**TCP – Congestion Control Algorithms Network programming in C**

Authors:

Maor Saadon 318532421

Duvi Amiram 305677494

**Contents**

**1 System Characterization 2**

1.1 System Overview

1.1.1 About the System  
1.1.2 CC – Algorithm   
1.1.3 How to Install and Run the Program

1.2 System Functionality

1.2.1 Code Description   
1.2.2 Flowchart  
1.2.3 Functions  
1.2.4 Interfaces

**2 Research findings**

2.1 Wireshark

2.1.1 0% lost  
2.1.2 10% lost  
2.1.3 15% lost  
2.1.4 20% lost

2.2 Average sending times comparisons

2.2.1 0% lost  
2.2.2 10% lost  
2.2.3 15% lost  
2.2.4 20% lost

2.3 Diagram

2.4 Conclusions

2.5 Bibliography

**1 System Characterization**

**1.1 System Overview**

**1.1.1 About the System**

‘TCP – Congestion Control Algorithms Network programming’ - like the  
name of the assignment our program reflects the communication between sender and receiver according to two congestion control algorithms (cubic, reno) while sending the file.

The Sender will send a message in this message will be a file, with at least 1MB size of a file, and the Receiver will receive it and measure the time it took for his program to receive the file. The receiver doesn’t really care about saving the file itself (or its content). He just cares about the Data-Frame that he gets.

The file will be sent in two parts first half and second half, each half will be sent according to one of the CC algorithms we learned in the lectures.   
The first half will be sent according to CUBIC algorithms and second half will be sent according to RENO algorithm. By this, and because the size of the two half is the same, we can see based on the result which algorithm is better for this present purpose.

In addition, we will use packet lost tools on Linux which is known as TC. We can see the communication and the packet lost by using Wireshark.

**1.1.2 CC – Algorithm**

At first let’s explain about congestion control, congestion on the internet happens when the computers send more packets to a particular link than that link can handle. If a link is receiving more packets than can fit on the wire, it will begin queueing up those packets, and if the link’s queues fill up, the link will begin dropping those packets.  
Therefore we have the CC - Algorithm. In this program we use the cubic and reno CC – Algorithm.

RENO:

The traditional approaches to congestion control in TCP, named Tahoe and Reno, operate by increasing the congestion window size, which you can think of as the “rate” at which senders send packets, exponentially until some threshold is reached. After that threshold is reached, they begin increasing linearly.

Chart, line chart, histogram

Description automatically generatedOnce they experience drops, they will aggressively drop the window size, and then enter “slow start” mode again, where they increase exponentially until they hit the threshold. Once the threshold is hit, they will increase linearly again.

The approaches Tahoe and Reno were developed in the late 80s and 90s, and worked then, but the internet has changed a lot since. Notably, there now exist longer, and higher bandwidth networks (long, fat networks). To fully take advantage of these networks, TCP senders need to send with much higher congestion windows.

Since Tahoe & Reno grow linearly after crossing the slow start threshold, if a lot more bandwidths become available on a link, it’ll take a long time for these algorithms to “discover” that available bandwidth.

CUBIC:

It turns out that to achieve better performance on “long, fat networks”, these congestion control algorithms need to do a better job of “exploring” for more bandwidth.

However, we don’t want to overload a network, or aggressively steal away available bandwidth from other senders on the same network.

It turns out that there’s a mathematical function for growing the congestion window that satisfies both constraints, the cubic function:

Chart, line chart

Description automatically generated

Observe a very powerful property of a cubic function: that as x is lower, t function grows very quickly, and then slows down as it approaches a particular point (the inflection point), and then after crossing that point begins growing quickly again.

Therefore, it’s took advantage of this property when designing a congestion control algorithm:

If the congestion window is grown as a cubic function of time since the last packet drop, and the inflection point is set to be the size of the congestion window at the last drop, the window has the following behavior:

1. Start growing fast
2. As the algorithm approaches the window at the last drop, the congestion window begins growing more slowly
3. If the congestion window gets to the point at which drops occurred the last time, and does not experience a drop, begin growing slowly again, but then increase more quickly

The concave portion of this, where the window begins growing quickly but then slows down, and the convex portion of this can be thought of two different phases. During the concave phase, the congestion window is catching up to where a packet was lost last time and slows down its growth to be fair to traditional TCP senders. If the algorithm gets there without experiencing drops, it can move into an “exploratory” phase, in which it grows quickly to discover newly available bandwidth.

**1.1.3 How to Install and Run the Program**

**1.2 System Functionality  
1.2.1 Code Description**

**IMPORTANT!**

We assume that the input is correct and that the user will not repeat sending the file more than 1000 times

Order of operations: sender's side.

File handling:

1. Text

   Description automatically generatedOpen the file in order to perform certain functions on it.
2. Text

   Description automatically generatedcalculate and save file size (used for sending the files in two parts of equal size).
3. store file contents in char array - preparing for sending

Text

Description automatically generated

1. Text

   Description automatically generatedclose the file

Socket handling:

1. Text

   Description automatically generatedOpen a socket from which to send the file.
2. Text

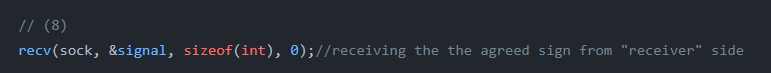
   Description automatically generatedconnect said socket to server side using the known server address. (in this case: local host).

Preparing for sending:

1. Text

   Description automatically generatedinitialize agreed upon sign ("signal") which would indicate that the receiving side (server side) acquired the file size information. Then, send the file size using send() method (see explanation below).

1. receive the agreed upon sign ("signal") from the receiver's side.

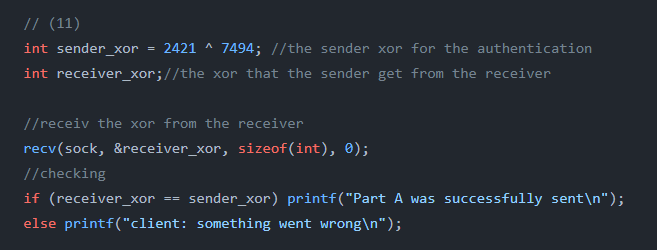


Sending both parts and authenticating:

1. Text

   Description automatically generatedset the CC algorithm to "cubic" using setsockopt() function
2. Text

   Description automatically generatedsend part A of the file (determined by setting the length of data to send to (filesize/2)).
3. initialize certain value (in this case: xor(2421, 7494) – last four digits of our ID) then, receive value from receiver and compare for authentication (determining that we sent the first part of the file to the desired receiver).



1. Text

   Description automatically generatedset the CC algorithm to "reno" using setsockopt() function and repeat step
2. Text

   Description automatically generatedask user if he wants to repeat the sending process or end the connection.
3. repeat process (while loop) as long as the user answers "yes".

Order of operations: receiver's side.

Handle socket

1. Open a server socket and initialize using sockarddr\_in struct (see below functionality of other functions in use). We also set the enable reuse option using setsockopt() function. This allows reusing the same port for receving multiple sending attempts.

Text

Description automatically generated

1. Bind the socket

Text

Description automatically generated

1. Make socket listen to incoming connections.

Text

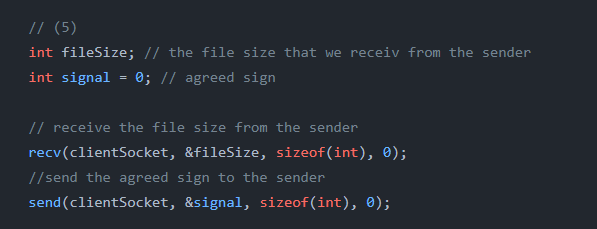
Description automatically generated

1. Accept incoming connections using accept() method, on a new socket (file descriptor).

Text

Description automatically generated

1. Receive file size info from sender and return agreed upon sign which indicates that the file size is known and the file can be sent.



1. Initialize arrays to store the time data of each attempt at receiving the file, and a counter variable to count the number of attempts.

Text

Description automatically generated

Receiving part A:

1. Set the CC algorithm in the listening socket to "cubic" using setsockopt() method (see explanation below) for the retrieval of the first part.

Text

Description automatically generated

1. Initialize buffer with the correct size for receiving the desires part of the file, and initialize a variable to keep track of the number of bytes that were received.

Text

Description automatically generated

1. Calculate time at the beginning of receiving process.

Text

Description automatically generated

1. Receive the bytes sent from "sender" using recv() method (see explanation below. If the connection was lost for some reason than the entire process will be terminated (determined by checking the "running" variable).

Text

Description automatically generated

1. Calculate time at the end of receiving process.

Text

Description automatically generated

1. Send agreed upon sign for authentication xor(2421 ^ 7494).

Text

Description automatically generated

Receiving part B:

1. Changing CC algorithm to Reno in the listening socket, and repeating steps 8 – 11 for the retrieval of part B.

Text

Description automatically generated

1. Close the connection.

Text

Description automatically generated

1. Calculate the total time it took to receive each part.

Text

Description automatically generated

1. Calculate the average receive time for each part using the "counter" variable which indicates the number of times the file was sent.

Text

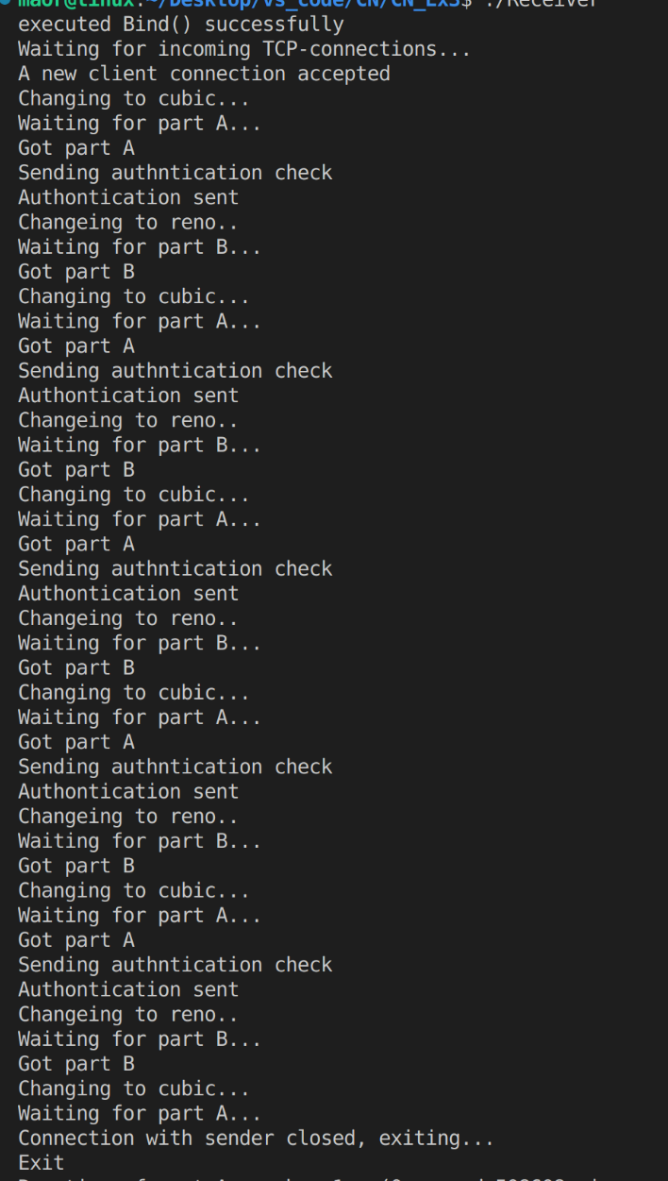
Description automatically generated

**1.2.2 Flowchart**

**1.2.3 Output**

Sending the file 5 times.

**Text

Description automatically generated**

**1.2.4 Functions**

Sender function:

1.To read the file we created we used the following functions:

* stream open functions**-  
  FILE \*fopen(const char \*pathname, const char \*mode);**The fopen() function opens the file whose name is the string pointed to by pathname and associates a stream with it.  
  r - Open text file for reading. The stream is positioned at the beginning of the file.  
  Upon successful completion fopen(), return a FILE pointer. Otherwise, NULL is returned and errno is set to indicate the error.
* Reposition a stream-  
  **int fseek(FILE \*stream, long offset, int whence);**The fseek() function sets the file position indicator for the stream pointed to by stream. The new position, measured in bytes, is obtained by adding offset bytes to the position specified by whence. If whence is set to SEEK\_SET, SEEK\_CUR, or SEEK\_END, the offset is relative to the start of the file, the current position indicator, or end-of-file, respectively.  
  Upon successful completion fseek() return 0.
* Reposition a stream-  
  **long ftell(FILE \*stream);**The ftell() function obtains the current value of the file position indicator for the stream pointed to by stream.  
  Upon successful completion, ftell() returns the current offset. Otherwise,   
  -1 is returned and errno is set to indicate the error.
* Binary stream input/output-  
  **size\_t fread(void \*ptr, size\_t size, size\_t nmemb, FILE \*stream);**The function fread() reads nmemb items of data, each size bytes long, from the stream pointed to by stream, storing them at the location given by ptr.On success, fread() return the number of items read or written. This number equals the number of bytes transferred only when size is 1. If an error occurs, or the end of the file is reached, the return value is a short item count (or zero). The file position indicator for the stream is advanced by the number of bytes successfully read or written.

fread() does not distinguish between end-of-file and error, and callers must use feof(3) and ferror(3) to deter‐mine which occurred.

* Close a stream -   
  **int fclose(FILE \*stream);**  
  The fclose() function flushes the stream pointed to by stream and closes the underlying file descriptor. The behavior of fclose() is undefined if the stream parameter is an illegal pointer, or is a descriptor already passed to a previous invocation of fclose().  
  Upon successful completion, 0 is returned. Otherwise, EOF is returned and errno is set to indicate the error. In either case, any further access (including another call to fclose()) to the stream results in undefined behavior.  
  The fclose() function may also fail and set errno for any of the errors specified for the routines close(2), write(2), or fflush(3).

2. To create a TCP Connection between the sender and receiver we used the following functions:

* Create an endpoint for communication-  
  **int socket(int domain, int type, int protocol);**  
  socket() creates an endpoint for communication and returns a file descriptor that refers to that endpoint. The file descriptor returned by a successful call will be the lowest-numbered file descriptor not currently open for the process.   
  The domain argument specifies a communication domain; this selects the protocol family which will be used for communication:   
  AF\_INET - IPv4 Internet protocols.  
  The socket has the indicated type, which specifies the communication semantics. Currently defined types are:

SOCK\_STREAM - Provides sequenced, reliable, two-way, connection-based byte streams. An out-of-band data transmission mechanism may be supported.  
The protocol specifies a particular protocol to be used with the socket. Normally only a single protocol exists to support a particular socket type within a given protocol family, in which case protocol can be specified as 0. However, it is possible that many protocols may exist, in which case a particular protocol must be specified in this manner. In our case is IPPROTO\_TCP.  
On success, a file descriptor for the new socket is returned. On error, -1 is returned, and errno is set appropriately.

* Fill memory with a constant byte-  
  **void \*memset(void \*s, int c, size\_t n);**The memset() function fills the first n bytes of the memory area pointed to by s with the constant byte c.  
  The memset() function returns a pointer to the memory area s.
* Convert values between host and network byte order-  
  **uint16\_t htons(uint16\_t hostshort);**The htons() function converts the unsigned short integer hostshort from host byte order to network byte order.
* Convert IPv4 and IPv6 addresses from text to binary form -   
  **int inet\_pton(int af, const char \*src, void \*dst);**  
  This function converts the character string src into a network address structure in the af address family, then copies the network address structure to dst. The af argument must be either AF\_INET or AF\_INET6. dst is written in network byte order.

In our case the following address families are currently supported:  
AF\_INET src points to a character string containing an IPv4 network address in dotted-decimal format, "ddd.ddd.ddd.ddd", where ddd is a decimal number of up to three digits in the range 0 to 255. The address is converted to a struct in\_addr and copied to dst, which must be sizeof(struct in\_addr) (4) bytes (32bits) long.  
inet\_pton() returns 1 on success (network address was successfully converted). 0 is returned if src does not contain a character string representing a valid network address in the specified address family.   
If af does not contain a valid address family, -1 is returned and errno is set to EAFNOSUPPORT.

* Initiate a connection on a socket-  
  **int connect(int sockfd, const struct sockaddr \*addr, socklen\_t addrlen);**The connect() system call connects the socket referred to by the file descriptor sockfd to the address specified by addr. The addrlen argument specifies the size of addr. The format of the address in addr is determined by the address space of the socket sockfd.

If the socket sockfd is of type SOCK\_DGRAM, then addr is the address to which datagrams are sent by default, and the only address from which datagrams are received. If the socket is of type SOCK\_STREAM or SOCK\_SEQPACKET, this call attempts to make a connection to the socket that is bound to the address specified by addr.  
If the connection or binding succeeds, zero is returned. On error, -1 is returned, and errno is set appropriately.

3. To Send the first and the second part of the file and to check for authentication we used the following functions:

* Send a message on a socket-  
  **ssize\_t send(int sockfd, const void \*buf, size\_t len, int flags);**send() are used to transmit a message to another socket.

The send() call may be used only when the socket is in a connected state.

The argument sockfd is the file descriptor of the sending socket.  
The message is found in buf and has length len.

If the message is too long to pass atomically through the underlying protocol, the error EMSGSIZE is returned, and the message is not transmitted.

No indication of failure to deliver is implicit in a send(). Locally detected errors are indicated by a return value of -1.

When the message does not fit into the send buffer of the socket, send() normally blocks, unless the socket has been placed in nonblocking I/O mode. In nonblocking mode, it would fail with the error EAGAIN or EWOULDBLOCK in this case.   
The flags argument is the bitwise OR of zero or more flags.  
in our case is 0 flag.  
On success, these calls return the number of bytes sent. On error, -1 is returned, and errno is set appropriately.

* Receive a message from a socket -   
  **ssize\_t recv(int sockfd, void \*buf, size\_t len, int flags);**recv() used to receive messages from a socket. They may be used to receive data on both connectionless and connection-oriented sockets. This page first describes common features of all three system calls, and then describes the differences between the calls.  
  If no messages are available at the socket, the receive calls wait for a message to arrive, unless the socket is nonblocking , in which case the value -1 is returned and the external variable errno is set to EA‐GAIN or EWOULDBLOCK. The receive calls normally return any data available, up to the requested amount, rather than waiting for receipt of the full amount requested.  
  These calls return the number of bytes received, or -1 if an error occurred. In the event of an error, errno is set to indicate the error.

When a stream socket peer has performed an orderly shutdown, the return value will be 0 (the traditional "end-of-file" return).

Datagram sockets in various domains (e.g., the UNIX and Internet domains) permit zero-length datagrams. When such a datagram is received, the return value is 0.

The value 0 may also be returned if the requested number of bytes to receive from a stream socket was 0.

4.To change the CC Algorithm -

* Copy a string -   
  **char \*strcpy(char \*dest, const char \*src);**The strcpy() function copies the string pointed to by src, including the terminating null byte ('\0'), to the buffer pointed to by dest. The strings may not overlap, and the destination string dest must be large enough to receive the copy.   
  The strcpy() functions return a pointer to the destination string dest.
* Calculate the length of a string -   
  **size\_t strlen(const char \*s);**

The strlen() function calculates the length of the string pointed to by s, excluding the terminating null byte ('\0'). The strlen() function returns the number of bytes in the string pointed to by s.

* Set options on sockets -  
  **int setsockopt(int sockfd, int level, int optname, const void \*optval,**

**socklen\_t optlen);**setsockopt() manipulate options for the socket referred to by the file descriptor sockfd. Options may exist at multiple protocol levels.   
When manipulating socket options, the level at which the option resides, and the name of the option must be specified. To manipulate options at the sockets API level, level is specified as SOL\_SOCKET. To manipulate options at any other level the proto‐col number of the appropriate protocol controlling the option is supplied.  
The arguments optval and optlen are used to access option values.  
The argument should be nonzero to enable a boolean option, or zero if the option is to be disabled.   
On success, zero is returned for the standard options. On error, -1 is returned, and errno is set appropriately.

5.To Close the TCP connection we used the following functions:

* Close a file descriptor-  
  close() closes a file descriptor, so that it no longer refers to any file and may be reused.

Receiver function:

1. to Create a TCP Connection between the sender and receiver we used the following functions:

* Create an endpoint for communication-  
  **int socket(int domain, int type, int protocol);**(see the explanation above)
* Set options on sockets -  
  **int setsockopt(int sockfd, int level, int optname, const void \*optval,**

**socklen\_t optlen);**(see the explanation above)

* Fill memory with a constant byte-  
  **void \*memset(void \*s, int c, size\_t n);**(see the explanation above)
* Convert values between host and network byte order-  
  **uint16\_t htons(uint16\_t hostshort);**(see the explanation above)
* Bind a name to a socket -   
  When a socket is created with socket(), it exists in a name space (address family) but has no address assigned to it. bind() assigns the address specified by addr to the socket referred to by the file descriptor sockfd. addrlen specifies the size, in bytes, of the address structure pointed to by addr. Traditionally, this operation is called “assigning a name to a socket”.  
  It is normally necessary to assign a local address using bind() before a SOCK\_STREAM socket may receive connections.  
  On success, zero is returned. On error, -1 is returned, and errno is set appropriately.
* Listen for connections on a socket -   
  **int listen(int sockfd, int backlog);**  
  listen() marks the socket referred to by sockfd as a passive socket, that is, as a socket that will be used to accept incoming connection requests using accept().

The sockfd argument is a file descriptor that refers to a socket of type SOCK\_STREAM or SOCK\_SEQPACKET.

The backlog argument defines the maximum length to which the queue of pending connections for sockfd may grow.

If a connection request arrives when the queue is full, the client may receive an error with an indication of ECONNREFUSED or, if the underlying protocol supports retransmission, the request may be ignored so that a later reattempt at connection succeeds.  
On success, zero is returned. On error, -1 is returned, and errno is set appropriately.

2.To get a connection from the sender we used the following functions:

* accept a connection on a socket  
  **int accept(int sockfd, struct sockaddr \*addr, socklen\_t \*addrlen);**  
  The accept() system call is used with connection-based socket types (SOCK\_STREAM, SOCK\_SEQPACKET). It extracts the first connection request on the queue of pending connections for the listening socket, sockfd, creates a new connected socket, and returns a new file descriptor referring to that socket. The newly created socket is not in the listening state. The original socket sockfd is unaffected by this call.  
  The argument sockfd is a socket that has been created with socket(), bound to a local address with bind(), and is listening for connections after a listen().  
  The argument addr is a pointer to a sockaddr structure. This structure is filled in with the address of the peer socket, as known to the communications layer. The exact format of the address returned addr is determined by the socket’s address family. When addr is NULL, nothing is filled in; in this case, addrlen is not used, and should also be NULL.  
  The addrlen argument is a value-result argument: the caller must initialize it to contain the size (in bytes) of the structure pointed to by addr; on return it will contain the actual size of the peer address.  
  The returned address is truncated if the buffer provided is too small; in this case, addrlen will return a value greater than was supplied to the call.   
  If no pending connections are present on the queue, and the socket is not marked as nonblocking, accept() blocks the caller until a connection is present. If the socket is marked nonblocking and no pending connections arenpresent on the queue, accept() fails with the error EAGAIN or EWOULDBLOCK.

3. To receive the first and the second part of the file we used the following functions:

* Receive a message from a socket -   
  **ssize\_t recv(int sockfd, void \*buf, size\_t len, int flags);**

4. To measure the time, it took to receive the first and the second part we used the following functions:

* Get time -  
  **int gettimeofday(struct timeval \*tv, struct timezone \*tz);**The functions gettimeofday() can set the time as well as a timezone.

5. To send back the authentication to the sender we used the following functions:

* Send a message on a socket-  
  **ssize\_t send(int sockfd, const void \*buf, size\_t len, int flags);**

6. To change the CC Algorithm we used the following functions:

* Copy a string -   
  **char \*strcpy(char \*dest, const char \*src);**(see the explanation above)
* Calculate the length of a string -   
  **size\_t strlen(const char \*s);**(see the explanation above)
* Set options on sockets -  
  **int setsockopt(int sockfd, int level, int optname, const void \*optval,**

**socklen\_t optlen);**(see the explanation above)

4.To Close the TCP connection we used the following functions:

* Close a file descriptor-  
  **int close(int fd);**close() closes a file descriptor, so that it no longer refers to any file and may be reused.

**1.2.5 Interfaces**

**2 Research findings**

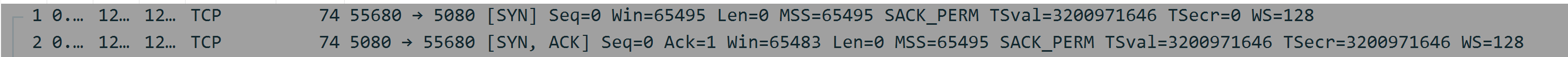
**A picture containing table

Description automatically generated2.1 Wireshark  
2.1.1 0% lost**Sending number 1:

**A picture containing background pattern

Description automatically generated**Sending number 5:

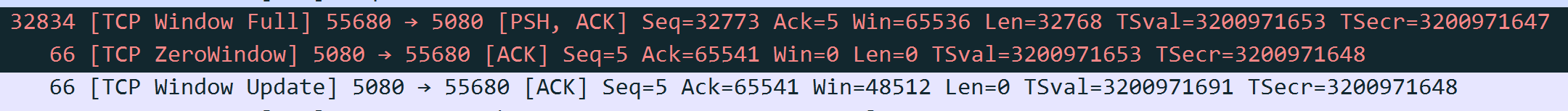
Zoom in:  
Open connection in line 1, 2.



The "[TCP Window Full]" message from Wireshark means that **the system sending this TCP segment has filled up the receive window of the other end with the tcp segment in this packet**. Or put differently: the last received window size of the other end is equal to the length of the TCP segment in this packet

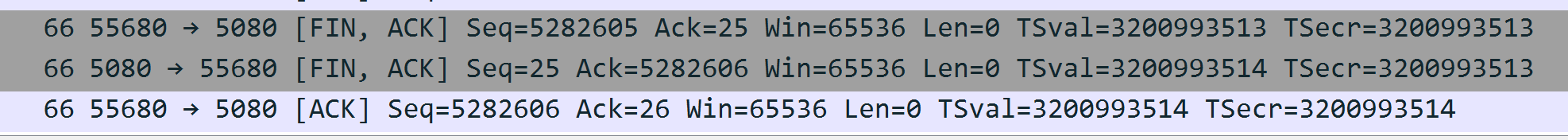
Zero Window means that **the receiver of the packets waves a "white flag" towards the sender, telling it to stop sending because there is no more buffer space for incoming packets**. This is in almost all cases a sign of the receiver being too slow to process the incoming packets in time.

A packet marked "TCP Window Update" simply **indicates that the sender's TCP receive buffer space has increased**. Look at the previous packet from the sender - note the Window Size value in the TCP header.



Checking for authentication in line 46, we can see that the receiver sent it after he get the first half of the file, and that the packet length of the is 4 (int)



close the socket 406, 407, 408

**2.1.2 10% lost**Sending number 1:

**Graphical user interface

Description automatically generated with medium confidence**

Sending number 5:

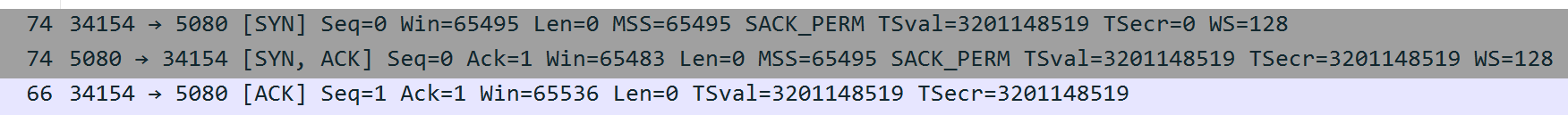
A picture containing background pattern

Description automatically generated

Zoom in:

Now we will see more packets that gets lost.

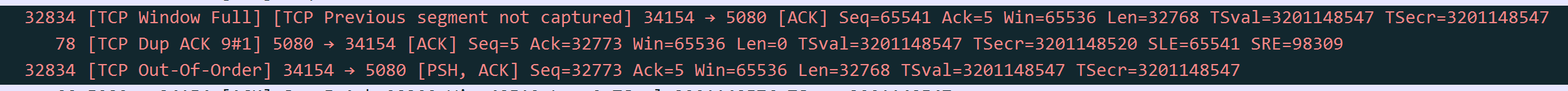
Open connection



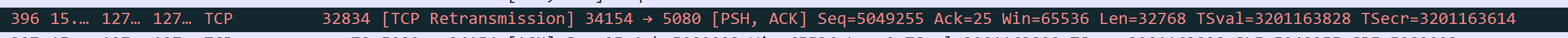
Lost packets:  
**"tcp previous segment not captured"** is an expert message created by Wireshark when it didn't see a packet that should have been in the trace; this warning was previously called "tcp previous segment lost".

**Dup ACK** means that you capture at the source of the data (not the receiving side). That is quite normal if the packet loss occurs somewhere in the path to the receiver.

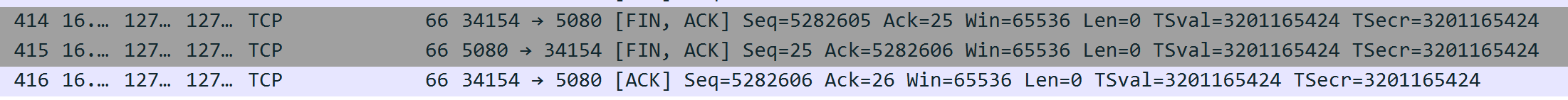
**TCP out-of-order** means that particular frame was received in a different order from which it was sent (after a later packet in the sequence). It is not generally a problem. It probably indicates there are multiple paths between source and destination - and one travels a through a longer path.



**TCP Retransmission** occurs when the sender retransmits a packet after the expiration of the acknowledgement



Close connection lines 414, 415, 416



**2.1.3 15% lost**

**A picture containing timeline

Description automatically generated**open connection in lines 1 2

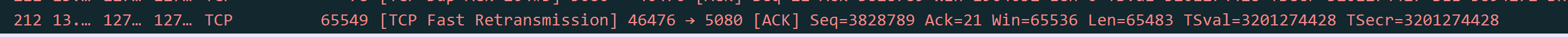
Close the connection in lines 256, 257, 258

**A picture containing graphical user interface

Description automatically generated**

Here we can see one more problem that accrue:

**TCP Fast Retransmission** occurs when the sender retransmits a packet before the expiration of the acknowledgement timer



**2.1.4 20% lost**

open connection in lines 1 2

**A picture containing table

Description automatically generated**

Close the connection in lines 342, 343, 344

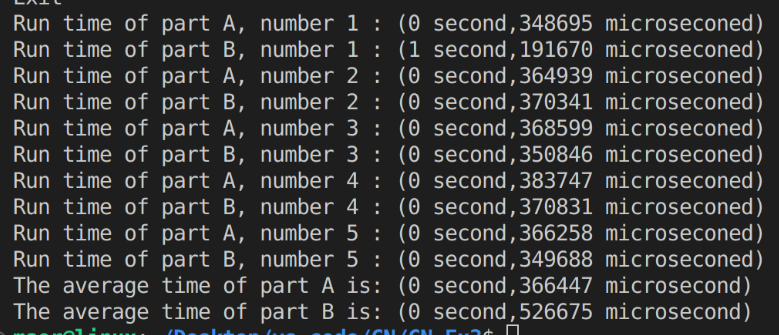
**Table

Description automatically generated**

**Text

Description automatically generated2.2 Average sending times comparisons  
2.2.1 0% lost**

we can see in this picture that the average time of sending part A according to CUBIC algorithm and the average time of sending part B according to RENO algorithm is very tight because we have 0% of packets, so the difference is minor.

**2.2.2 10% lost**

we can see in this picture that the average time of sending part A according to CUBIC algorithm is smaller than average time of sending part B according to RENO algorithm. Now we have 10% packets lost therefore the gap is bigger than in 0% packets lost.

**2.2.3 15% lost**

**A screen shot of a computer

Description automatically generated with low confidence**

We can see the difference of the average time

**2.2.4 20% lost**

**Text

Description automatically generated**

We can see the difference of the average time

**2.3 Diagram**

**2.4 Conclusions**

We can understand from the previous pages that sending the first half of  
the file according to CUBIC CC – Algorithm is faster than sending the second half of the file according to RENO CC – Algorithm.

This can be seen in all five sending of the file, and this is the reason we send five times -> to be assured that our results are reliable.

According to the diagram: the more packets we have that are lost, then the gap between the times of sending first half to sending the second half increases. Also, we can see that the more packets we have that are lost, then the time to send the file is getting longer.

In Wireshark’s recording we can see the difference between the 0% lost to 10% lost to 15% lost and to 20% lost. The more losses we have the more ‘black packets’ appear in Wireshark.

In addition, it is possible to see in Wireshark’s recording the principles that we learned in class about the TCP algorithm actualize, such as:

Increasing the receiver's window when it is full and decreasing the window when it reaches the thresholdץ

And we can see in the receiver's ACK messages, the principle of Cumulativeness, which means that the receiver sends the last ACK that he received, which contains all the ACK before it.

**2.5 Bibliography**

Cubic and Reno CC – Algorithm  
<https://squidarth.com/rc/programming/networking/2018/08/01/congestion-cubic.html>

How to calculate time  
<https://www.youtube.com/watch?v=cunJcNgtxMk&feature=youtu.be>

About the functions  
‘man’ command on vs terminal

For many things

[ChatGPT: Optimizing Language Models for Dialogue (openai.com)](https://openai.com/blog/chatgpt/)

https://www.google.com/search?gs\_ssp=eJzj4tTP1TcwMU02T1JgNGB0YPBiS8\_PT89JBQBASQXT&q=google&oq=googlr&aqs=chrome.1.69i57j46i10i131i199i433i465i512j0i10i131i433i512l4j69i60j69i65.3825j0j4&sourceid=chrome&ie=UTF-8