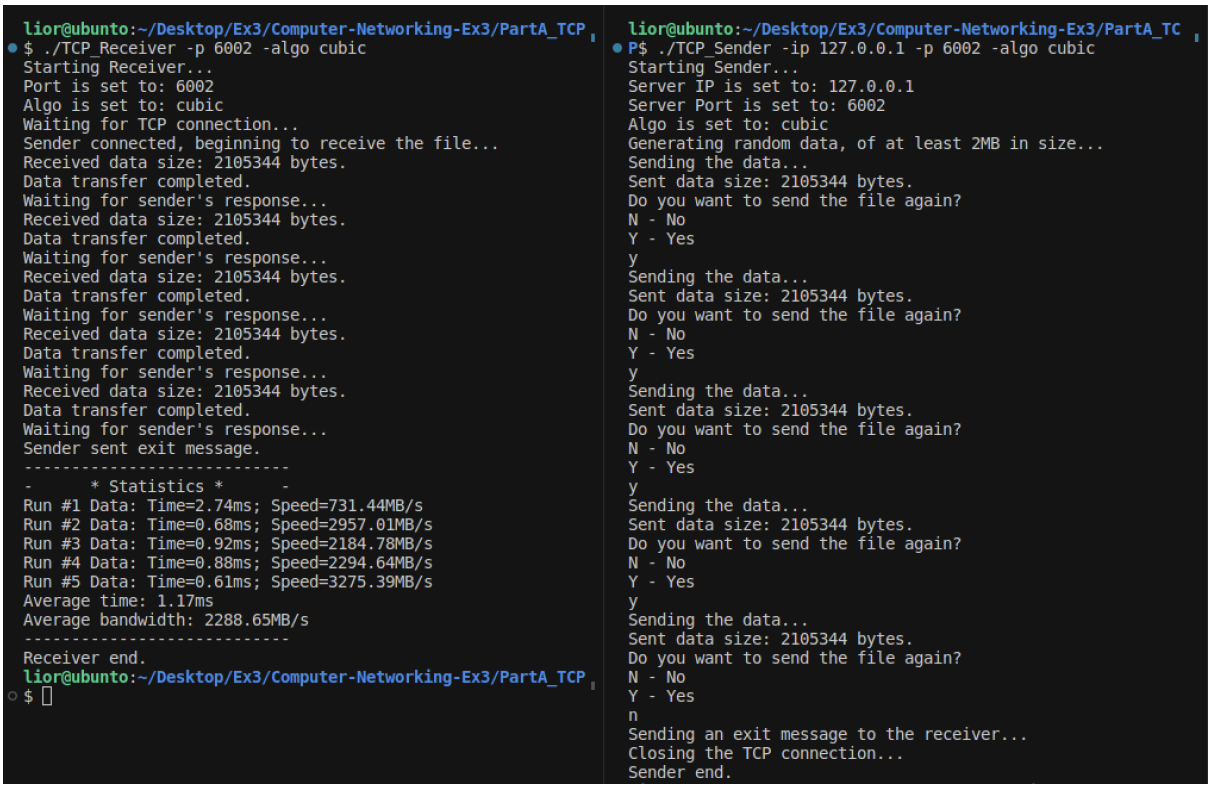
10% packet loss:



ID’s: 5303621, 211791041

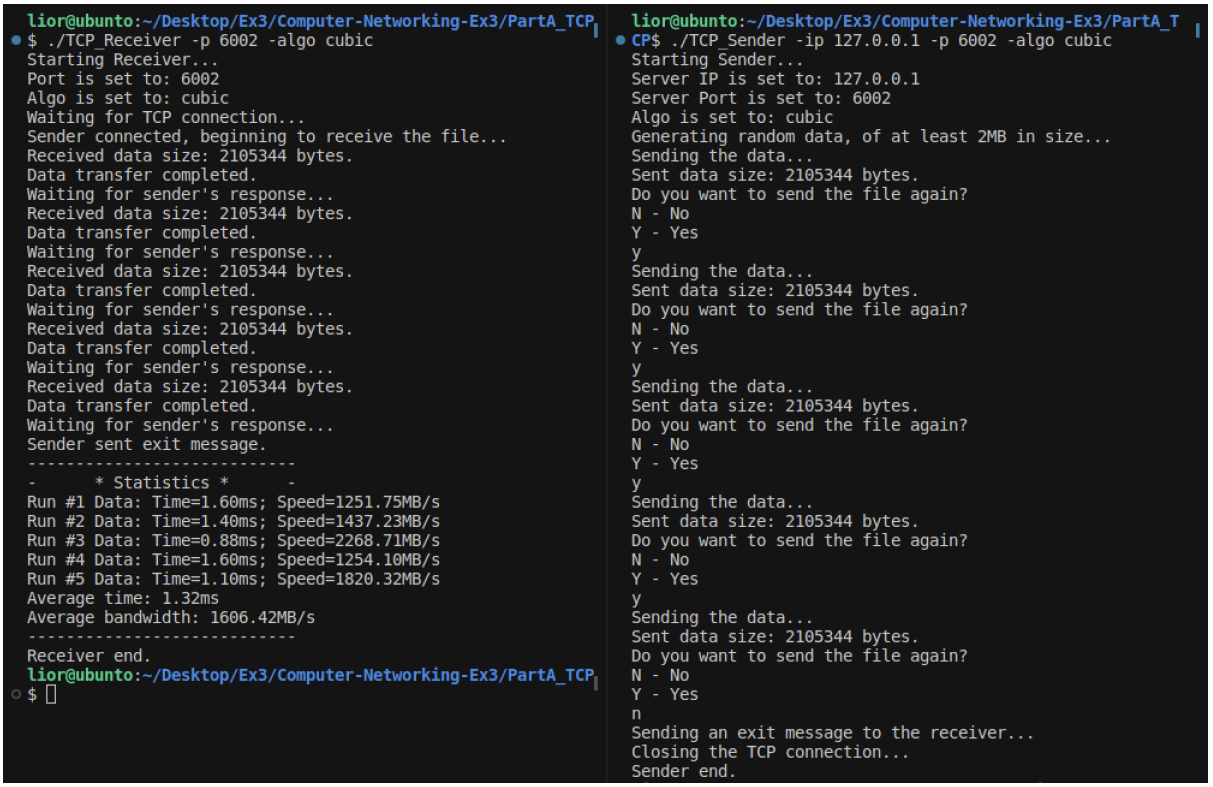
receiver: cubic
sender: cubic

0% packet loss:

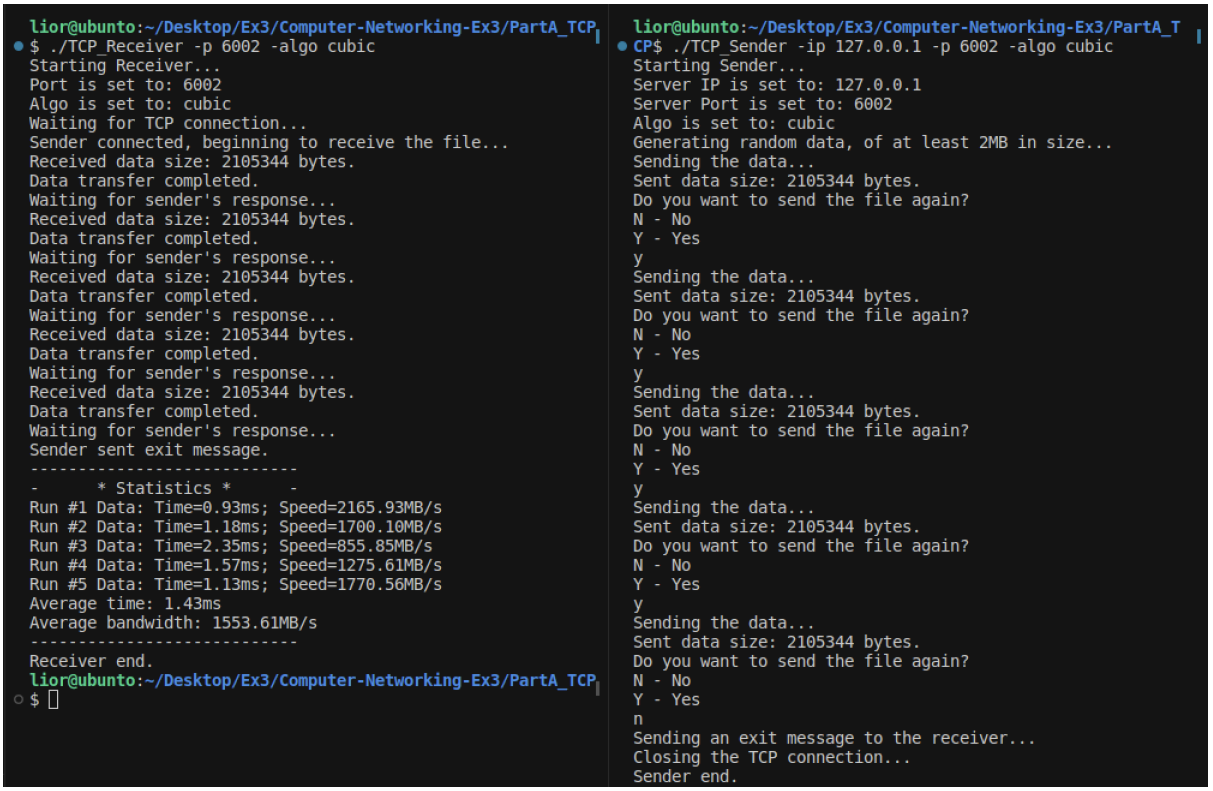


ID’s: 5303621, 211791041

2% packet loss:

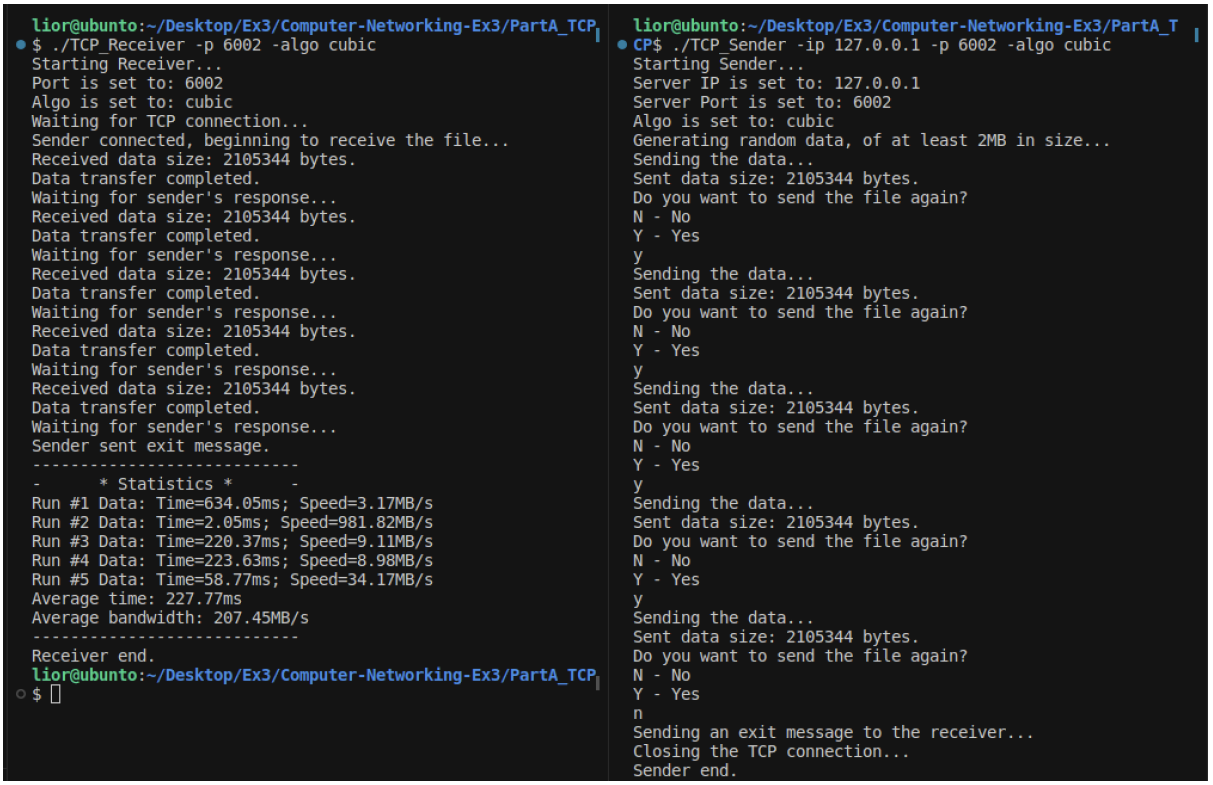


5% packet loss:



ID’s: 5303621, 211791041

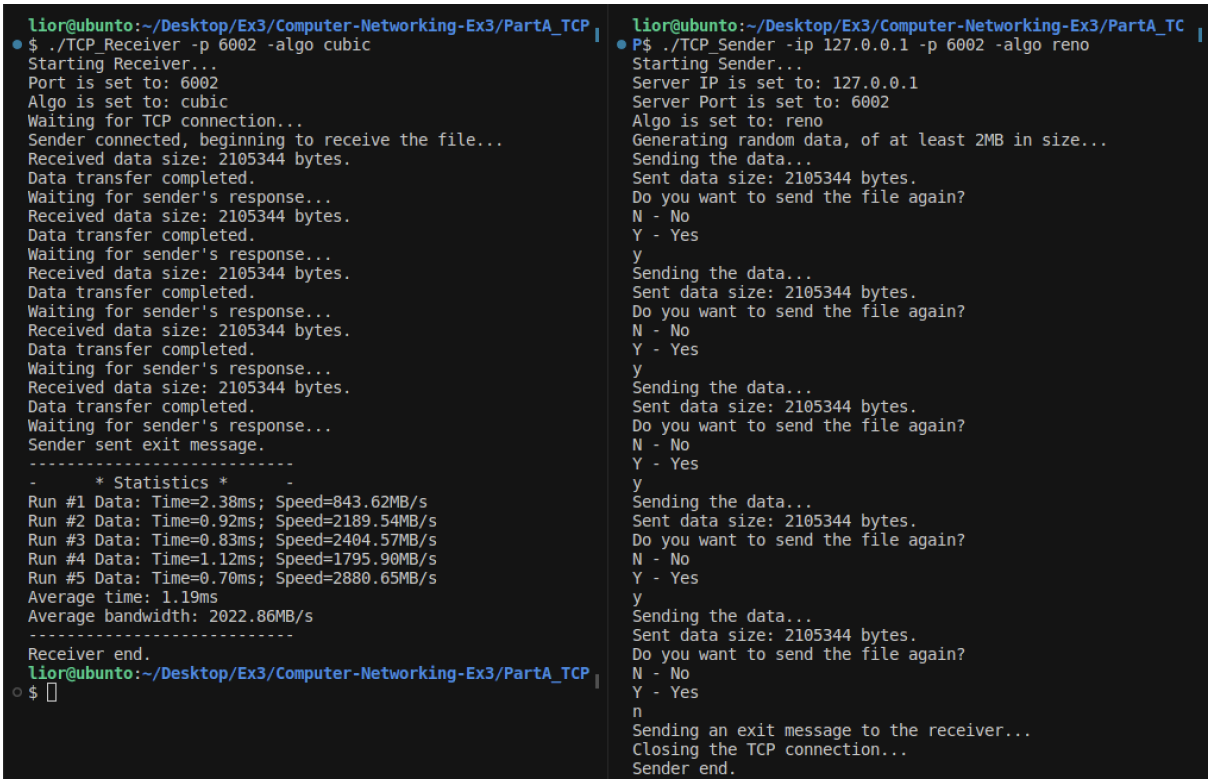
10% packet loss:



ID’s: 5303621, 211791041

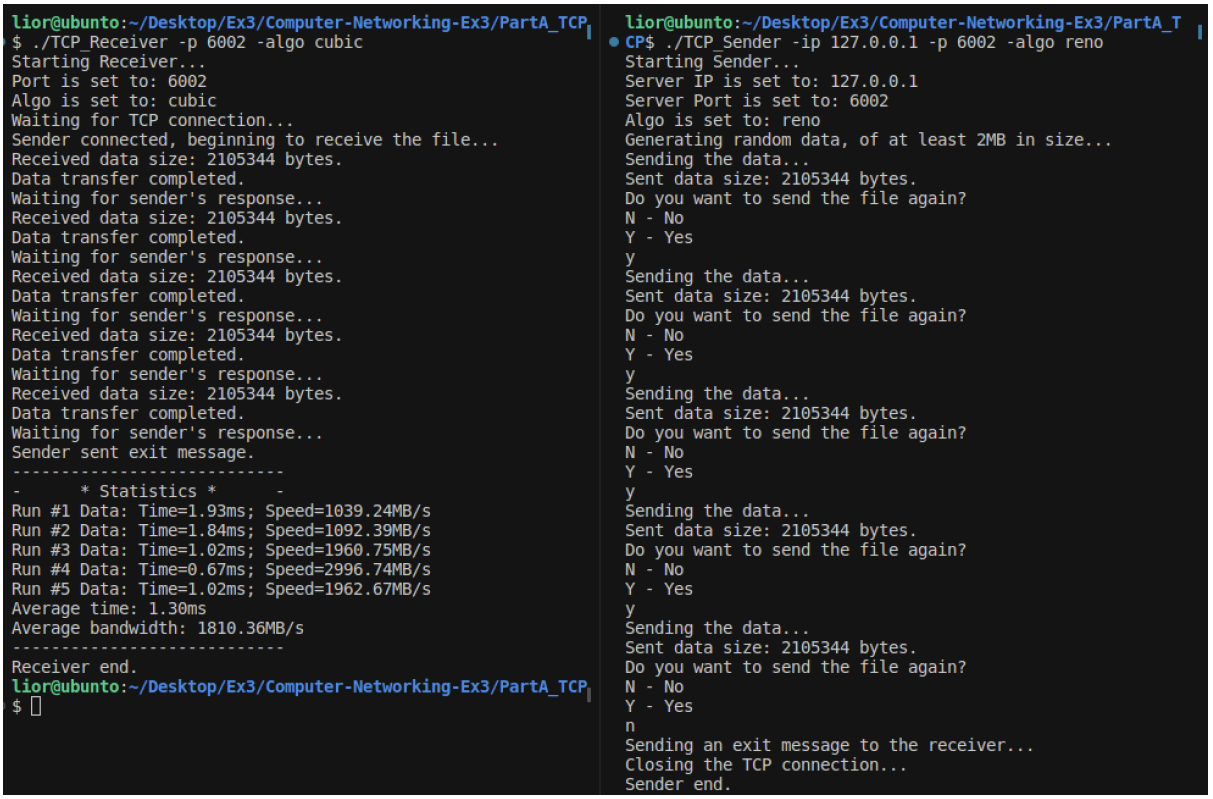
receiver: cubic
sender: reno

0% packet loss:

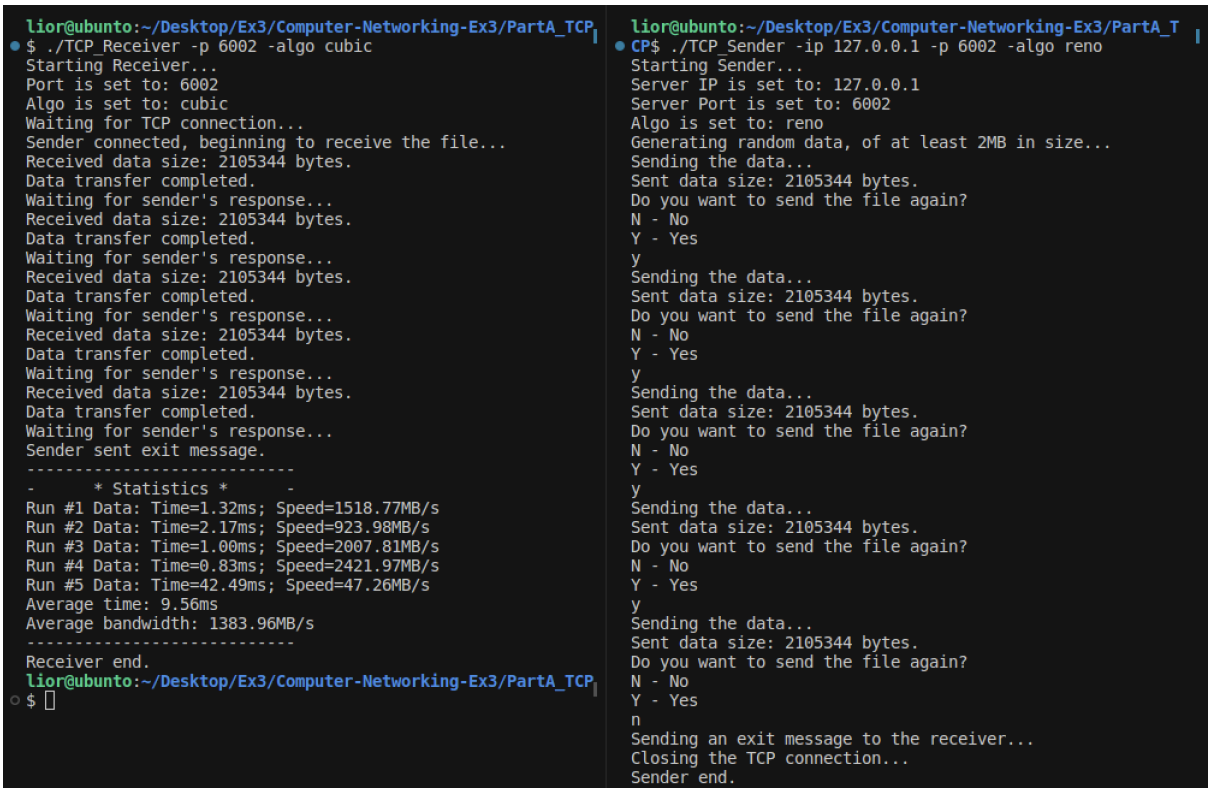


ID’s: 5303621, 211791041

2% packet loss:

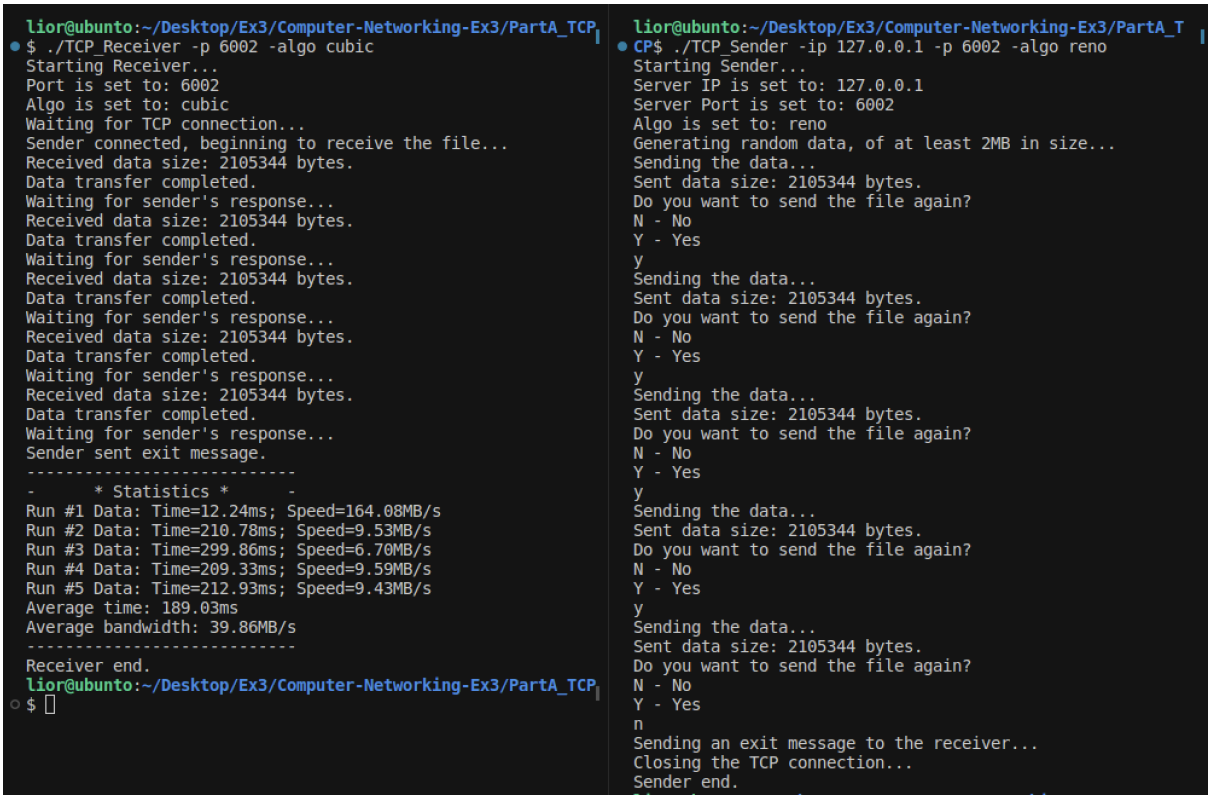


5% packet loss:



ID’s: 5303621, 211791041

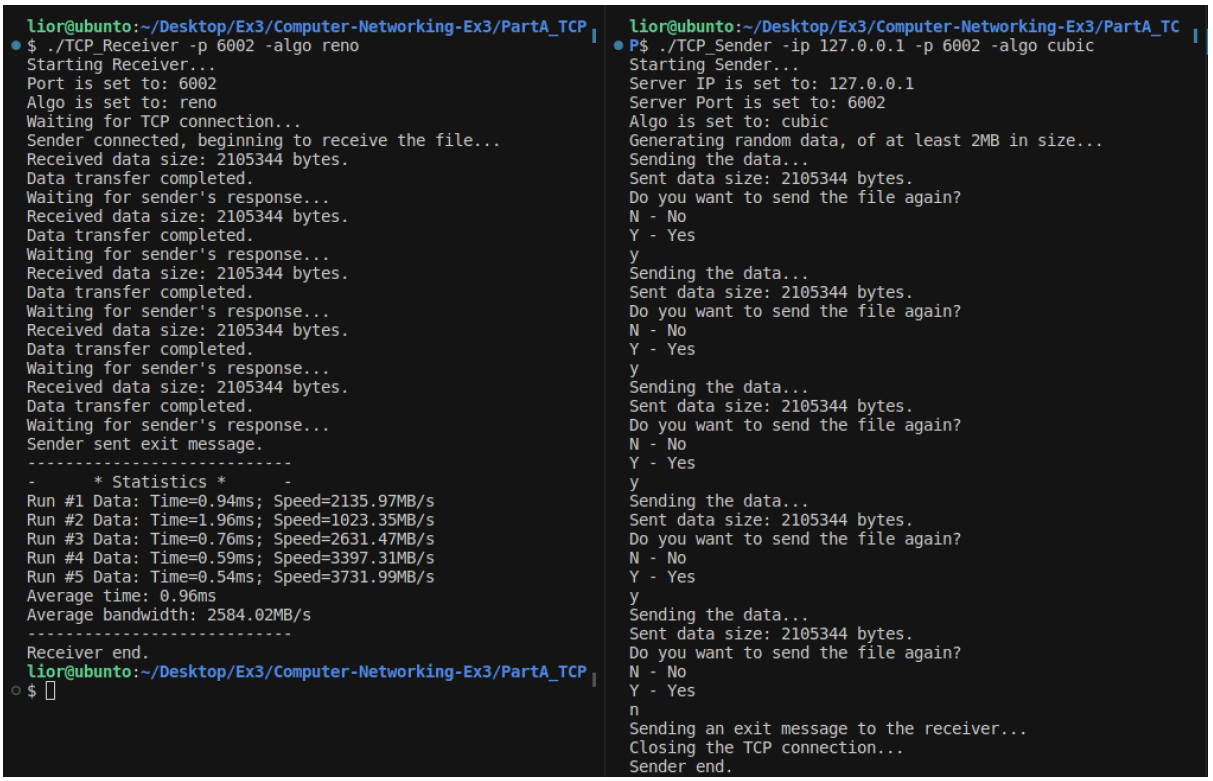
10% packet loss:



ID’s: 5303621, 211791041

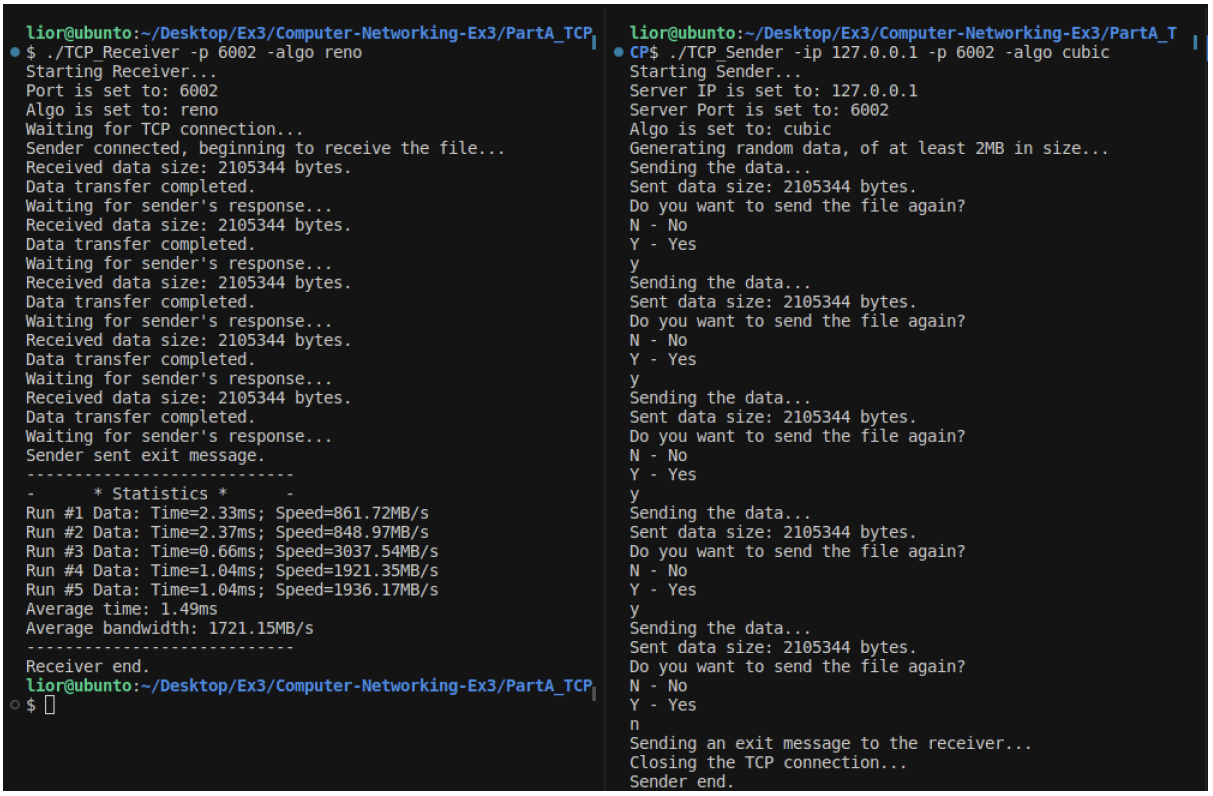
receiver: reno
sender: cubic

0% packet loss:

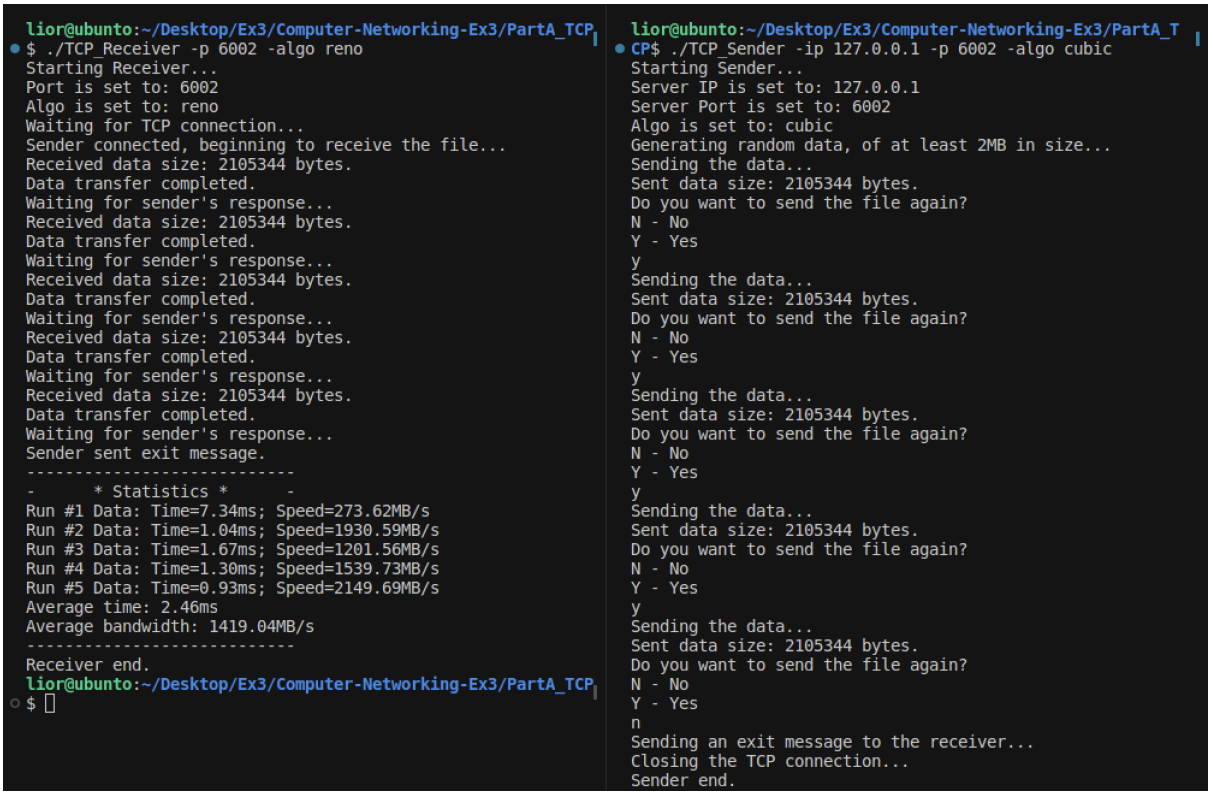


ID’s: 5303621, 211791041

2% packet loss:

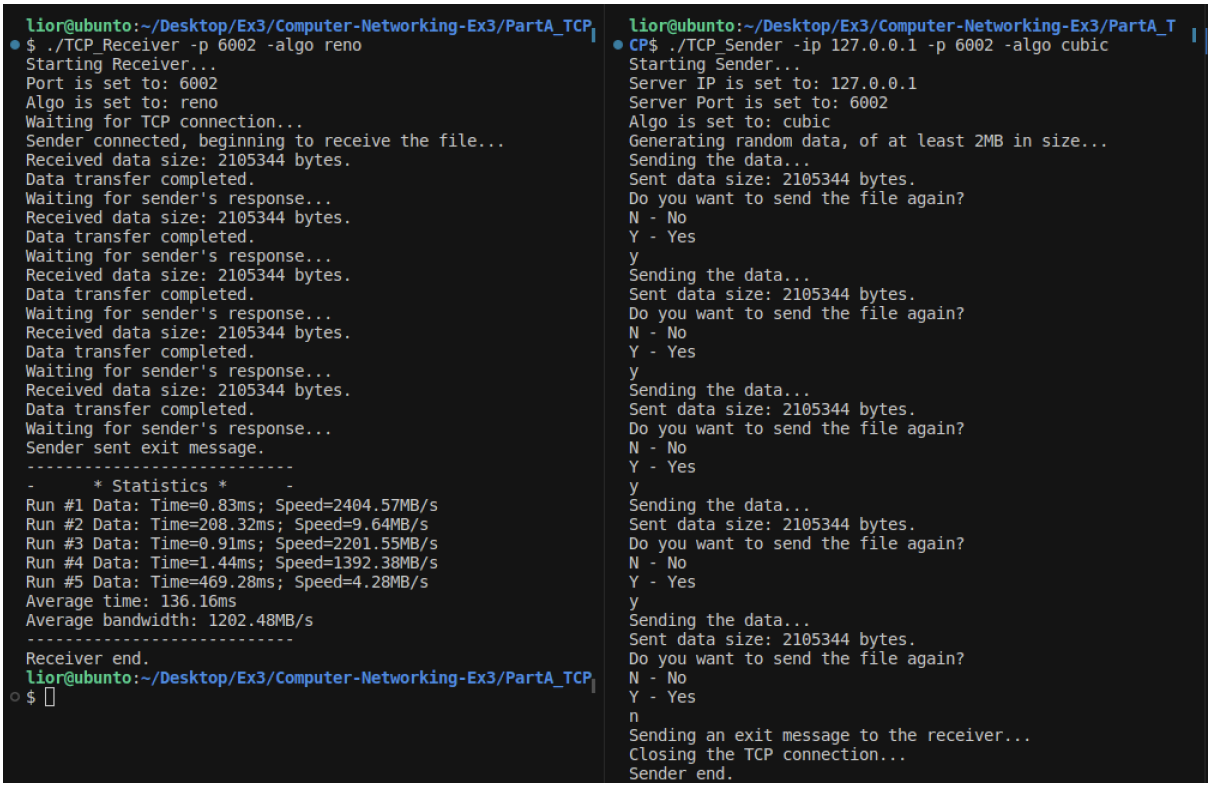


5% packet loss:



ID’s: 5303621, 211791041

10% packet loss:



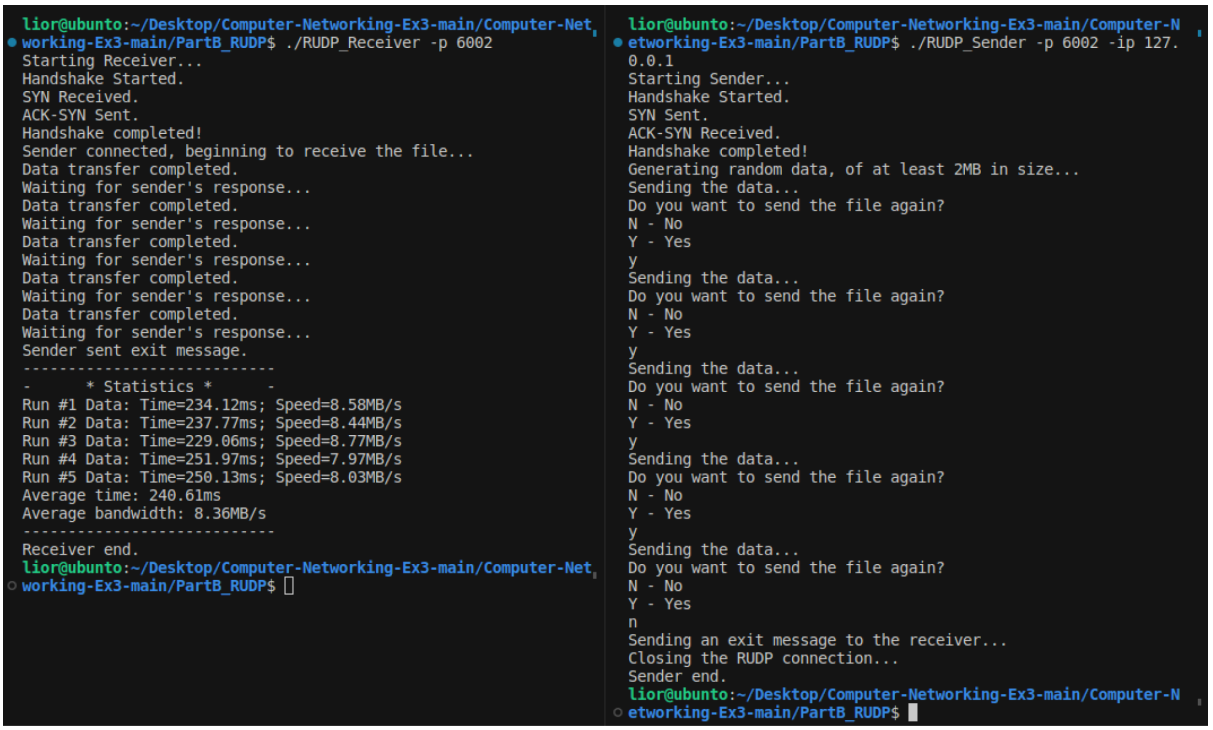
```
lior@ubunto:~/Desktop/Ex3/Computer-Networking-Ex3/PartA_TCP
$ ./TCP_Receiver -p 6002 -algo reno
Starting Receiver...
Port is set to: 6002
Algo is set to: reno
Waiting for TCP connection...
Sender connected, beginning to receive the file...
Received data size: 2105344 bytes.
Data transfer completed.
Waiting for sender's response...
Received data size: 2105344 bytes.
Data transfer completed.
Waiting for sender's response...
Received data size: 2105344 bytes.
Data transfer completed.
Waiting for sender's response...
Received data size: 2105344 bytes.
Data transfer completed.
Waiting for sender's response...
Received data size: 2105344 bytes.
Data transfer completed.
Waiting for sender's response...
Sender sent exit message.
---------------------------
-      * Statistics *      -
Run #1 Data: Time=0.83ms; Speed=2404.57MB/s
Run #2 Data: Time=208.32ms; Speed=9.64MB/s
Run #3 Data: Time=0.91ms; Speed=2201.55MB/s
Run #4 Data: Time=1.44ms; Speed=1392.38MB/s
Run #5 Data: Time=469.28ms; Speed=4.28MB/s
Average time: 136.16ms
Average bandwidth: 1202.48MB/s
---------------------------
Receiver end.
lior@ubunto:~/Desktop/Ex3/Computer-Networking-Ex3/PartA_TCP
$
```

```
lior@ubunto:~/Desktop/Ex3/Computer-Networking-Ex3/PartA_T
CP$ ./TCP_Sender -ip 127.0.0.1 -p 6002 -algo cubic
Starting Sender...
Server IP is set to: 127.0.0.1
Server Port is set to: 6002
Algo is set to: cubic
Generating random data, of at least 2MB in size...
Sending the data...
Sent data size: 2105344 bytes.
Do you want to send the file again?
N - No
Y - Yes
y
Sending the data...
Sent data size: 2105344 bytes.
Do you want to send the file again?
N - No
Y - Yes
y
Sending the data...
Sent data size: 2105344 bytes.
Do you want to send the file again?
N - No
Y - Yes
y
Sending the data...
Sent data size: 2105344 bytes.
Do you want to send the file again?
N - No
Y - Yes
y
Sending the data...
Sent data size: 2105344 bytes.
Do you want to send the file again?
N - No
Y - Yes
n
Sending an exit message to the receiver...
Closing the TCP connection...
Sender end.
```

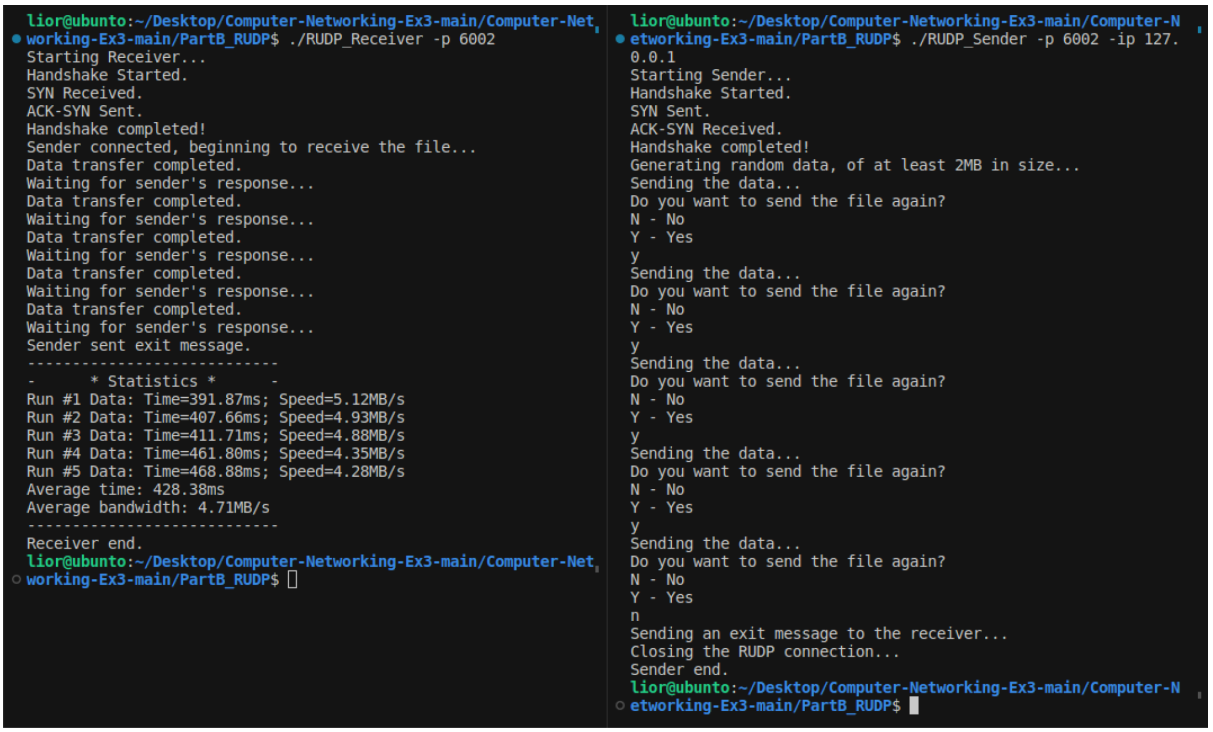
ID’s: 5303621, 211791041

RUDP:

0% packet loss:

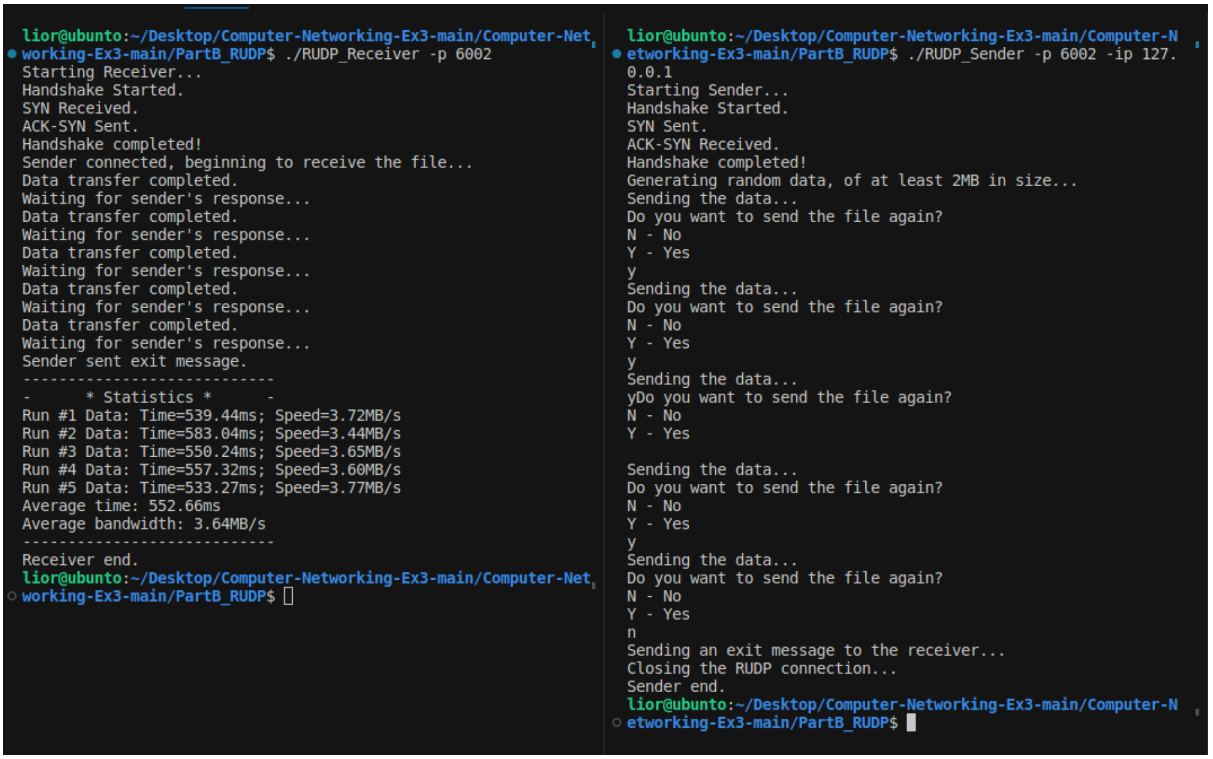


2% packet loss:



ID’s: 5303621, 211791041

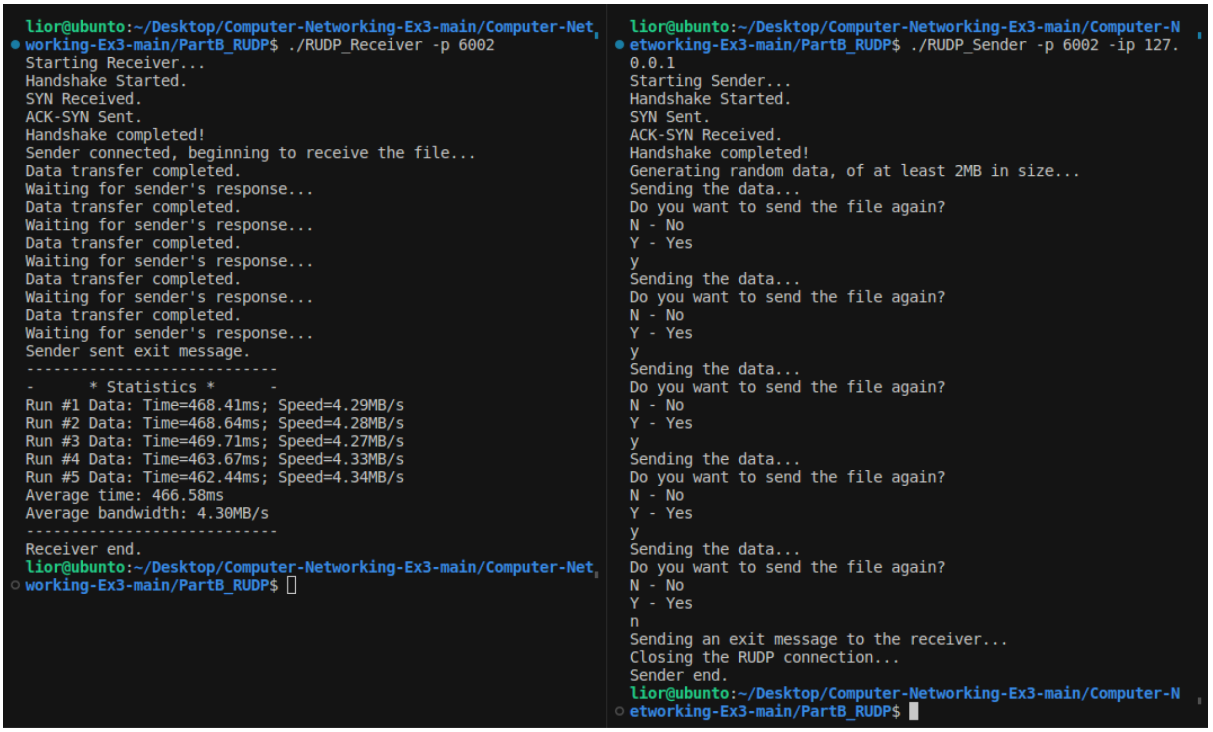
5% packet loss:



```
lior@ubuntu:~/Desktop/Computer-Networking-Ex3-main/Computer-Net
working-Ex3-main/PartB_RUDP$ ./RUDP_Receiver -p 6002
Starting Receiver...
Handshake Started.
SYN Received.
ACK-SYN Sent.
Handshake completed!
Sender connected, beginning to receive the file...
Data transfer completed.
Waiting for sender's response...
Data transfer completed.
Waiting for sender's response...
Data transfer completed.
Waiting for sender's response...
Data transfer completed.
Waiting for sender's response...
Data transfer completed.
Waiting for sender's response...
Sender sent exit message.
----------------------------
-      * Statistics *      -
Run #1 Data: Time=539.44ms; Speed=3.72MB/s
Run #2 Data: Time=583.04ms; Speed=3.44MB/s
Run #3 Data: Time=550.24ms; Speed=3.65MB/s
Run #4 Data: Time=557.32ms; Speed=3.60MB/s
Run #5 Data: Time=533.27ms; Speed=3.77MB/s
Average time: 552.66ms
Average bandwidth: 3.64MB/s
----------------------------
Receiver end.
lior@ubuntu:~/Desktop/Computer-Networking-Ex3-main/Computer-Net
working-Ex3-main/PartB_RUDP$
```

```
lior@ubuntu:~/Desktop/Computer-Networking-Ex3-main/Computer-N
etworking-Ex3-main/PartB_RUDP$ ./RUDP_Sender -p 6002 -ip 127.
0.0.1
Starting Sender...
Handshake Started.
SYN Sent.
ACK-SYN Received.
Handshake completed!
Generating random data, of at least 2MB in size...
Sending the data...
Do you want to send the file again?
N - No
Y - Yes
y
Sending the data...
Do you want to send the file again?
N - No
Y - Yes
y
Sending the data...
yDo you want to send the file again?
N - No
Y - Yes

Sending the data...
Do you want to send the file again?
N - No
Y - Yes
y
Sending the data...
Do you want to send the file again?
N - No
Y - Yes
n
Sending an exit message to the receiver...
Closing the RUDP connection...
Sender end.
lior@ubuntu:~/Desktop/Computer-Networking-Ex3-main/Computer-N
etworking-Ex3-main/PartB_RUDP$
```

10% packet loss:



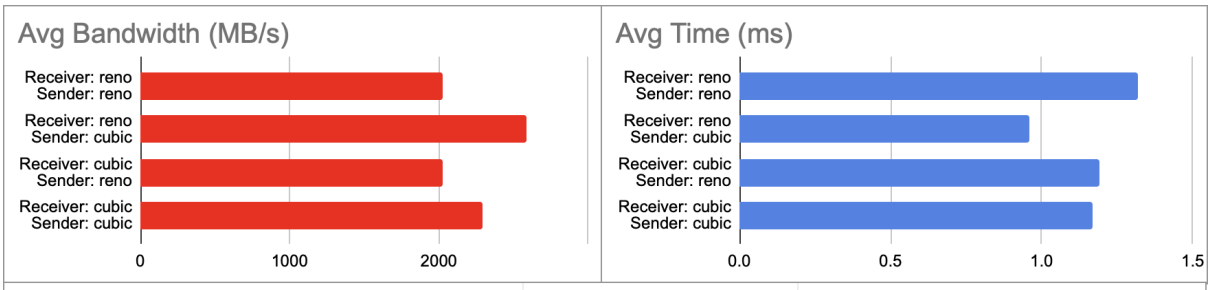
```
lior@ubuntu:~/Desktop/Computer-Networking-Ex3-main/Computer-Net
working-Ex3-main/PartB_RUDP$ ./RUDP_Receiver -p 6002
Starting Receiver...
Handshake Started.
SYN Received.
ACK-SYN Sent.
Handshake completed!
Sender connected, beginning to receive the file...
Data transfer completed.
Waiting for sender's response...
Data transfer completed.
Waiting for sender's response...
Data transfer completed.
Waiting for sender's response...
Data transfer completed.
Waiting for sender's response...
Data transfer completed.
Waiting for sender's response...
Sender sent exit message.
----------------------------
-      * Statistics *      -
Run #1 Data: Time=468.41ms; Speed=4.29MB/s
Run #2 Data: Time=468.64ms; Speed=4.28MB/s
Run #3 Data: Time=469.71ms; Speed=4.27MB/s
Run #4 Data: Time=463.67ms; Speed=4.33MB/s
Run #5 Data: Time=462.44ms; Speed=4.34MB/s
Average time: 466.58ms
Average bandwidth: 4.30MB/s
----------------------------
Receiver end.
lior@ubuntu:~/Desktop/Computer-Networking-Ex3-main/Computer-Net
working-Ex3-main/PartB_RUDP$
```

```
lior@ubuntu:~/Desktop/Computer-Networking-Ex3-main/Computer-N
etworking-Ex3-main/PartB_RUDP$ ./RUDP_Sender -p 6002 -ip 127.
0.0.1
Starting Sender...
Handshake Started.
SYN Sent.
ACK-SYN Received.
Handshake completed!
Generating random data, of at least 2MB in size...
Sending the data...
Do you want to send the file again?
N - No
Y - Yes
y
Sending the data...
Do you want to send the file again?
N - No
Y - Yes
y
Sending the data...
Do you want to send the file again?
N - No
Y - Yes
y
Sending the data...
Do you want to send the file again?
N - No
Y - Yes
y
Sending the data...
Do you want to send the file again?
N - No
Y - Yes
n
Sending an exit message to the receiver...
Closing the RUDP connection...
Sender end.
lior@ubuntu:~/Desktop/Computer-Networking-Ex3-main/Computer-N
etworking-Ex3-main/PartB_RUDP$
```

ID's: 5303621, 211791041

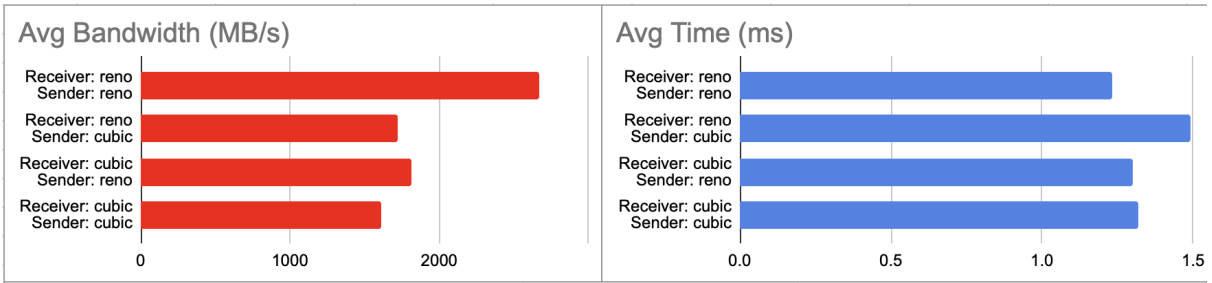
סיכום: (בונוס)

0% packet loss:



	Avg Time (ms)	Avg Bandwidth (MB/s)
Receiver: reno Sender: reno	1.32	2027.69
Receiver: reno Sender: cubic	0.96	2584.02
Receiver: cubic Sender: reno	1.19	2022.86
Receiver: cubic Sender: cubic	1.17	2288.65

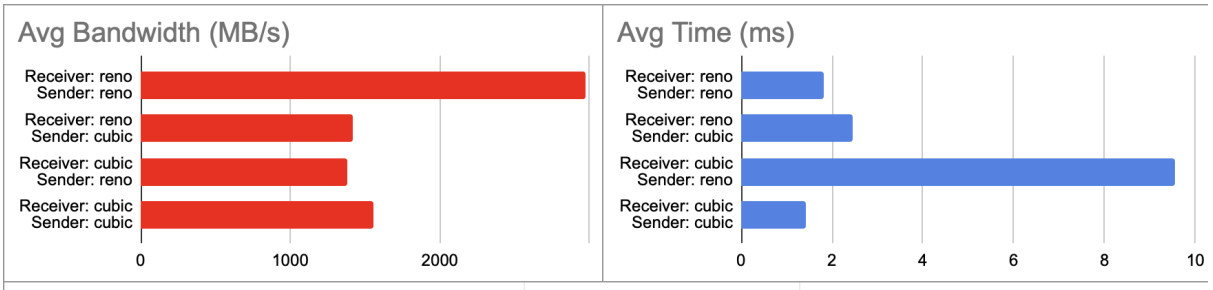
2% packet loss:



	Avg Time (ms)	Avg Bandwidth (MB/s)
Receiver: reno Sender: reno	1.23	2666.61
Receiver: reno Sender: cubic	1.49	1721.15
Receiver: cubic Sender: reno	1.3	1810.36
Receiver: cubic Sender: cubic	1.32	1606.42

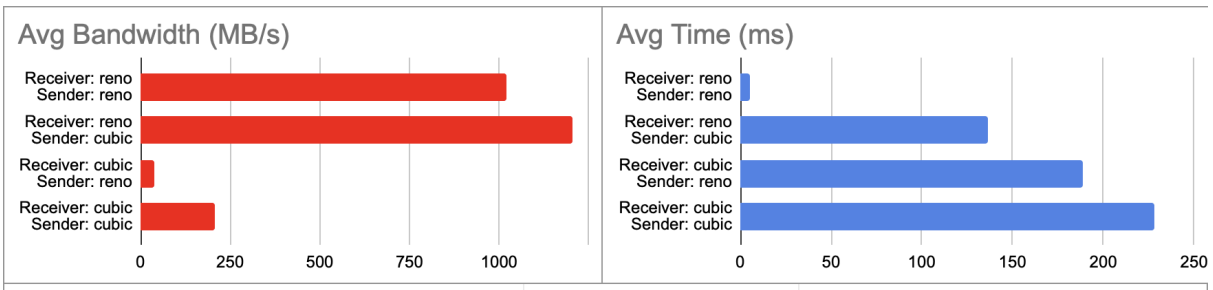
ID's: 5303621, 211791041

5% packet loss:



	Avg Time (ms)	Avg Bandwidth (MB/s)
Receiver: reno Sender: reno	1.82	2968.38
Receiver: reno Sender: cubic	2.46	1419.04
Receiver: cubic Sender: reno	9.56	1383.96
Receiver: cubic Sender: cubic	1.43	1553.61

10% packet loss:



	Avg Time (ms)	Avg Bandwidth (MB/s)
Receiver: reno Sender: reno	5.35	1019.68
Receiver: reno Sender: cubic	136.16	1202.48
Receiver: cubic Sender: reno	189.03	39.86
Receiver: cubic Sender: cubic	227.77	207.45

ID's: 5303621, 211791041

חלק ג שאלות

1. כשיש packet loss נמוך, נשים לב ש-cubic מניב תוצאות יותר טובות מאשר reno עם רוחב פס גדול יותר וזמן ממוצע נמוך יותר, עם זאת כאשר קליטת הנתונים מתבצעת עם reno ושליחת נתונים באמצעות cubic מתקבל לנו התוצאות הטובות ביותר של רוחב פס גדול וזמן ממוצע נמוך. עבור שיעורי אובדן פאקטות גבוהים יותר, רוחב הפס הממוצע של כולם יורד משמעותית והזמן הממוצע בכולם עולה משמעותית, למרות זאת reno מסתגל הרבה יותר טוב מאשר cubic לאובדן הפאקטות ומצליח לשמור על זמני הגעת הקבצים בממוצע נמוך עם רוחב פס גבוה לאורך כל הניסוי. כתוצאה מכך reno מציגה לנו פתרון יעיל ואמין יותר מcubic במקרים של אובדן פאקטות.

2. עבור RUDP, הוספנו מספר דברים שכבר קיימים ב-TCP כדי להבטיח אמינות מעל UDP; הוספנו שליחת פאקטות אישור (ACK) עם קבלת פאקטה (שידור פאקטות מתרחש רק לאחר קבלת אישור לאותן פאקטות, מה שהופך את התהליך לאמין אך גם איטי מאוד ולא יעיל), מעקב אחר מספר SEQ של פאקטה ווידוי שזו אכן הפאקטה שמצפים לקבל, checksum נוסף בheader של ה RUDP, ביצירת הsocket הוספנו handshake בו הלקוח שולח SYN ומחכה לSYN-ACK והשרת מחכה לSYN ושורת SYN-ACK, והוספנו שליחה חוזרת של פאקטות שהלכו לאיבוד (שלא קיבלנו עליהן ACK), כך שיש הבטחה שכל המידע יגיע, כמו בTCP. אך עם כל אלה, TCP עדיין אמינה יותר; TCP משתמש באלגוריתמים ל-congestion control המייעלים משמעותית את מהירות השידור. לכן, כאשר אובדן הפאקטות גבוה ויש צורך לשדר מחדש פאקטות רבות, TCP היא הבחירה המועדפת, כי היא עושה את זה בדרך היעילה ביותר, עם אלגוריתמים שנכתבו לפעול בצורה המיטבית ביותר. אנחנו משתמשים בtimeout ושליחה של מלא פאקטות בלי בקרה מושקעת כפי שיש בTCP. תוצאות הניסוי שלנו תומכות במסקנה זו, שכן זמני קליטת הקבצים היו ארוכים יותר באופן עקבי עם RUDP באותם תנאי אובדן פאקטות בהשוואה ל-TCP

3. בהתבסס על הניסוי, הבחנו כי TCP יעיל יותר הן מבחינת זמן אספקת פאקטות והן מבחינת רוחב הפס הממוצע. כתוצאה מכך, TCP היא הבחירה המועדפת כאשר אנו זקוקים להעברת נתונים מהירה, אמינה ויעילה. מצד שני, RUDP מתאים יותר כאשר אנו דורשים העברת נתונים אמינה אך וחשוב לנו להעביר רק את המידע הנחוץ ולצמצם כמה שיותר את הpacket overhead. בנוסף, UDP, שעליו בנוי RUDP, תומך בתקשורת שידור ורב-שידור, מה שמאפשר לשלוח חבילה אחת למספר נמענים בו-זמנית. תכונה זו חשובה במיוחד בשליחת מיילים, שיחות ועידה בווידאו ותרחישים אחרים שבהם אותם נתונים צריכים להיות מועברים למספר לקוחות בו-זמנית.

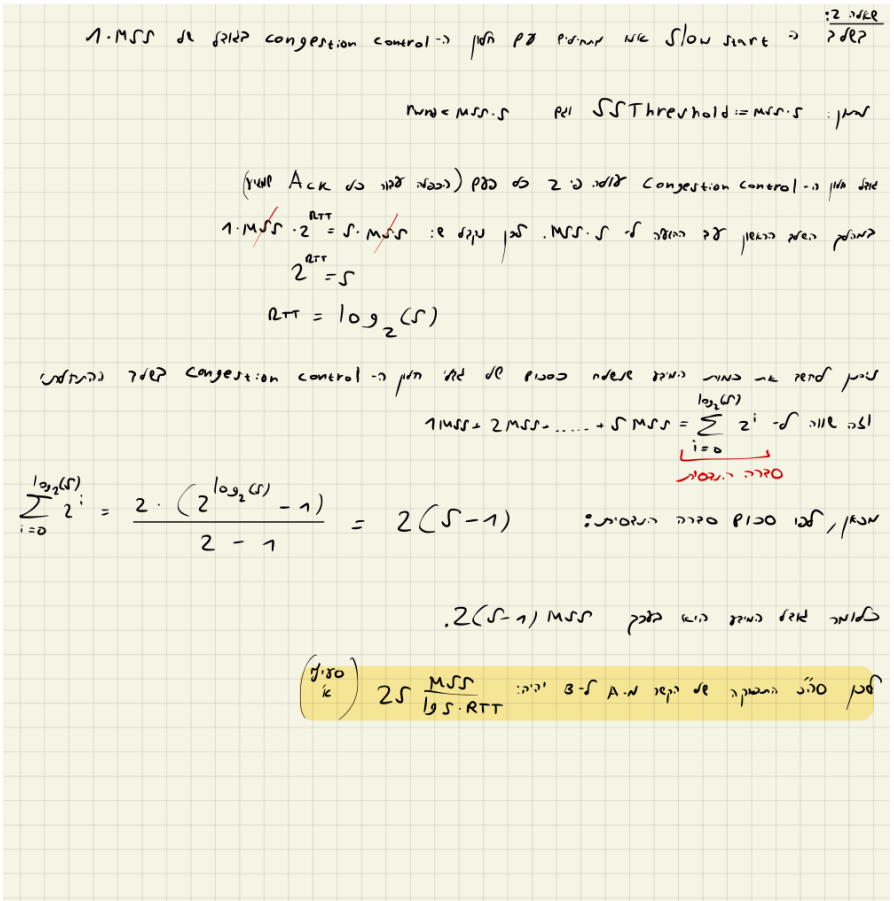
<u>שאלות נוספות:</u>

1. העלאת ssthreshold בתחילת חיבור יכולה להועיל לחיבורים ארוכים ברשת אמינה עם RTT גבוה, שכן RTT גדול מצביע על עיכוב ניכר בין השולח למקבל, אך החיבור הוא כן אמין, לכן כנראה שאנחנו לא מנצלים מספיק את הרוחב פס. אם נגדיל את הssthreshold, יהיה לנו יותר זמן ב-Slow Start, שם נגדל בקצב וכמות הפאקטות שנשלחות אקספוננציילית (זה יכול להיות מסוכן אם הרשת לא אמינה, אבל זה לא המצב פה).

ID's: 5303621, 211791041

הסבר: בTCP, ה-congestion control מתחיל בדרך כלל בשלב ה"התחלה האיטית", כאשר השולח מעביר נתונים בהתחלה לאט ובהדרגה מגדיל את גודל החלון, כל פעם פי 2 של ה-congestion control שלו, עד שהוא מגיע לגבול שהוא קבע לעצמו. משם הוא ממשיך להגדיל לינארית. אם המון פאקטות נאבדות, הוא ישר מקטין את החלון וזוכר להבא להקטין גם את הגבול שהוא קבע ולהיות זהיר יותר.
בחיבורים ארוכים עם RTT גבוה, אם ה-SSThreshold הראשוני מוגדר נמוך מדי, שלב זה עשוי להימשך זמן רב יותר להשלמתו, כלומר, יקח יותר זמן להגיע לניצול טוב של הרוחב פס. על ידי הגדלת ה-SSThreshold, ניתן לקצר את שלב ההתחלה האיטית (בכך שננצל יותר ויותר, יותר מהר), מה שמאפשר לשולח להגיע לקצב שידור אופטימלי מהר יותר. לאחר ההתחלה האיטית, TCP עובר לשלב "congestion avoidance", שם הוא ממשיך להגדיל את חלון ה-congestion control בהדרגה לינארית. כל מה שאמרנו פה מועיל במיוחד בחיבורים ארוכים ואמינים, כלומר יעיל במיוחד ב**תרחיש 5**, בו גם הRTT קטן, כלומר אנחנו מקבלים תגובה לפאקטות מאוד מהר.
תרחיש 1 גם יכול להיות אבל בגלל שיש לו RTT גדול, התגובות מגיעות לאט יותר ולכן גם אם נגדיל את הגבול והרשת תהיה אמינה, עם הרבה מידע לשלוח, כנראה שהיתרון בהגדלת הגבול לא יהיה משמעותי כי הRTT יהווה סוג של bottleneck כזה. בכל שאר התרחישים לרוב יש סכנה לאיבוד חבילות בגלל שהרשת לא אמינה או שאין כל כך הרבה מידע לשלוח ואין סיבה להעמיס במכה, אלא כדאי להיות זהירים.

.2



ID's: 5303621, 211791041

3.

שאלה 3:

נשתמש בספרת הביקורת הנמוכה ביותר, נקבל ש- $X=1$

ידוע ש- $1Gbps = 10^9$ bits/sec לכן:

$$\text{גודל החלון כדי להגיע לתפוקה מקסימלית} = \frac{\overset{8\cdot 10^9}{\text{bandwith}} \cdot RTT}{\text{גודל חבילה} = 8\cdot 10^3}$$

נמצא את RTT:

$$RTT = 2\cdot\left(\text{זמן ביט} + \text{זמן פיזור}\right)$$

$$\frac{8\cdot 10^3}{8\cdot 10^9} = \frac{\text{גודל חבילה}}{\text{Bit rate}} \qquad \frac{10^3}{2\cdot 10^8} = \frac{\text{מרחק}}{\text{קצב התפשטות}}$$

נקבל:

$$RTT = 2\cdot\left(\frac{8\cdot 10^3}{8\cdot 10^9} + \frac{10^3}{2\cdot 10^8}\right) = 1.2\cdot 10^{-5}$$

סה"כ:

$$\text{גודל החלון כדי להגיע לתפוקה מקסימלית} = \frac{8\cdot 10^9 \cdot 1.2\cdot 10^{-5}}{8\cdot 10^3} = 12$$