



Congestion Control Exercise

22.12.2022

Zev Kehat, ID 203283908

Tuvia Smadar, ID 315638577

Overview

This project was built as an exercise in a Networks & Communication course, taken as part of our BSc studies in Ariel University.

The objective of the exercise was to deepen our understanding of TCP client-server connections, with a focus on exploring congestion control algorithms and how they affect TCP data transfer, given different packet-loss levels.

The project steps:

1. We wrote the following program, based on two entities we created – Sender (acts as a TCP client) and Receiver (acts as a TCP server):
 - a. Once the Receiver accepts the TCP connection from the Sender, the Sender sends the first half (exactly 50%) of a ~2MB text file to the Receiver, which in turn records the receival time of the file. For this stage, both entities use the "Cubic" CC algorithm.
 - b. The Receiver lets the Sender know it received the file, using an authentication method (explanation below).
 - c. Both entities change the CC algorithm to "Reno", the Sender sends the second half of the text file, and the Receiver records the receival time.
 - d. The Sender decides to either resend the file or exit the program (user decision), and notifies the Receiver of its decision:
 - 1) If the Sender decides to resend the file, the CC algorithms of both Sender and Receiver are reset to "Cubic", and the above steps are repeated.
 - 2) If the Sender decides to exit the program, an exit signal is sent to the Receiver, and both sockets are closed, thus closing the connection.
 - 3) The receiver, having recorded the receival times for both halves of the file, prints:
 - a) The receival times for each half file sent.
 - b) The average receival times for each half of the file.
 - c) The average receival time for the entire file (both halves).
2. We simulated packet losses¹ of 0%, 10%, 15% and 20%. For each level of packet loss, we ran the program and sent the file from the Sender to the Receiver **5 times**.
3. While running the simulations, we recorded the network traffic using Wireshark, using the filter `"tcp && ip.src == 127.0.0.1"`².
4. The project was built in the C programming language and was run entirely on a Linux OS.

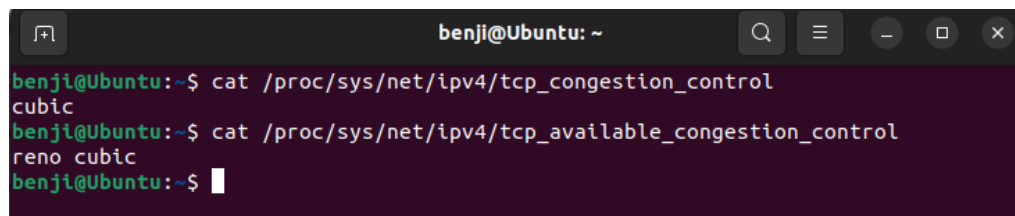
¹ Using the TC tool on Linux.

² Both Sender and Receiver were defined to be on the local host IP.

Code Overview

1. General comments:

- a. We were faced with a number of decisions on whether to use **macros**, or add functions to retrieve the data needed (e.g., having the Sender check the file size, using the `fseek()` and `ftell()` functions, sending the size as a *long int* to the Receiver and then creating the Receiver buffer with half the file size). Due to the objective of the exercise and our hope to keep the traffic to object-relevant packets, we decided in most cases to use macros (unless there was a project-related advantage to using functions).
- b. Variables that are used throughout the program are written before `main()` as **Global Variables**, so that the `main()` section holds the minimum number of variables defined within it. This way, global variables are easily understood and changeable if needed.
- c. The timing results are in **microseconds** – attempting to run the program with larger time units resulted mostly in zeros, as most of the receival times (without severe packet loss) are in the hundreds of microseconds (too small for nanoseconds).
- d. Because the program is written in C, most **errors are returned as "-1"**, therefore after many of the operations in the code, an `if(something == -1)` condition will appear. The exercise is meant to explore a functioning exchange between the Sender and Receiver, and so most of the "-1" returns will bring the program to an early ending.
- e. As mentioned, this program was designed to run on a **Linux OS**. Running it on a different OS or a simulated Linux IDE might result in errors, and reliable results are not guaranteed.
- f. **CC algorithms** – the first algorithm used in the code is the default algorithm defined in the Linux os – **Cubic**. By calling the `setsockopt()` again with the `TCP_CONGESTION` argument, the system changes the CC algorithm to the other available one – **Reno**.
 - 1) We confirmed the algorithms the system uses before writing the program:



```
benji@Ubuntu: ~  
benji@Ubuntu:~$ cat /proc/sys/net/ipv4/tcp_congestion_control  
cubic  
benji@Ubuntu:~$ cat /proc/sys/net/ipv4/tcp_available_congestion_control  
reno cubic  
benji@Ubuntu:~$
```

g. **Running the program:**


- 1) There are 2 ".c" files, `Sender.c` and `Receiver.c`, a "Makefile", and the text file to send.
- 2) Running the "make all" command in a Linux terminal (opened from the programs folder) will compile 2 executable programs – "sender" and "receiver" (lowercase!).
- 3) To run the program, two terminal windows must be opened from the executables' destination – in the first, the command `./receiver` will run the Receiver, and in the second, the command `./sender` will run the Sender. **Note:** The Sender automatically attempts to connect to the Receiver once run, so if the Sender is run before the Receiver, the Sender's connection will fail, and the Sender will close.

2. "Sender.c":

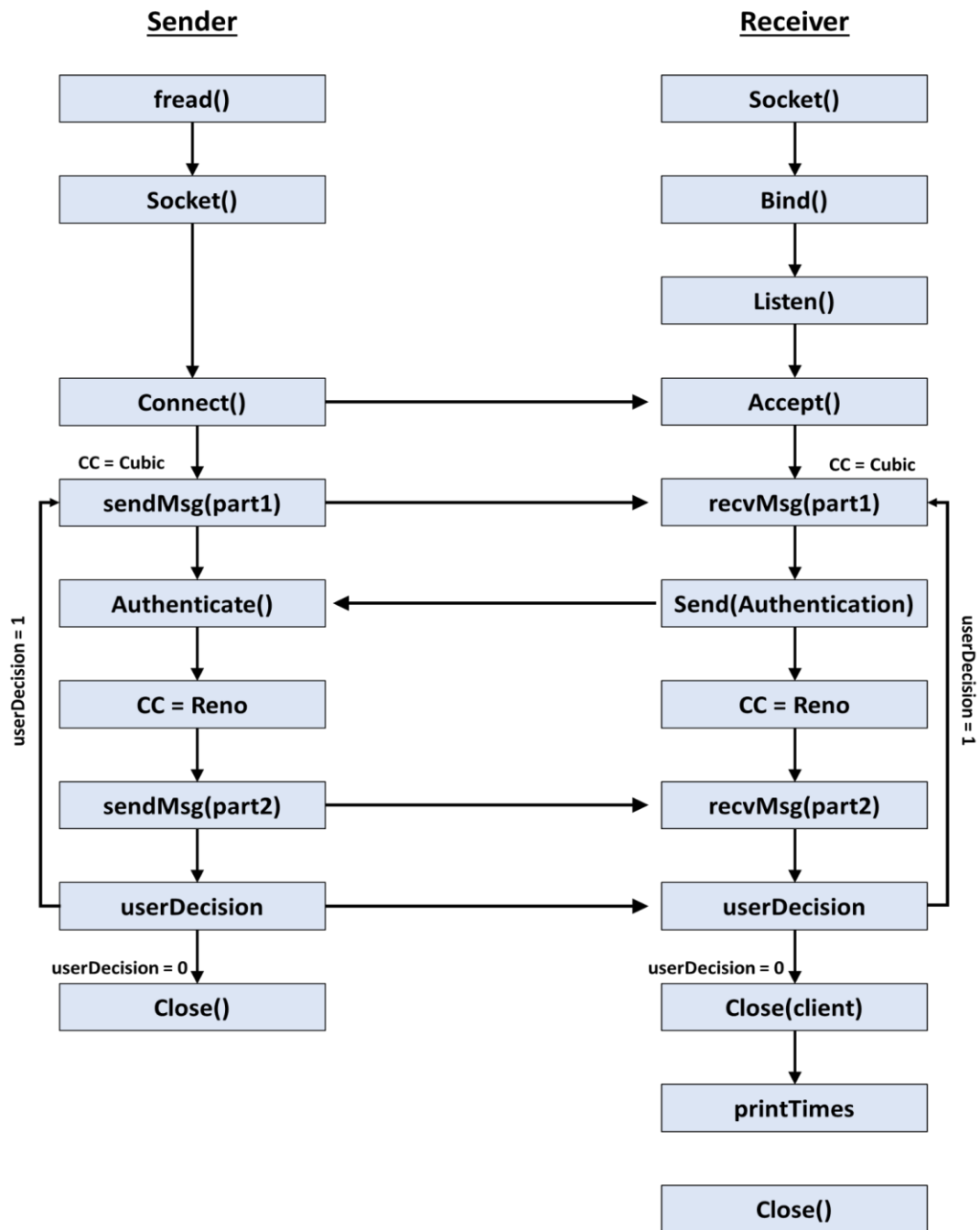
- a. Included libraries: Enabling functionalities such as system input and output, internet protocol structures and functions, and types needed for program (such as strings).
- b. Macros:
 - 1) Internet address values – IP and port.
 - 2) File size.
- c. Methods:
 - 1) ***sendMsg*** – method used to send data to another peer:
 - a) int socket – socket fd through which data is being sent.
 - b) char *fileText – pointer to the string (data) to send.
 - c) int length – length of data being sent (in bytes).
 - d) Method will print out sending status (fail, closed connection, partial send, successful send).
 - 2) ***authenticate*** – method used to check that the Receiver received first half of the file:
 - a) The method receives an int from the given socket file descriptor and compares it to the global variable "authentication" defined for Sender.
 - b) Returns 1 if number received and constant are equals, otherwise returns -1.
- d. Global Variables:
 - 1) char *fileName – name of the file to send.
 - 2) char fileText[FILESIZE] – array to which the text of the file will be copied.
 - 3) int authentication = 11973 – result of xor between the last 4 digits of our IDs.
 - 4) char ccAlgorithm[8] = {0} – array which will contain name of CC algorithm being used.
- e. Main() Steps:
 - 1) Open file, save file content as the fileText array, and close file when done copying.
 - 2) Create socket (named senderSocket) and connect to the Receiver.
 - 3) Endless while loop for sending the file. While will break upon error or user decision:
 - a) Start with Cubic CC algorithm.
 - b) Send first half of the file.
 - c) Check for authentication with Receiver.
 - d) Change the CC algorithm to Reno.
 - e) Send second half of the file.
 - f) Get a decision from the user and send that decision to the Receiver:
 - (1) 1 – resend the file (return to step (1)).
 - (2) 0 – exit program (break while loop).
 - g) Close the senderSocket and end program.

3. **"Receiver.c":**

- a. Included libraries: Enabling functionalities such as system input and output, internet protocol structures and functions, time structures and types needed for program (such as strings).
- b. Macros:
 - 1) Port chosen for the Receiver socket.
 - 2) File size (see comment 2.a.1 above).
- c. Methods:
 - 1) **recvMsg** – method used to receive data from another peer:
 - 2) int socket – socket fd through which data is being sent.
 - 3) char *buffer – pointer to the string (data) in which to write received data.
 - 4) Method will erase the data currently in the buffer, and receive data as long as the buffer has space (buffer is defined to be exactly the size of received data). When buffer has been filled with received data, it will return the number of bytes received (or -1 if there was an error and no data was received).
- d. Global Variables:
 - 1) int authentication = 11973 – result of xor between the last 4 digits of our IDs.
 - 2) char buffer[FILESIZE/2] – array to which the text of half the file will be received.
 - 3) char ccAlgorithm[8] = {0} – array which will contain name of CC algorithm being used.
- e. Main() Steps:
 - 1) Initiate arrays and values to contain and later calculate receival times. The arrays represent the two halves of the file, and are 1000 spaces long each, as per the instructions of the exercise to be able to receive up to 1000 reruns of the program.
 - 2) Create a server-like socket (named receiverSocket), bind it to the internet address of the server and listen for connections.
 - 3) Accept a connection (from Sender) and name the client's socket clientSocket.
 - 4) Enter while loop that repeats as long as indicator flag is set to 1. Flag value will change upon error or receival of exit code from Sender:
 - a) Create two objects of type "timeval" (start, stop), to measure receival times.
 - b) Receive first half of the file. Calculate receival times in microseconds, and save the result.
 - c) Send the authentication number to the Sender, to indicate receival of first half and readiness for second half.
 - d) Change the CC algorithm to Reno.
 - e) Receive second half of the file. Calculate receival times in microseconds, and save the result.
 - f) Receive the Sender's decision, to either resend the file or exit the program:
 - (1) Resend the file – change the CC algorithm and return to beginning of loop.
 - (2) Exit program - break while loop.

- 
- 5) After exiting while loop:
 - a) Close clientSocket (disconnecting from Sender).
 - b) Print out receival times for file halves.
 - c) Print out the average receival time for each half of the file.
 - d) Print out the average receival time for the whole file.
 - e) Close the receiverSocket.

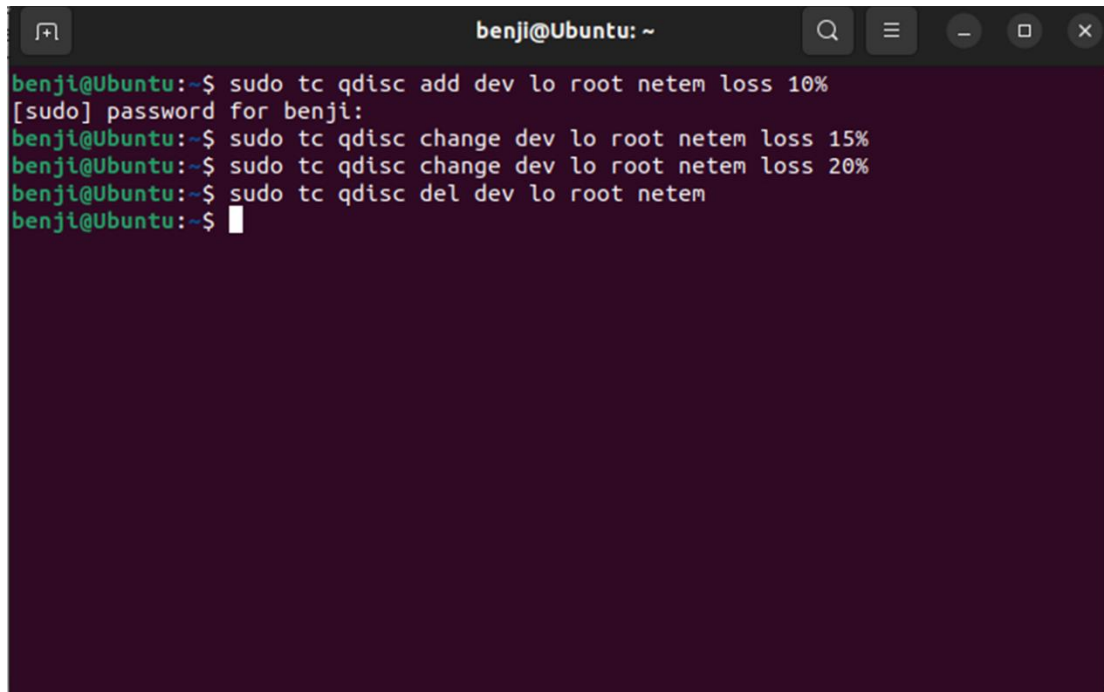
Flow Chart



Terminal Screenshots (Receiver)

1. Steps conducted:

- a. We ran the program 4 times, each time setting the packet loss (PL) level to a different percentage – 0%, 10%, 15%, 20%.
- b. In each of the 4 runs, we resent the file an extra 4 times, meaning the file was sent a total of 5 times for each PL level between 0%-20%.

A terminal window titled 'benji@Ubuntu: ~' with standard Ubuntu window controls. The terminal shows a series of commands to configure packet loss on the 'lo' interface using 'tc qdisc'. The commands are: 'sudo tc qdisc add dev lo root netem loss 10%', followed by a password prompt '[sudo] password for benji:', then 'sudo tc qdisc change dev lo root netem loss 15%', 'sudo tc qdisc change dev lo root netem loss 20%', and finally 'sudo tc qdisc del dev lo root netem'. The prompt returns to 'benji@Ubuntu:~\$' after the last command.

```
benji@Ubuntu:~$ sudo tc qdisc add dev lo root netem loss 10%
[sudo] password for benji:
benji@Ubuntu:~$ sudo tc qdisc change dev lo root netem loss 15%
benji@Ubuntu:~$ sudo tc qdisc change dev lo root netem loss 20%
benji@Ubuntu:~$ sudo tc qdisc del dev lo root netem
benji@Ubuntu:~$
```


2. Run with 0% packet loss:

- All sent messages were received successfully by the Receiver (exactly 1,048,583 bytes each).
- Average receival time for the first half of the file (Cubic) was 9,143³ microseconds.
- Average receival time for the second half of the file (Reno) was 450 microseconds.
- Average receival time for the entire file was **4,796 microseconds**.

```
benji@Ubuntu: ~/Desktop/Networks/Ex3Files
A new client connection has been accepted.
The CC algorithm now being used is: Cubic
Received message: 1048583 bytes.
The CC algorithm now being used is: Reno
Received message: 1048583 bytes.
User has decided to resend the file.
The CC algorithm now being used is: Cubic
Received message: 1048583 bytes.
The CC algorithm now being used is: Reno
Received message: 1048583 bytes.
User has decided to resend the file.
The CC algorithm now being used is: Cubic
Received message: 1048583 bytes.
The CC algorithm now being used is: Reno
Received message: 1048583 bytes.
User has decided to resend the file.
The CC algorithm now being used is: Cubic
Received message: 1048583 bytes.
The CC algorithm now being used is: Reno
Received message: 1048583 bytes.
User has decided to resend the file.
The CC algorithm now being used is: Cubic
Received message: 1048583 bytes.
The CC algorithm now being used is: Reno
Received message: 1048583 bytes.
Exit code received, goodbye.
Send number 1 took:
44206 microseconds for first half.
845 microseconds for second half.
Send number 2 took:
368 microseconds for first half.
297 microseconds for second half.
Send number 3 took:
345 microseconds for first half.
378 microseconds for second half.
Send number 4 took:
383 microseconds for first half.
382 microseconds for second half.
Send number 5 took:
417 microseconds for first half.
351 microseconds for second half.
Average receival time for the first half of the file is: 9143 microseconds
.
Average receival time for the second half of the file is: 450 microseconds
.
Average receival time for the entire file is: 4796 microseconds.
```

³ The first half in send number 1 took an abnormally long time.

3. Run with 10% packet loss:

- All sent messages were received successfully by the Receiver (exactly 1,048,583 bytes each).
- Average receive time for the first half of the file (Cubic) was 211,152 microseconds.
- Average receive time for the second half of the file (Reno) was 91,839 microseconds.
- Average receive time for the entire file was **151,459 microseconds**.

```
benji@Ubuntu: ~/Desktop/Networks/Ex3Files
A new client connection has been accepted.
The CC algorithm now being used is: Cubic
Received message: 1048583 bytes.
The CC algorithm now being used is: Reno
Received message: 1048583 bytes.
User has decided to resend the file.
The CC algorithm now being used is: Cubic
Received message: 1048583 bytes.
The CC algorithm now being used is: Reno
Received message: 1048583 bytes.
User has decided to resend the file.
The CC algorithm now being used is: Cubic
Received message: 1048583 bytes.
The CC algorithm now being used is: Reno
Received message: 1048583 bytes.
User has decided to resend the file.
The CC algorithm now being used is: Cubic
Received message: 1048583 bytes.
The CC algorithm now being used is: Reno
Received message: 1048583 bytes.
User has decided to resend the file.
The CC algorithm now being used is: Cubic
Received message: 1048583 bytes.
The CC algorithm now being used is: Reno
Received message: 1048583 bytes.
Exit code received, goodbye.
Send number 1 took:
416666 microseconds for first half.
211580 microseconds for second half.
Send number 2 took:
769 microseconds for first half.
490 microseconds for second half.
Send number 3 took:
886 microseconds for first half.
18137 microseconds for second half.
Send number 4 took:
415325 microseconds for first half.
227836 microseconds for second half.
Send number 5 took:
222115 microseconds for first half.
1156 microseconds for second half.
Average receive time for the first half of the file is: 211152 microsecon
ds.
Average receive time for the second half of the file is: 91839 microsecon
ds.
Average receive time for the entire file is: 151495 microseconds.
```

4. **Run with 15% packet loss:**

- a. All sent messages were received successfully by the Receiver (exactly 1,048,583 bytes each).
- b. Average receive time for the first half of the file (Cubic) was 118,862 microseconds.
- c. Average receive time for the second half of the file (Reno) was 170,484 microseconds.
- d. Average receive time for the entire file was **144,673 microseconds**.

```
benji@Ubuntu: ~/Desktop/Networks/Ex3Files
A new client connection has been accepted.
The CC algorithm now being used is: Cubic
Received message: 1048583 bytes.
The CC algorithm now being used is: Reno
Received message: 1048583 bytes.
User has decided to resend the file.
The CC algorithm now being used is: Cubic
Received message: 1048583 bytes.
The CC algorithm now being used is: Reno
Received message: 1048583 bytes.
User has decided to resend the file.
The CC algorithm now being used is: Cubic
Received message: 1048583 bytes.
The CC algorithm now being used is: Reno
Received message: 1048583 bytes.
User has decided to resend the file.
The CC algorithm now being used is: Cubic
Received message: 1048583 bytes.
The CC algorithm now being used is: Reno
Received message: 1048583 bytes.
User has decided to resend the file.
The CC algorithm now being used is: Cubic
Received message: 1048583 bytes.
The CC algorithm now being used is: Reno
Received message: 1048583 bytes.
Exit code received, goodbye.
Send number 1 took:
224949 microseconds for first half.
224075 microseconds for second half.
Send number 2 took:
76685 microseconds for first half.
255977 microseconds for second half.
Send number 3 took:
196 microseconds for first half.
222741 microseconds for second half.
Send number 4 took:
362 microseconds for first half.
74080 microseconds for second half.
Send number 5 took:
292122 microseconds for first half.
75551 microseconds for second half.
Average receive time for the first half of the file is: 118862 microsecon
ds.
Average receive time for the second half of the file is: 170484 microseco
nds.
Average receive time for the entire file is: 144673 microseconds.
```

5. **Run with 20% packet loss:**

- a. All sent messages were received successfully by the Receiver (exactly 1,048,583 bytes each).
- b. Average receive time for the first half of the file (Cubic) was 4,106,282 microseconds.
- c. Average receive time for the second half of the file (Reno) was 1,197,228 microseconds.
- d. Average receive time for the entire file was **2,651,755 microseconds**.

```
benji@Ubuntu: ~/Desktop/Networks/Ex3Files
A new client connection has been accepted.
The CC algorithm now being used is: Cubic
Received message: 1048583 bytes.
The CC algorithm now being used is: Reno
Received message: 1048583 bytes.
User has decided to resend the file.
The CC algorithm now being used is: Cubic
Received message: 1048583 bytes.
The CC algorithm now being used is: Reno
Received message: 1048583 bytes.
User has decided to resend the file.
The CC algorithm now being used is: Cubic
Received message: 1048583 bytes.
The CC algorithm now being used is: Reno
Received message: 1048583 bytes.
User has decided to resend the file.
The CC algorithm now being used is: Cubic
Received message: 1048583 bytes.
The CC algorithm now being used is: Reno
Received message: 1048583 bytes.
User has decided to resend the file.
The CC algorithm now being used is: Cubic
Received message: 1048583 bytes.
The CC algorithm now being used is: Reno
Received message: 1048583 bytes.
Exit code received, goodbye.
Send number 1 took:
18015907 microseconds for first half.
3936584 microseconds for second half.
Send number 2 took:
1299174 microseconds for first half.
819370 microseconds for second half.
Send number 3 took:
974371 microseconds for first half.
674139 microseconds for second half.
Send number 4 took:
19729 microseconds for first half.
244629 microseconds for second half.
Send number 5 took:
222232 microseconds for first half.
311421 microseconds for second half.
Average receive time for the first half of the file is: 4106282 microseco
nds.
Average receive time for the second half of the file is: 1197228 microsec
onds.
Average receive time for the entire file is: 2651755 microseconds.
```

Wireshark Captures⁴

1. General Comments:

- a. When running the program with the different PL levels on a relatively "sterile" environment (home network), the first obvious fact was the difference in the size of the different captures:
 - 1) 0% PL – 4.82 MB, 190 Packets.
 - 2) 10% PL – 7.24 MB, 249 Packets.
 - 3) 15% PL – 8.65 MB, 297 Packets.
 - 4) 20% PL – 9.90 MB, 341 Packets.
- b. From the data above, and other trials we conducted, it seems that the biggest change in size is between the 0% PL and the 10% PL.
- c. To verify the PL levels, we compared the captures while filtering to show only TCP retransmissions in each run, using the `tcp.analysis.retransmission` filter:
 - 1) 0% PL – 0 retransmissions, as expected.
 - 2) 10% PL – 5.2% of packets.
 - 3) 15% PL – 6.4% of packets.
 - 4) 20% PL – 9.1% of packets.

The data, and especially the variations between percentages of retransmission packets in the different PL levels, shows a logical pattern but also shows the problem in analyzing the data to definitive conclusions. Seeing as the packet loss is random, the volume of packets lost changes and affects how the program runs.

⁴ **Note:** All Wireshark captures were handed in as part of the assignment zip file.

2. 0% PL:

These are the first 3 packets we can see on the 0% PL. These packets show the Three-Way Handshake between the Sender and the Receiver.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	127.0.0.1	127.0.0.1	TCP	76	38598 → 1604 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM=...
2	0.000024125	127.0.0.1	127.0.0.1	TCP	76	1604 → 38598 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495...
3	0.000042616	127.0.0.1	127.0.0.1	TCP	68	38598 → 1604 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=34752915...

After that we can see all the first half of file sent by chunks:

No.	Time	Source	Destination	Protocol	Length	Info
35	0.001502134	127.0.0.1	127.0.0.1	TCP	32809	38598 → 1604 [PSH, ACK] Seq=556598 Ack=1 Win=65536 Len=32741 ...
36	0.001522310	127.0.0.1	127.0.0.1	TCP	68	1604 → 38598 [ACK] Seq=1 Ack=589339 Win=1375232 Len=0 TSval=3...
37	0.001527546	127.0.0.1	127.0.0.1	TCP	32809	38598 → 1604 [ACK] Seq=589339 Ack=1 Win=65536 Len=32741 TSval=...
38	0.001531288	127.0.0.1	127.0.0.1	TCP	68	1604 → 38598 [ACK] Seq=1 Ack=622080 Win=1506176 Len=0 TSval=3...
39	0.001535982	127.0.0.1	127.0.0.1	TCP	32809	38598 → 1604 [PSH, ACK] Seq=622080 Ack=1 Win=65536 Len=32741 ...
40	0.001539758	127.0.0.1	127.0.0.1	TCP	68	1604 → 38598 [ACK] Seq=1 Ack=654821 Win=1637120 Len=0 TSval=3...
41	0.001581975	127.0.0.1	127.0.0.1	TCP	32809	38598 → 1604 [ACK] Seq=654821 Ack=1 Win=65536 Len=32741 TSval=...
42	0.001585474	127.0.0.1	127.0.0.1	TCP	68	1604 → 38598 [ACK] Seq=1 Ack=687562 Win=1768064 Len=0 TSval=3...
43	0.001592625	127.0.0.1	127.0.0.1	TCP	32809	38598 → 1604 [PSH, ACK] Seq=687562 Ack=1 Win=65536 Len=32741 ...
44	0.001596165	127.0.0.1	127.0.0.1	TCP	68	1604 → 38598 [ACK] Seq=1 Ack=720303 Win=1899008 Len=0 TSval=3...
45	0.001600935	127.0.0.1	127.0.0.1	TCP	32809	38598 → 1604 [ACK] Seq=720303 Ack=1 Win=65536 Len=32741 TSval=...
46	0.001604235	127.0.0.1	127.0.0.1	TCP	68	1604 → 38598 [ACK] Seq=1 Ack=753044 Win=2030080 Len=0 TSval=3...
47	0.001609090	127.0.0.1	127.0.0.1	TCP	32809	38598 → 1604 [PSH, ACK] Seq=753044 Ack=1 Win=65536 Len=32741 ...
48	0.001612367	127.0.0.1	127.0.0.1	TCP	68	1604 → 38598 [ACK] Seq=1 Ack=785785 Win=2161024 Len=0 TSval=3...
49	0.001617257	127.0.0.1	127.0.0.1	TCP	32809	38598 → 1604 [ACK] Seq=785785 Ack=1 Win=65536 Len=32741 TSval=...
50	0.001620945	127.0.0.1	127.0.0.1	TCP	68	1604 → 38598 [ACK] Seq=1 Ack=818526 Win=2291968 Len=0 TSval=3...
51	0.001625813	127.0.0.1	127.0.0.1	TCP	32809	38598 → 1604 [PSH, ACK] Seq=818526 Ack=1 Win=65536 Len=32741 ...
52	0.001629113	127.0.0.1	127.0.0.1	TCP	68	1604 → 38598 [ACK] Seq=1 Ack=851267 Win=2422912 Len=0 TSval=3...
53	0.001633856	127.0.0.1	127.0.0.1	TCP	32809	38598 → 1604 [ACK] Seq=851267 Ack=1 Win=65536 Len=32741 TSval=...
54	0.001637197	127.0.0.1	127.0.0.1	TCP	68	1604 → 38598 [ACK] Seq=1 Ack=884008 Win=2553856 Len=0 TSval=3...
55	0.001642183	127.0.0.1	127.0.0.1	TCP	32809	38598 → 1604 [PSH, ACK] Seq=884008 Ack=1 Win=65536 Len=32741 ...
56	0.001645466	127.0.0.1	127.0.0.1	TCP	68	1604 → 38598 [ACK] Seq=1 Ack=916749 Win=2684928 Len=0 TSval=3...
57	0.045032268	127.0.0.1	127.0.0.1	TCP	68	[TCP ACKed unseen segment] 1604 → 38598 [ACK] Seq=1 Ack=10477...
58	0.045058301	127.0.0.1	127.0.0.1	TCP	939	[TCP Previous segment not captured] 38598 → 1604 [PSH, ACK] S...
59	0.045063663	127.0.0.1	127.0.0.1	TCP	68	[TCP ACKed unseen segment] 1604 → 38598 [ACK] Seq=1 Ack=10485...
60	0.045139016	127.0.0.1	127.0.0.1	TCP	72	1604 → 38598 [PSH, ACK] Seq=1 Ack=1048584 Win=3089664 Len=4 T...

In the screenshot above we can see the data transfer (packets 4-59).

In packets 57-59 we can see the Receiver warning the Sender that he ACKed an unseen segment, thus having the Sender resend the needed segment. Right after the Sender will send the last bytes of the first half of file.

In packet 60 we can see the authentication number being sent (int of the size of 4 bytes):

▶ Frame 60: 72 bytes on wire (576 bits), 72 bytes captured (576 bits) on interface any, id 0	
▶ Linux cooked capture	
▶ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1	
▶ Transmission Control Protocol, Src Port: 1604, Dst Port: 38598, Seq: 1, Ack: 1048584, Len: 4	
▶ Data (4 bytes)	
0000	00 00 03 04 00 06 00 00 00 00 00 00 45 38 08 00E8..
0010	45 00 00 38 90 a4 40 00 40 06 ac 19 7f 00 00 01 E..8..@. @.....
0020	7f 00 00 01 06 44 96 c6 99 9e 5c e2 90 53 1d f8 ...D.. \..S..
0030	80 18 5e 4a fe 2c 00 00 01 01 08 0a cf 24 bd c0 ...^J, .. \$..
0040	cf 24 bd c0 c5 2e 00 00 ..\$.....

(2ec5) = $ID1 \wedge ID2$ in Hexadecimal. *(look at row 0040)

At this stage, each one of the peers will change the CC algorithm being used, and the Sender will send the second half of the file:

60	0.045139016	127.0.0.1	127.0.0.1	TCP	72	1604 → 38598	[PSH, ACK] Seq=1 Ack=1048584 Win=3089664 Len=4 T...
61	0.045141232	127.0.0.1	127.0.0.1	TCP	68	38598 → 1604	[ACK] Seq=1048584 Ack=5 Win=65536 Len=0 TSval=34...
62	0.045384606	127.0.0.1	127.0.0.1	TCP	65551	38598 → 1604	[ACK] Seq=1048584 Ack=5 Win=65536 Len=65483 TSva...
63	0.045427510	127.0.0.1	127.0.0.1	TCP	65551	38598 → 1604	[ACK] Seq=1114067 Ack=5 Win=65536 Len=65483 TSva...
64	0.045446237	127.0.0.1	127.0.0.1	TCP	68	1604 → 38598	[ACK] Seq=5 Ack=1179550 Win=3112448 Len=0 TSval=...
65	0.045473181	127.0.0.1	127.0.0.1	TCP	65551	38598 → 1604	[ACK] Seq=1179550 Ack=5 Win=65536 Len=65483 TSva...
66	0.045519453	127.0.0.1	127.0.0.1	TCP	65551	38598 → 1604	[ACK] Seq=1245033 Ack=5 Win=65536 Len=65483 TSva...
67	0.045539091	127.0.0.1	127.0.0.1	TCP	68	1604 → 38598	[ACK] Seq=5 Ack=1310516 Win=3112448 Len=0 TSval=...
68	0.045565091	127.0.0.1	127.0.0.1	TCP	65551	38598 → 1604	[ACK] Seq=1310516 Ack=5 Win=65536 Len=65483 TSva...
69	0.045605532	127.0.0.1	127.0.0.1	TCP	65551	38598 → 1604	[ACK] Seq=1375999 Ack=5 Win=65536 Len=65483 TSva...
70	0.045616622	127.0.0.1	127.0.0.1	TCP	68	1604 → 38598	[ACK] Seq=5 Ack=1441482 Win=3112448 Len=0 TSval=...
71	0.045644192	127.0.0.1	127.0.0.1	TCP	65551	38598 → 1604	[ACK] Seq=1441482 Ack=5 Win=65536 Len=65483 TSva...
72	0.045678428	127.0.0.1	127.0.0.1	TCP	65551	38598 → 1604	[ACK] Seq=1506965 Ack=5 Win=65536 Len=65483 TSva...
73	0.045688970	127.0.0.1	127.0.0.1	TCP	68	1604 → 38598	[ACK] Seq=5 Ack=1572448 Win=3112448 Len=0 TSval=...
74	0.045714078	127.0.0.1	127.0.0.1	TCP	65551	38598 → 1604	[ACK] Seq=1572448 Ack=5 Win=65536 Len=65483 TSva...
75	0.045744503	127.0.0.1	127.0.0.1	TCP	65551	38598 → 1604	[ACK] Seq=1637931 Ack=5 Win=65536 Len=65483 TSva...
76	0.045753112	127.0.0.1	127.0.0.1	TCP	68	1604 → 38598	[ACK] Seq=5 Ack=1703414 Win=3112448 Len=0 TSval=...
77	0.045776472	127.0.0.1	127.0.0.1	TCP	65551	38598 → 1604	[ACK] Seq=1703414 Ack=5 Win=65536 Len=65483 TSva...

In the second half of the file we can see that the Reno updated the segment size (from 32KB to 64KB) and it stays like that in the next packets from the Sender.

The following screenshot shows the Sender resending the user decision value, because of an uncaptured segment (as you can see in row 0040, we chose 1 – to resend the file).

```

> Frame 81: 69 bytes on wire (552 bits), 69 bytes captured (552 bits) on interface any, id 0
> Linux cooked capture
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
> Transmission Control Protocol, Src Port: 38598, Dst Port: 1604, Seq: 2097167, Ack: 5, Len: 1
> Data (1 byte)

```

```

0000 00 00 03 04 00 06 00 00 00 00 00 00 42 63 08 00 .....Bc..
0010 45 00 00 35 39 3c 40 00 40 06 03 85 7f 00 00 01 E..59<@. @.....
0020 7f 00 00 01 96 c6 06 44 90 63 1d ff 99 9e 5c e6 .....D .c.....\
0030 80 18 02 00 fe 29 00 00 01 01 08 0a cf 24 ea 0d .....).$.
0040 cf 24 bd ec 01 .....$.

```

80	0.088514804	127.0.0.1	127.0.0.1	TCP	68	[TCP ACKed unseen segment] 1604 → 38598 [ACK] Seq=5 Ack=20971...
81	11.385625521	127.0.0.1	127.0.0.1	TCP	69	[TCP Previous segment not captured] 38598 → 1604 [PSH, ACK] S...
82	11.385641353	127.0.0.1	127.0.0.1	TCP	68	[TCP ACKed unseen segment] 1604 → 38598 [ACK] Seq=5 Ack=20971...

The rest of the file contains the resending of the file, for another 4 times, meaning a repeat of the above (excluding the handshake, because the TCP connection stays alive). Finally, we see the user decision to exit the program being sent (1 byte of data, containing "00"), after which the connection will close:

185	18.375572868	127.0.0.1	127.0.0.1	TCP	69	38598 → 1604 [PSH, ACK] Seq=10485835 Ack=21 Win=65536 Len=1 T...
186	18.375590261	127.0.0.1	127.0.0.1	TCP	68	1604 → 38598 [ACK] Seq=21 Ack=10485836 Win=3144704 Len=0 TSva...
187	18.375698959	127.0.0.1	127.0.0.1	TCP	68	1604 → 38598 [FIN, ACK] Seq=21 Ack=10485836 Win=3144704 Len=0...
188	18.417054682	127.0.0.1	127.0.0.1	TCP	68	38598 → 1604 [ACK] Seq=10485836 Ack=22 Win=65536 Len=0 TSval=...
189	19.376152305	127.0.0.1	127.0.0.1	TCP	68	38598 → 1604 [FIN, ACK] Seq=10485836 Ack=22 Win=65536 Len=0 T...
190	19.376168763	127.0.0.1	127.0.0.1	TCP	68	1604 → 38598 [ACK] Seq=22 Ack=10485837 Win=3144704 Len=0 TSva...

```

> Frame 185: 69 bytes on wire (552 bits), 69 bytes captured (552 bits) on interface any, id 0
> Linux cooked capture
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
> Transmission Control Protocol, Src Port: 38598, Dst Port: 1604, Seq: 10485835, Ack: 21, Len: 1
> Data (1 byte)

```

```

0000 00 00 03 04 00 06 00 00 00 00 00 00 75 36 08 00 .....u6..
0010 45 00 00 35 39 c8 40 00 40 06 02 f9 7f 00 00 01 E..59@. @.....
0020 7f 00 00 01 96 c6 06 44 90 e3 1e 3b 99 9e 5c f6 .....D .;.....
0030 80 18 02 00 fe 29 00 00 01 01 08 0a cf 25 05 5b .....).%.
0040 cf 25 00 10 00 .....%.

```

3. 10% PL:

In 10% PL we can see the same handshake as we've seen in 0% PL:

Apply a display filter ... <Ctrl-/>						
No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	127.0.0.1	127.0.0.1	TCP	76	50762 → 1604 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM=...
2	0.000041348	127.0.0.1	127.0.0.1	TCP	76	1604 → 50762 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495...
3	0.000061252	127.0.0.1	127.0.0.1	TCP	68	50762 → 1604 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=34754177...

In this screenshot we can see that the Cubic sends the packets with some loss (TCP Out-Of-Order) and the following Window Updates from the Receiver. The Window Update means that the Receiver has told the Sender to update the receive window. The length of the following packets is updated to chunks of 32 KB:

4	0.000139865	127.0.0.1	127.0.0.1	TCP	32809	50762 → 1604 [ACK] Seq=1 Ack=1 Win=65536 Len=32741 TSval=3475...
5	0.000146378	127.0.0.1	127.0.0.1	TCP	68	1604 → 50762 [ACK] Seq=1 Ack=32742 Win=48640 Len=0 TSval=3475...
6	0.000159717	127.0.0.1	127.0.0.1	TCP	32809	50762 → 1604 [PSH, ACK] Seq=32742 Ack=1 Win=65536 Len=32741 T...
7	0.208487589	127.0.0.1	127.0.0.1	TCP	32809	[TCP Retransmission] 50762 → 1604 [PSH, ACK] Seq=32742 Ack=1 ...
8	0.208503893	127.0.0.1	127.0.0.1	TCP	80	1604 → 50762 [ACK] Seq=1 Ack=65483 Win=65536 Len=0 TSval=3475...
9	0.208515407	127.0.0.1	127.0.0.1	TCP	32809	50762 → 1604 [ACK] Seq=65483 Ack=1 Win=65536 Len=32741 TSval=...
10	0.208520268	127.0.0.1	127.0.0.1	TCP	32809	50762 → 1604 [PSH, ACK] Seq=98224 Ack=1 Win=65536 Len=32741 T...
11	0.208533309	127.0.0.1	127.0.0.1	TCP	68	1604 → 50762 [ACK] Seq=1 Ack=98224 Win=48640 Len=0 TSval=3475...
12	0.208670634	127.0.0.1	127.0.0.1	TCP	68	1604 → 50762 [ACK] Seq=1 Ack=130965 Win=65536 Len=0 TSval=347...
13	0.208691125	127.0.0.1	127.0.0.1	TCP	32809	50762 → 1604 [ACK] Seq=130965 Ack=1 Win=65536 Len=32741 TSval...
14	0.208725038	127.0.0.1	127.0.0.1	TCP	68	1604 → 50762 [ACK] Seq=1 Ack=163706 Win=196480 Len=0 TSval=34...
15	0.208737164	127.0.0.1	127.0.0.1	TCP	32809	[TCP Previous segment not captured] 50762 → 1604 [ACK] Seq=19...
16	0.208745550	127.0.0.1	127.0.0.1	TCP	32809	50762 → 1604 [PSH, ACK] Seq=229188 Ack=1 Win=65536 Len=32741 S...
17	0.208754968	127.0.0.1	127.0.0.1	TCP	32809	[TCP Previous segment not captured] 50762 → 1604 [PSH, ACK] S...
18	0.208762888	127.0.0.1	127.0.0.1	TCP	32809	50762 → 1604 [ACK] Seq=327411 Ack=1 Win=65536 Len=32741 TSval...
19	0.208795425	127.0.0.1	127.0.0.1	TCP	80	[TCP Window Update] 1604 → 50762 [ACK] Seq=1 Ack=163706 Win=4...
20	0.208814265	127.0.0.1	127.0.0.1	TCP	32809	[TCP Out-Of-Order] 50762 → 1604 [PSH, ACK] Seq=163706 Ack=1 W...
21	0.208823357	127.0.0.1	127.0.0.1	TCP	32809	50762 → 1604 [PSH, ACK] Seq=360152 Ack=1 Win=65536 Len=32741 ...
22	0.208838857	127.0.0.1	127.0.0.1	TCP	88	[TCP Window Update] 1604 → 50762 [ACK] Seq=1 Ack=163706 Win=5...
23	0.208851062	127.0.0.1	127.0.0.1	TCP	32809	[TCP Out-Of-Order] 50762 → 1604 [ACK] Seq=261929 Ack=1 Win=65...
24	0.208863629	127.0.0.1	127.0.0.1	TCP	88	[TCP Window Update] 1604 → 50762 [ACK] Seq=1 Ack=163706 Win=7...
25	0.208876884	127.0.0.1	127.0.0.1	TCP	32809	50762 → 1604 [ACK] Seq=392893 Ack=1 Win=65536 Len=32741 TSval...
26	0.208891722	127.0.0.1	127.0.0.1	TCP	80	1604 → 50762 [ACK] Seq=1 Ack=261929 Win=851328 Len=0 TSval=34...

TCP Dup ACK, TCP Retransmission:

52	0.209104371	127.0.0.1	127.0.0.1	TCP	80	[TCP Dup ACK 46#1] 1604 → 50762 [ACK] Seq=1 Ack=753044 Win=17...
53	0.209313279	127.0.0.1	127.0.0.1	TCP	88	[TCP Window Update] 1604 → 50762 [ACK] Seq=1 Ack=753044 Win=1...
54	0.209333890	127.0.0.1	127.0.0.1	TCP	32809	[TCP Retransmission] 50762 → 1604 [PSH, ACK] Seq=949490 Ack=1...


TCP Dup ACK is a type of packet sent by a TCP Receiver to acknowledge receipt of a duplicate packet. This can happen when a packet is lost in transit and the Sender retransmits it. The Receiver receives both the original packet and the retransmitted packet, and sends a Dup ACK to the Sender to let it know that it received a duplicate packet. The Sender will adapt accordingly, by reducing the congestion window or retransmitting lost packets. Dup ACKs can also be used as a mechanism for fast retransmit, which is a technique used to improve the recovery of lost packets in TCP.

Next we can see the last segment of the first half of file and the authentication being sent and received. The CC algorithm will change to Reno:

58	0.416950108	127.0.0.1	127.0.0.1	TCP	939	50762 → 1604 [PSH, ACK] Seq=1047713 Ack=1 Win=65536 Len=871 T...
59	0.416955854	127.0.0.1	127.0.0.1	TCP	68	1604 → 50762 [ACK] Seq=1 Ack=1048584 Win=2263808 Len=0 TSval=...
60	0.417080596	127.0.0.1	127.0.0.1	TCP	72	1604 → 50762 [PSH, ACK] Seq=1 Ack=1048584 Win=2263808 Len=4 T...
61	0.417085172	127.0.0.1	127.0.0.1	TCP	68	50762 → 1604 [ACK] Seq=1048584 Ack=5 Win=65536 Len=0 TSval=34...
62	0.417214548	127.0.0.1	127.0.0.1	TCP	65551	50762 → 1604 [ACK] Seq=1048584 Ack=5 Win=65536 Len=65483 TSva...

Like in the 0% PL, we can see that the Reno sends data in bigger chunks, while the Cubic lets it stay at the same level.

245	19.522850986	127.0.0.1	127.0.0.1	TCP	69	[TCP Previous segment not captured] 50762 → 1604 [PSH, ACK] S...
246	19.522936163	127.0.0.1	127.0.0.1	TCP	68	[TCP ACKed unseen segment] 1604 → 50762 [FIN, ACK] Seq=21 Ack...
247	19.563938014	127.0.0.1	127.0.0.1	TCP	68	50762 → 1604 [ACK] Seq=10485836 Ack=22 Win=65536 Len=0 TSval=...
248	20.729009971	127.0.0.1	127.0.0.1	TCP	68	50762 → 1604 [FIN, ACK] Seq=10485836 Ack=22 Win=65536 Len=0 T...
249	20.729050551	127.0.0.1	127.0.0.1	TCP	68	1604 → 50762 [ACK] Seq=22 Ack=10485837 Win=3145728 Len=0 TSva...



In the last packets we can see something very interesting – In 245 the Sender sends an exit command and a notice that the previous segment was not captured.

In 246 the Receiver gets the notice that the Sender wants to quit the program together with the notice regarding the missing segment. Before being able to fix the issue, the Sender closes the socket and the connection is lost. Why does this happen? Maybe Wireshark doesn't read every single packet? Maybe it's our clicking too fast to send again before the used threads and processes finished their jobs? Our assumption is that this happens because of the immediate closing of the connection by both sides, without waiting for a full confirmation that the entire file was received.

4. 15% PL:

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	127.0.0.1	127.0.0.1	TCP	76	50196 → 1604 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM=...
2	0.000012197	127.0.0.1	127.0.0.1	TCP	76	1604 → 50196 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495...
3	0.000022355	127.0.0.1	127.0.0.1	TCP	68	50196 → 1604 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=34755136...

As usual the 3-Way Hand-Shake in the first 3 packets.

8	0.000661281	127.0.0.1	127.0.0.1	TCP	32809	[TCP Previous segment not captured] 50196 → 1604 [PSH, ACK] S...
9	0.000668295	127.0.0.1	127.0.0.1	TCP	80	[TCP Dup ACK 7#1] 1604 → 50196 [ACK] Seq=1 Ack=65483 Win=6553...
10	0.016906239	127.0.0.1	127.0.0.1	TCP	32809	[TCP Retransmission] 50196 → 1604 [ACK] Seq=65483 Ack=1 Win=6...
11	0.016942429	127.0.0.1	127.0.0.1	TCP	68	1604 → 50196 [ACK] Seq=1 Ack=130965 Win=128 Len=0 TSval=34755...
12	0.016967327	127.0.0.1	127.0.0.1	TCP	68	[TCP Window Update] 1604 → 50196 [ACK] Seq=1 Ack=130965 Win=6...

In this screenshot we can see that the ACK(11) that refers to (8) comes right after the Retransmission packet that also refers to (8). in (12) we can see that the receiver tells the sender to decrease the window because of the dup ACK in (9) as part of Cubic mechanism.

62	0.225179279	127.0.0.1	127.0.0.1	TCP	939	50196 → 1604 [PSH, ACK] Seq=1047713 Ack=1 Win=65536 Len=871 T...
63	0.225355048	127.0.0.1	127.0.0.1	TCP	72	1604 → 50196 [PSH, ACK] Seq=1 Ack=1048584 Win=2992000 Len=4 T...
64	0.225376166	127.0.0.1	127.0.0.1	TCP	68	50196 → 1604 [ACK] Seq=1048584 Ack=5 Win=65536 Len=0 TSval=34...
65	0.225606974	127.0.0.1	127.0.0.1	TCP	65551	50196 → 1604 [ACK] Seq=1048584 Ack=5 Win=65536 Len=65483 TSva...

In packet (4) we can see the middle of the file right after that the authentication and then we will change the Congestion Control algorithm to RENO which force the chunks get bigger (64KB) and Cubic will let them stay like that in the next time we will send the file again.

294	13.118366821	127.0.0.1	127.0.0.1	TCP	68	1604 → 50196 [FIN, ACK] Seq=21 Ack=10485836 Win=3145728 Len=0...
295	13.160815555	127.0.0.1	127.0.0.1	TCP	68	50196 → 1604 [ACK] Seq=10485836 Ack=22 Win=65536 Len=0 TSval=...
296	14.119438948	127.0.0.1	127.0.0.1	TCP	68	50196 → 1604 [FIN, ACK] Seq=10485836 Ack=22 Win=65536 Len=0 T...
297	14.119471743	127.0.0.1	127.0.0.1	TCP	68	1604 → 50196 [ACK] Seq=22 Ack=10485837 Win=3145728 Len=0 TSva...

In the level of packet loss we cant see a real difference except the number of packets that send overall increased which looks quite intuitive, is it everything? Maybe theres more information in the last PL precentage.

5.20% PL:

1 0.000000000	127.0.0.1	127.0.0.1	TCP	76 37962 → 1604 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM=...
2 1.013390836	127.0.0.1	127.0.0.1	TCP	76 [TCP Retransmission] 37962 → 1604 [SYN] Seq=0 Win=65495 Len=0...
3 1.013477034	127.0.0.1	127.0.0.1	TCP	76 1604 → 37962 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495...
4 1.013510971	127.0.0.1	127.0.0.1	TCP	68 37962 → 1604 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=34757089...
5 1.013639882	127.0.0.1	127.0.0.1	TCP	32809 37962 → 1604 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=3475...

In this Handshake we can see something really wierd, the first SYN didnt came through so the Cubic Retransmission comes into action and sends a new SYN to the receiver.(time: 1.013Sec)

21 7.394520825	127.0.0.1	127.0.0.1	TCP	68 1604 → 37962 [ACK] Seq=1 Ack=261929 Win=128 Len=0 TSval=34757...
22 9.910569530	127.0.0.1	127.0.0.1	TCP	196 [TCP Window Full] 37962 → 1604 [PSH, ACK] Seq=261929 Ack=1 Wi...
23 9.910591890	127.0.0.1	127.0.0.1	TCP	68 1604 → 37962 [ACK] Seq=1 Ack=262057 Win=65408 Len=0 TSval=347...

TCP Window Full is a condition that can occur when the receiver receive window is full and is unable to receive any more data. The receive window is a buffer in the receiver memory that stores the data that is being transmitted by the sender.

When the receive window is full, the receiver sends an ACK packet to the sender indicating that it is unable to receive any more data. The sender will then stop sending data until the receiver's receive window is able to accept more data.

TCP Window Full can occur for a variety of reasons, such as when the receiver is unable to process the data fast enough or when the network connection is congested and the data is being transmitted too slowly. In these cases, the sender will need to wait for the receiver's receive window to become available before it can continue sending data.

335 53.821344540	127.0.0.1	127.0.0.1	TCP	69 37962 → 1604 [PSH, ACK] Seq=10485835 Ack=21 Win=65536 Len=1 T...
336 53.821355345	127.0.0.1	127.0.0.1	TCP	68 1604 → 37962 [ACK] Seq=21 Ack=10485836 Win=3145088 Len=0 TSva...
337 53.821475622	127.0.0.1	127.0.0.1	TCP	68 1604 → 37962 [FIN, ACK] Seq=21 Ack=10485836 Win=3145728 Len=0...
338 53.865616215	127.0.0.1	127.0.0.1	TCP	68 37962 → 1604 [ACK] Seq=10485836 Ack=22 Win=65536 Len=0 TSval=...
339 54.822021041	127.0.0.1	127.0.0.1	TCP	68 37962 → 1604 [FIN, ACK] Seq=10485836 Ack=22 Win=65536 Len=0 T...
340 55.093457516	127.0.0.1	127.0.0.1	TCP	68 [TCP Retransmission] 37962 → 1604 [FIN, ACK] Seq=10485836 Ack...
341 55.093516867	127.0.0.1	127.0.0.1	TCP	68 1604 → 37962 [ACK] Seq=22 Ack=10485837 Win=3145728 Len=0 TSva...

The last FIN is quite weird – the last FIN should be sent and it gets lost so the TCP Retransmission acts so fast that the client didn't close the port yet and still ACKed the FIN.

Conclusions

1. Comments:

- a. **Disclaimer** – the following conclusions are of limited **certainty**, due to the fact that there are many factors which can affect the results:
 - 1) Network Bandwidth – we found considerable differences in the results when running the program on a home network than from when we ran it on the university network.
 - 2) Random PL Tool – in order to simulate PL in the network, we used the Linux TC tool and set it to different percentages of PL. However, the randomization of the TC tool can differ in its affect on packets being sent, so that for one "send" operation there will be severe PL, where as another "send" will go through smoothly. This, of course, had an impact on the results shown.
 - 3) Collected Data – following the above, in order to reach a definitive conclusion, the test must be run in different network settings and receival times must be recorded for a significantly greater number of reruns.
- b. **Synced Algorithms** – we decided to focus our exploration on synced CC algorithms, meaning that the both the Sender and the Receiver run with the same CC algorithms throughout the test-runs. This was in order to learn about the Reno and Cubic CC algorithms themselves, rather than try to interpret the results of out-of-sync messaging between them.
- c. In the receiver side there's no change of the congestion control, the only difference is the flow control and in this assignment we're required to investigate only the congestion, and not the flow.

	0% loss	10% loss	15% loss	20% loss
Cubic time	9143 microsec	211,152 microsec	118,862 microsec	4,106,282 microsec
Reno time	450 microsec	91,839 microsec	170,484 microsec	1,197,228 microsec

The results are not definitive. We recorded a few runs of the program, and the results differed – in some cases Reno was faster, and in some Cubic outperformed Reno. Following is what we learned about the differences between the two CC algorithms:



2. **Cubic (2006):**

There are several factors that contribute to the performance of the TCP Cubic congestion control algorithm:

- a. Slow start: TCP Cubic uses a slow start mechanism to gradually increase the transmission rate after a period of congestion. This helps to avoid overloading the network with too much data and allows the transmission rate to gradually ramp up as the network conditions permit.
- b. Fast convergence: TCP Cubic is designed to achieve fast convergence to the maximum transmission rate, which means it can quickly find a transmission rate that maximizes the utilization of the available bandwidth.
- c. Adaptability: TCP Cubic is designed to adapt to changing network conditions and to maintain a stable transmission rate in the presence of fluctuating congestion levels.
- d. Fairness: TCP Cubic is designed to be fair to other flows in the network, meaning it does not attempt to monopolize the available bandwidth.
- e. Robustness: TCP Cubic is designed to be robust in the presence of packet loss, meaning it can maintain a stable transmission rate even in the presence of significant packet loss.

3. **Reno (1990):**

There are several factors that contribute to the performance of the TCP Reno congestion control algorithm:

- a. Slow start: Like described in Cubic section, TCP Reno uses slow start mechanism to gradually increase the transmission rate after a period of congestion. This helps to avoid overloading the network with too much data and allows the transmission rate to gradually ramp up as the network conditions permit.
- b. Congestion avoidance: TCP Reno uses a congestion avoidance mechanism to reduce the transmission rate when congestion is detected. This helps to prevent further network congestion and allows the transmission rate to gradually ramp up again as the network conditions permit,
- c. Fast retransmit: TCP Reno includes a fast retransmit mechanism that allows it to quickly retransmit lost packets in the event of packet loss. This helps to maintain a stable retransmission rate and to recover from packet loss more quickly.
- d. Fast recovery: TCP Reno includes a fast recovery mechanism that allows it to quickly recover from periods of congestion. This helps to maintain a stable transmission rate to quickly adapt to changes in network conditions.

Summary

Our Experience

In this project we faced a number of challenges – implementing a client-server TCP data exchange in C language, researching and implementing CC algorithms, creating a data set to present our findings, and following and proving our findings using Wireshark captures.

Our process for the assignment was to first complete the code which will allow a flawless execution of the Receiver-Sender interaction, and then run the program multiple times in order to understand the flow and see it play out on Wireshark. When running the program and analyzing the data transfer and TCP exchange, we were able to recognize and fix mistakes in our code.

In retrospect, we would have liked to spend more of our time creating a more diverse data set, running the same program with the same text file on different networks in order to see the changes play out.

This was an important learning experience – both in the challenges and our gaining experience on how to approach network research (or any research for that matter) in the future.

Technical – What We Learned

“Reno” and “Cubic” are both congestion control algorithms that are used in TCP.

These algorithms help to improve the efficiency of data transfer over a network by adjusting the rate at which data is sent based on the available bandwidth and the congestion of the network.

In general, Cubic is considered to be faster than Reno due to the fact that Cubic is a more recent congestion control algorithm that was developed to address some of the limitations of Reno. Cubic uses a more sophisticated cubic-function approach to determining the sending rate, and a number of other features, which allow it to adapt more quickly to changing network conditions and achieve higher performance, as opposed to Reno which uses a linear approach to increase the sending rate and thus can be slow in adapting to network changes.

Bottom Line – Which Algorithm is Better?

It seemed to us that in most of the runs the Cubic algorithm performed better, especially with severe packet loss. However, we decided to show results that do not necessarily point to the same conclusions, as an example of how much different factors (and some chance) can change the outcome.