318848413	325092781

Exercise 3

Table Of Contents

- This is an hyper-linked supported table.

Subjects

About This Exercise

In this exercise, we are implementing and getting familiar with TCP connection and socket creating in the C programming language.
We will send messages between 2 end point (Similarly to Server - Client architecture).
These message will be sent over with packet lost tool with each round of testing being at different % number of packet loss.
Also, we will change frequency the CC algorithm (Congestion Control) upon which our connection will be based and our messages will be sent.

Few main points:

- All testing were done on Ubuntu 22.04LTS.
- The testings include: WireShark monitoring and terminal executing programs.
- The file we used is called “TextFIle.txt”, and it contains 2,097,148 chars, making its size 2,097,148 bytes (over 1MB).
  Thus, the arrays used to hold the file would be of size 2,097,149 (+1 for “\0”).
- Rarely, since I have an M1 chip, things will run fast enough and our time will show 0.000… but that’s only because we don’t use high precision method of printing.
  Using %.*e in printf() will solve it but makes reading time difficult.
  In this code, I use %.12f to print 12 number after the dot.
- In our code, we chose to send every first half in Cubic CC and second half in Reno CC.
- In our code, we used port number 8080, which is usually used for web servers.
- We noticed too late that the calculation of the average for the whole file was wrong (we didn’t need to divide by 2). However, this doesn’t affect the code or any of our testings.

Files included:

- ☑ Sender.c - Code file for the sender part in C.
- ☑ Receiver.c - Code file for the receiver part in C.
- ☑ TextFile.txt - The file we use to send over our TCP connection.
- ☑ 0% Loss.pcapng - Wireshark documentation when 0% loss is on.
- ☑ 10% Loss.pcapng - Wireshark documentation when 10% loss is on.
- ☑ 15% Loss.pcapng - Wireshark documentation when 15% loss is on.
- ☑ 20% Loss.pcapng - Wireshark documentation when 20% loss is on.
- ☑ Makefile - File to compile the Sender and Receiver to executable files.

Abstract

This research study discusses the detail overview in developing and creating a client-server based communication using socket programming in C computing environment.

This research also deals with congestion control in TCP connections and how to modify it to the our preferences at each end of the connection.
Since it is TCP, we use TCP segments to demonstrate the concepts of socket programming and its communication.

Socket programming style, C libraries, and time measuring are also considered in the research stage. The communications between client server processes using socket mechanism were mainly analyzed using also the “Wireshark” program.

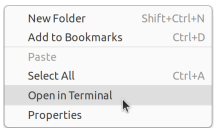
The main objective of this research study is to demonstrate the principles and concepts behind socket programming as well as getting to know the different CC algorithm and make conclusions about their benefits, downsides and different use and implementations.

Instructions To Run Program

To make it easy for testing and checking the code in this submission,
we are adding instructions on how anyone approaching this should run the included files.

Step 1: Compile with Makefile.

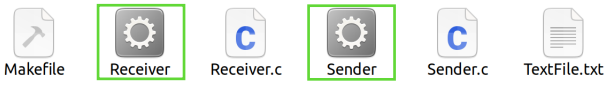
We added a “Makefile” to the files included in this exercise.
On ubuntu, open the terminal in directory of the files.
One way of doing so is right clicking on the folder and selecting “open in terminal”:



Now that the terminal is opened, run the command “make” or “make all” to compile:

```
xdz@xdz-QEMU-Virtual-Machine:~/Downloads/Networking_Ex3$ make
gcc      Sender.c    -o Sender
gcc      Receiver.c    -o Receiver
```

You should now see the compiled files “Sender” and “Receiver”.



- Note that every change that is made in “Receiver.c” or “Sender.c” requires to compile the files again.

Step 2: Run the Receiver and Sender.

Open 2 terminals in the directory of the files (one for Receiver and other for sender).
First, run Receiver using the command “./Receiver”:

```
xdz@xdz-QEMU-Virtual-Machine:~/Downloads/Networking_Ex3$ ./Receiver
(=) Socket created successfully.
(=) Binding was successful!
(=) Waiting for incoming TCP-connections...
```

Now, run Sender using the command “./Sender”:

```
xdz@xdz-QEMU-Virtual-Machine:~/Downloads/Networking_Ex3$ ./Sender
(=) File opened successfully.
(=) Socket created successfully.
(=) Connection with server established.

(+) Set CC algorithm to: cubic
(+) Sent the first half of the message.
(+) Authentication was successful...
(+) Set CC algorithm to: reno
(+) Sent the second half of the message.

Do you want to send file? Enter Y for Yes or N for No.
```

- Note that you must first run receiver. Otherwise, error will occur.

Step 3: Testing program.

When first running Sender, he will automatically send the file one time, and after that will ask if to send file again. Enter ’N’ for closing connection and exiting. Otherwise, to send again, Enter any key (yes, it says ‘Y’ but any other key will also work).

- Note that exiting program without safely closing the connection can cause an error next time since the port is still in use. You can fix that by changing defined port in code and recompiling

Step 4: Testing on Wireshark.

We included 4 pcapng files which can be opened in Wireshark and monitored.

Methods Used In Sender.c

- To explain better what our methods do, we give them the parameters from the code.
- Green - Name of methods. Pink - Parameters. Blue - Defined parameters.

Methods we created :

`size_t send_message(char message[], int socketFD)` – Method meant to send through the socket "socketFD" half of the file starting from the address which "message" pointer holds.

Methods for file reading:

`__sFILE fopen(const char* FILE_NAME, const char* "r")` – Opens the file whose name is the string "FILE_NAME" and associates a stream with it. Also, it sets mode to reading ("r") which means it returns a pointer to the start of the file. Otherwise, null is returned.

`size_t fread(void* message, size_t sizeof(char), size_t FILE_SIZE, __sFILE* file_pointer)` – Method reads every char from the stream pointed to by "file_pointer" till reaching "FILE_SIZE", and stores the data at the location given by "message".

`size_t fclose(__sFILE* file_pointer)` – Method dissociated the "file_pointer" stream from its underlying file, or in other word, closes the stream and flushes the buffers.

Methods for creating socket:

`size_t socket(int AF_INET, int SOCK_STREAM, int 0)` – Method associates "AF_INET" family address (IPv4) as the communications domain within which communication will take place. The "SOCK_STREAM" specifies the semantics of communication as providing sequenced, reliable, two-way connection based byte streams (TCP). The flag is set to "0" which is default in our course. A -1 is returned if an error occurs, otherwise the return value is a descriptor referencing the socket.

`int inet_pton(int AF_INET, const char* IP_ADDRESS, void* &serverAddress.sin_addr)` – Method converts the "IP_ADDRESS" (in our case 127.0.0.1) to binary (network format). It returns -1 if some error occurred or 0 if the conversion was unsuccessful.

`void memset(void* &serverAddress, int '\0', size_t sizeof(serverAddress)` – Method wipes "serverAddress" struct from garbage and resets all of its values to '\0' .

`int connect(int socketFD, const sockaddr* serverAddress, socklen_t sizeof(serverAddress)` – Method receives the "socketFD" we created and attempts to make a connection to the sever specified by "serverAddress". Upon success, 0 is returned. Otherwise, -1.

Method for changing CC:

`int setsockopt(int socketFD, int IPPROTO_TCP, int TCP_CONGESTION, const void* "CC", socklen_t strlen("CC"))` – Method changes the CC algorithm on "socketFD" to the one specified by "CC" string with size "strlen(CC)".
Upon success, 0 is returned. Otherwise, -1.

Method for receiving authentication:

`ssize_t recv(int socketFD, void* msg, size_t 5, int 0)` – Method to receive message from “socketFD” and store it in “msg”. The xor message we expect is of “5” length. The flag is “0”. Returns the number of bytes received or -1 if an error occurred. For TCP socket, returns 0 If peer closed Its half side of the connection.

`int sprintf(char* xor, const char* “%d”, int ID1^ID2)` – Method to convert ID1^ID2 from int format “%d” to string and store at “xor” array.

`int strncmp(const char* xor, const char* msg, size_t 5)` – Method compares between the strings stores at “xor” and “msg" lexicographically between no more then “5” characters. Since it compares strings, characters that appear after ‘\0’ are not compared.

Method for sending message:

`ssize_t send(int socketFD, const void* message + totalLengthSent, size_t FILE_SIZE/2 - totalLengthSent, int 0)` – Method used to transmit the message starting from “message + totalLengthSent” with length “FILE_SIZE/2 - totalLengthSent” over “socketFD”.
The message is sent to the server which “socketFD” is connected to.
Upon success, returns the number of bytes sent. Otherwise, -1.

Method for closing socket:

`int close(int socketFD)` – Method closes the socket by deleting the file descriptor “socketFD”.

Methods Used In Receiver.c

Methods we created :

`int recv_message(int clientSocket)` – Method meant to receive message from the socket descriptor “clientSocket”. In our case, upon every call, the method responsible for receiving the halves from the sender, re-allocating memory if necessary, changing CC, and sending authentication to sender.

`int recv_half(int clientSocket, int 0/1)` – Method meant to receive a specific half of message from the socket “clientSocket”. It also measures time for every half sent, and store that time in “times” array either in row “0” or “1” (according to which half it is).

Methods for creating socket:

`size_t socket(int AF_INET, int SOCK_STREAM, int 0)` – Method associates “AF_INET” family address (IPv4) as the communications domain within which communication will take place. The “SOCK_STREAM” specifies the semantics of communication as providing sequenced, reliable, two-way connection based byte streams (TCP). The flag is set to “0” which is default in our course. A -1 is returned if an error occurs, otherwise the return value is a descriptor referencing the socket.

`int setsocketopt(int socketFD, int SOL_SOCKET, int SO_REUSEADDR, const void* &enableReuse, socklen_t sizeof(enableReuse))` – Method fixes bind failure when address already in use (bits from “socketFD” in use). Upon success, returns 0. Otherwise, -1.

`void memset(void* &serverAddress, int ‘\0’, size_t sizeof(serverAddress)` – Method wipes “serverAddress” struct from garbage and resets all of its values to ‘\0’ .

`int bind(int socketFD, const sockaddr* &serverAddress, socklen_t sizeof(serverAddress))` – Method binds the address and port specified by “serverAddress” struct of size “`sizeof(serverAddress)`” to socket “socketFD”.
Upon success, 0 is returned. Otherwise, -1.

`int listen(int socketFD, int MAX_CONNECTIONS)` – Method listens for connections on the socket “socketFD”. “MAX_CONNECTIONS” defined as 300 and specifies the maximum length for the queue of pending connections. Upon success, returns 0. Otherwise, -1.

`int accept(int socketFD, sockaddr* &clientAddress, socklen_t sizeof(clientAddress))` – Method accept a connection on the socket “socketFD” from the request queue, and returns a new socket descriptor.
Upon success, returns a non-negative integer that is a descriptor of the accepted socket. Otherwise, -1.

Methods for managing memory:

void *malloc(size_t __size)

`void* malloc(size_t__size size)` – Method allocates ‘size’ bytes of memory and returns a pointer to the allocated memory. Otherwise, in error, NULL pointer is returned.

`void* malloc(void*__ptr times[i], size_t__size size)` – Method re-allocates memory for “times[i]” pointer with new “size” bytes of memory, and returns a pointer to the new re-allocated memory. Otherwise, in error, NULL pointer is returned.

`void free(void* times[I]/times)` – Method deallocates the memory allocation pointed to by “times[I]” or “times”. If one of them is a NULL pointer, no operation is performed.

Method for changing CC:

`int setsockopt(int clientSocket, int IPPROTO_TCP, int TCP_CONGESTION, const void* "CC", socklen_t strlen("CC"))` – Method changes the CC algorithm on "clientSocket" to the one specified by "CC" string with size "strlen(CC)".
Upon success, 0 is returned. Otherwise, -1.

Method for sending authentication:

`int sprintf(char* xor, const char* "%d", int ID1^ID2)` – Method to convert ID1^ID2 from int format "%d" to string and store at "xor" array.

`ssize_t send(int clientSocket, const void* xor, size_t 5, int 0)` – Method used to transmit the xor authentication stored at "xor" array with length "5" over "clientSocket".
The authentication is sent to the sender which "clientSocket" is connected to.
Upon success, returns the number of bytes sent. Otherwise, -1.

Method for receiving message:

`ssize_t recv(int clientSocket, void* buffer, size_t FILE_SIZE/2 - received bytes, int 0)` – Method to receive message from "clientSocket" and store it in "buffer".
The message we expect each time is of "FILE_SIZE/2 -received bytes" length. The flag is "0".
Returns the number of bytes received or -1 if an error occurred.
For TCP socket, returns 0 If peer closed Its half side of the connection.

`void bzero(void* buffer, size_t FILE_SIZE+1)` – Method to write "FILE_SIZE+1" zeroed bytes to the memory pointed by "buffer".

`int gettimeofday(timeval* &begin/end, void* 0)` – Method to get date and time.

`int strcmp(const char* buffer, const char* "exit")` – Method compares between the strings stores at "buffer" and the string "exit" lexicographically.

Method for closing socket:

`int close(int clientSocket)` – Method closes the socket by deleting the file descriptor "clientSocket".

Explaining Sender.c

- Since we already explained about all methods previously, we will only explain what we are doing in the code instead of explaining what the methods do.

Defined Variables:

```
#define PORT 8080
#define IP_ADDRESS "127.0.0.1"
#define FILE_SIZE 2097148
#define FILE_NAME "TextFile.txt"
#define ID1 2781
#define ID2 8413
```

- PORT - We defined as 8080. The port to which we identify the connection.
- IP_ADDRESS - We define as the string “127.0.0.1” (which is the loop-back” address, or in other words, the address on which we communicate locally).
- FILE_SIZE - We define as 2,097,148. This it the byte size of the file we use.
- FILE_NAMW - We define the file name to be “TextFile.txt”.
  This is the also the directory of the file since it’s in the same folder.
- ID1 - 4 last digit of Shalev’s ID.
- ID2 - 4 last digit of Ron’s ID.

Reading File:

We first create a file pointer to point to the start of the file.
We also check if there was error in opening file and print a massage accordingly.
After that, we create an array called “message” of size “SIZE+1” (+1 is for “\0”) and read the file to it.
Eventually, we close the stream for the file pointer.

```
//------------------------------Read File----------------------------
FILE* file_pointer;
file_pointer = fopen( filename: FILE_NAME,  mode: "r");

// Check if we were successful in opening file.
if(file_pointer == NULL) {
    printf("(-) Error in opening file! -> fopen() failed with error code: %d\n", errno);
    exit(EXIT_FAILURE); // Exit program and return EXIT_FAILURE (defined as 1 in stdlib.h).
}
else {
    printf("(=) File opened successfully.\n");
}

// Create array for holding the file.
char message[FILE_SIZE + 1] = {0}; // file size + 1 for the \0.
// Read from file to "message".
fread( ptr: message,  size: sizeof(char),  nitems: FILE_SIZE,  stream: file_pointer);

// Closes the stream. All buffers are flushed.
fclose(file_pointer);
```

Creating TCP Connection:

We first create the socket using the socket() method.
We give it “SOCK_STREAM” to identify the connection as TCP.
The method returns a socket descriptor which we hold in the variable “SocketFD”.
We also check if there was error in creating socket and print a massage accordingly.

After that, we create a struct sockaddr_in called “serverAddress” to hold the details of the connection, meaning the ip address and port.
We then reset it using memset() method and then assign it the ip address (using inet_pton() function to convert the address from IPv4 to network byte order), and the port (using htons() function to convert the port also to network byte order).
After that, we use inet_pton() to convert the address to binary and check if we were successful.

```
//--------------------------------Create TCP Connection----------------------------
// Creates socket named "socketFD". FD for file descriptor.
int socketFD = socket(AF_INET, SOCK_STREAM, 0);

// Check if we were successful in creating socket.
if(socketFD == -1) {
    printf("(-) Could not create socket! -> socket() failed with error code: %d\n", errno);
    exit(EXIT_FAILURE); // Exit program and return EXIT_FAILURE (defined as 1 in stdlib.h).
}
else {
    printf("(=) Socket created successfully.\n");
}

// Create sockaddr_in for IPv4 for holding ip address and port and clean it.
struct sockaddr_in serverAddress;
memset(&serverAddress, '\0', sizeof(serverAddress));

// Assign port and address to "serverAddress".
serverAddress.sin_family = AF_INET;
serverAddress.sin_port = htons(PORT); // Short, network byte order.
serverAddress.sin_addr.s_addr = inet_addr(IP_ADDRESS);

// Convert address to binary.
if (inet_pton(AF_INET, IP_ADDRESS, &serverAddress.sin_addr) <= 0)
{
    printf("(-) Failed to convert IPv4 address to binary! -> inet_pton() failed with error code: %d\n", errno);
    exit(EXIT_FAILURE); // Exit program and return EXIT_FAILURE (defined as 1 in stdlib.h).
}
```

Now, we use the connect() method to try and connect to the receiver (server-side).
After that, we check if the connection was successful or not, and print a massage accordingly.

```
//Create connection with server.
int connection = connect(socketFD, (struct sockaddr*) &serverAddress, sizeof(serverAddress));

// Check if we were successful in connecting with server.
if(connection == -1) {
    printf("(-) Could not connect to server! -> connect() failed with error code: %d\n", errno);
    exit(EXIT_FAILURE); // Exit program and return// EXIT_FAILURE (defined as 1 in stdlib.h).
}
else {
    printf("(=) Connection with server established.\n\n");
}
```

CC Change + Send First Part:

We create a while loop which will allow the user to send the file and repeat process multiple times.
We first change the CC algorithm in sender side to “cubic” using setsockopt() method.
Then we check if the change was successful or not, and print a massage accordingly.
After that, we send the first half of the message using the method we created “send_message()”.

```
//------------------------------Send Message------------------------------
int ans;
while (true) {

    //--------------------------------Change CC To Cubic-----------------------------------------------
    if (setsockopt(socketFD, IPPROTO_TCP, TCP_CONGESTION, "cubic", 5) == -1) {
        printf("(-) Failed to change cc algorithm! -> setsocketopt() failed with error code: %d\n", errno);
    }
    else {
        printf("(+) Set CC algorithm to: cubic\n");
    }

    //---------------------------Send First Half----------------------------
    if (send_message(message, socketFD) == -1) {
        printf("(-) Failed to send first half of message! -> send() failed with error code: %d\n", errno);
    }
    else {
        printf("(+) Sent the first half of the message.\n");
    }
```

send_message():

We create a variable called “totalLengthSent” and initialize it to 0. It will help us keep track of the number of bytes we sent over the connection and make sure each time we don’t send more than half the file size.
We do so using a while loop running as long as (totalLengthSent < FILE_SIZE/2).
Each time, we use send() to send the data to the receiver. In case of an error, we return -1.
Otherwise, we update “totalLengthSent” by adding the bytes sent. In the end, we return 1.

```
// Method for sending halves of message.
size_t send_message(char message[], int socketFD) {
    size_t totalLengthSent = 0; // Variable for keeping track of number of bytes sent.
    while (totalLengthSent < FILE_SIZE/2) {
        ssize_t bytes = send(socketFD, message + totalLengthSent, FILE_SIZE/2 - totalLengthSent, 0);
        if (bytes == -1) {
            return -1;
        }
        totalLengthSent += bytes;
    }
    return 1;
}
```

Check For Authentication:

We first create array named “msg” of size 6 (xor result of size 5 + 1 for ‘\0’)
We then use recv() to receive the xor result from receiver and store in “msg”.
After doing so, we create an array named “xor” and use sprintf to calculate the xor in sender side and store the result in ‘xor’. Then we check if both strings are equal using strncmp() and print accordingly.

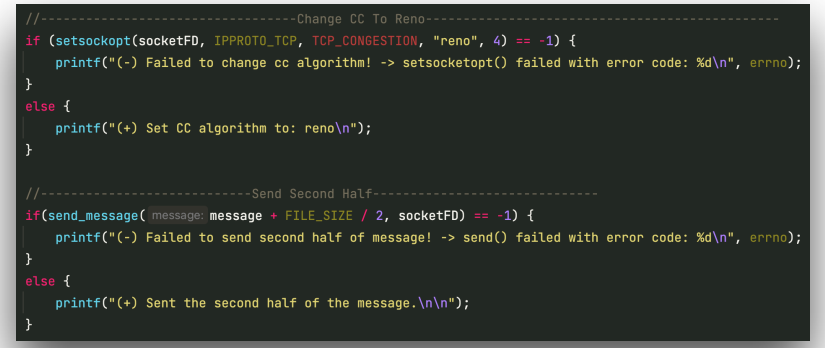
```
//------------------------------Receive Authentication---------------------------------
char msg[6] = {0}; // Array for holding the xor result from receiver. It's of size 5. +1 for the \0.
if (recv(socketFD, msg, 5, 0) <= 0) { // Check if we got an error (-1) or peer closed half side of the socket (0).
    printf("(-) Error in receiving data or peer closed half side of the socket.");
}
char xor[6] = {0}; // Array of holding the xor result. It's of size 5. +1 for the \0.
sprintf(xor, "%d", ID1 ^ ID2); // Convert int (xor result) to string and put in "xor" array.
if (strncmp(xor, msg, 5) == 0) {
    printf("(+) Authentication was successful...\n");
} else {
    printf("(-) Authentication failed...\n");
}
```

CC Change + Send Second Part:

We now change the CC algorithm from “cubic” to “reno” using setsockopt() method.

Then we check if the change was successful or not, and print a massage accordingly.

After that, we send the second half of the message using the method we created “send_message()”.



```
//--------------------------------Change CC To Reno-------------------------------------------
if (setsockopt(socketFD, IPPROTO_TCP, TCP_CONGESTION, "reno", 4) == -1) {
    printf("(-) Failed to change cc algorithm! -> setsocketopt() failed with error code: %d\n", errno);
}
else {
    printf("(+) Set CC algorithm to: reno\n");
}

//----------------------------Send Second Half-----------------------------
if(send_message( message: message + FILE_SIZE / 2, socketFD) == -1) {
    printf("(-) Failed to send second half of message! -> send() failed with error code: %d\n", errno);
}
else {
    printf("(+) Sent the second half of the message.\n\n");
}
```

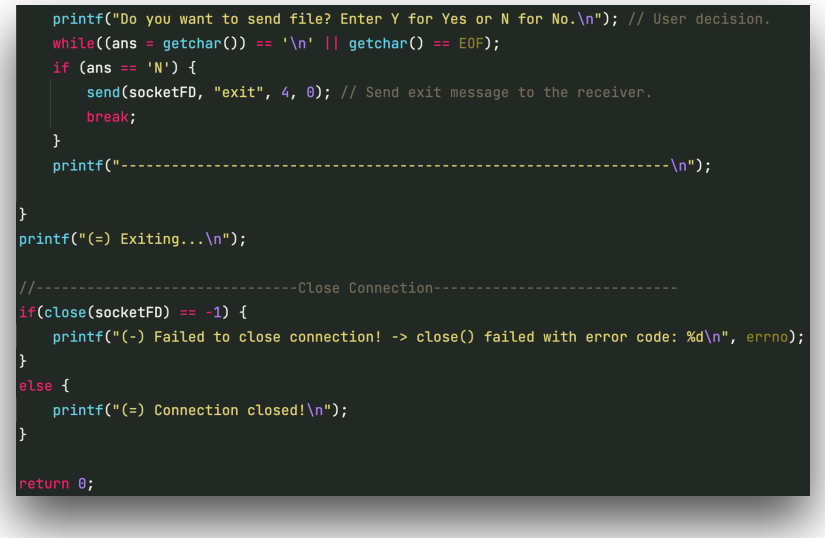
User Decision:

At the end of every file sending, we ask the user if he wants to send the file again and repeat the process or not and close the connection.

We use while loop to wait from a char reading from the user.

If the user enters ’N’, we send an “exit” message to the receiver, break from the loop, close the connection using close() method and print message accordingly. After that, the program returns 0.

Otherwise, if ‘Y’ or any char is entered, we start the loop from the beginning and send the file again.



```
    printf("Do you want to send file? Enter Y for Yes or N for No.\n"); // User decision.
    while((ans = getchar()) == '\n' || getchar() == EOF);
    if (ans == 'N') {
        send(socketFD, "exit", 4, 0); // Send exit message to the receiver.
        break;
    }
    printf("-----------------------------------------------------------------\n");

}
printf("(=) Exiting...\n");

//------------------------------Close Connection----------------------------
if(close(socketFD) == -1) {
    printf("(-) Failed to close connection! -> close() failed with error code: %d\n", errno);
}
else {
    printf("(=) Connection closed!\n");
}

return 0;
```

Explaining Receiver.c

Defined Variables:

```
#define PORT 8080
#define MAX_CONNECTIONS 300
#define FILE_SIZE 2097148
#define ID1 2781
#define ID2 8413
```

- PORT - We defined as 8080. The port to which we identify the connection.
- MAX_CONNECTIONS - We define as 300.
  This is the maximum length for the queue of pending connections.
- FILE_SIZE - We define as 2,097,148. This it the byte size of the file we use.
- ID1 - 4 last digit of Shalev's ID.
- ID2 - 4 last digit of Ron's ID.

Global Variables:

```
int numOfTimes = 0; //  Global variable for number of times saved.
int columns = 5; // Global variable representing number of columns in "times" array.
char buffer[FILE_SIZE + 1]; // Global array for holding the message. His size is FILE_SIZE + 1 for the \0.
double** times = NULL; // Global 2D array for holding times. First row is for first half, Second row is for second half.
```

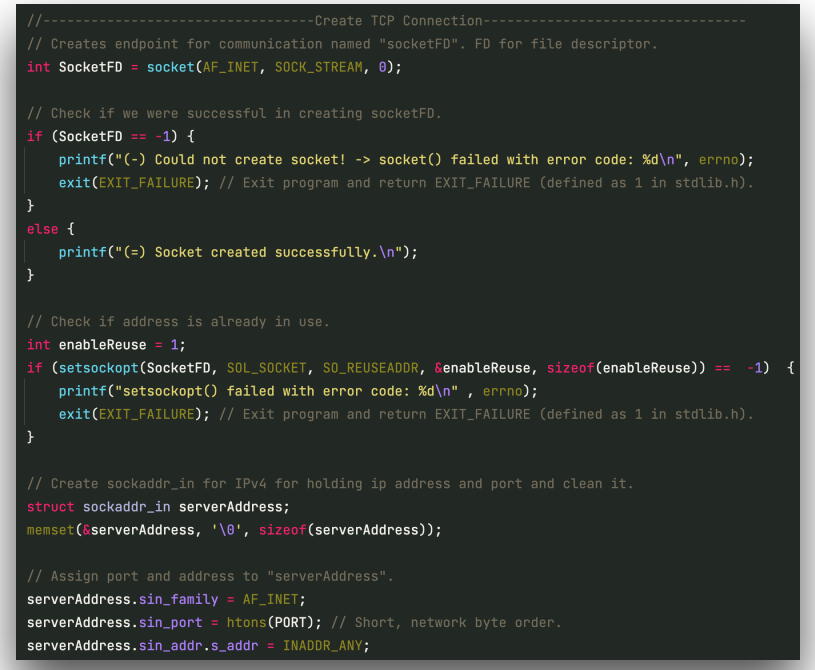
- numOfTimes - Variable that keeps track of number of times we sent file, or in other words, how many rounds where we measured times were.
- columns - Variable that presents the number of columns in "times" array.
  Every column i represents the ith round of sending file.
  We initialize it to 5 since we will send the file at least 5 times.
- buffer[FILE_SIZE + 1] - Array for holding the message.
- times** - 2D array with 2 rows and dynamically changed number of columns to store the times at each round: First row for "cubic"(first half) and second row for "reno" (second row).

Creating TCP Connection:

We first create the socket using the socket() method.
We give it “SOCK_STREAM” to identify the connection as TCP.
The method returns a socket descriptor which we hold in the variable “SocketFD”.
We also check if there was error in creating socket and print a massage accordingly.

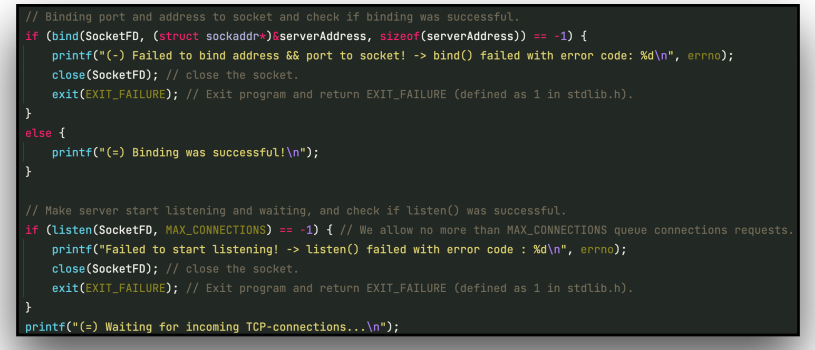
To prevent “address already in use” we use “setsockopt()” method.

After that, we create a struct sockaddr_in called “serverAddress” to hold the details of the connection, meaning the ip address and port.
We then reset it using memset() method and then assign it the ip address (using “INADDR_ANY” since we don’t want to bind the socket to any specific ip but to any one who wants to connect.
In our case, only the sender will want to connect), and the port (using htons() function to convert the port to network byte order).



```
//---------------------------------Create TCP Connection---------------------------------
// Creates endpoint for communication named "socketFD". FD for file descriptor.
int SocketFD = socket(AF_INET, SOCK_STREAM, 0);

// Check if we were successful in creating socketFD.
if (SocketFD == -1) {
    printf("(-) Could not create socket! -> socket() failed with error code: %d\n", errno);
    exit(EXIT_FAILURE); // Exit program and return EXIT_FAILURE (defined as 1 in stdlib.h).
}
else {
    printf("(=) Socket created successfully.\n");
}

// Check if address is already in use.
int enableReuse = 1;
if (setsockopt(SocketFD, SOL_SOCKET, SO_REUSEADDR, &enableReuse, sizeof(enableReuse)) ==  -1)  {
    printf("setsockopt() failed with error code: %d\n" , errno);
    exit(EXIT_FAILURE); // Exit program and return EXIT_FAILURE (defined as 1 in stdlib.h).
}

// Create sockaddr_in for IPv4 for holding ip address and port and clean it.
struct sockaddr_in serverAddress;
memset(&serverAddress, '\0', sizeof(serverAddress));

// Assign port and address to "serverAddress".
serverAddress.sin_family = AF_INET;
serverAddress.sin_port = htons(PORT); // Short, network byte order.
serverAddress.sin_addr.s_addr = INADDR_ANY;
```

We then use the “bind()” method to bind the address and port to the socket.
If an error occurred, we close the socket. Either way, we print a message accordingly.
After that, we use the “listen()” method to start and listen for possible connections to receiver.
Again, if an error occurred, we close the socket. Either way, we print a message accordingly.



```
// Binding port and address to socket and check if binding was successful.
if (bind(SocketFD, (struct sockaddr*)&serverAddress, sizeof(serverAddress)) == -1) {
    printf("(-) Failed to bind address && port to socket! -> bind() failed with error code: %d\n", errno);
    close(SocketFD); // close the socket.
    exit(EXIT_FAILURE); // Exit program and return EXIT_FAILURE (defined as 1 in stdlib.h).
}
else {
    printf("(=) Binding was successful!\n");
}

// Make server start listening and waiting, and check if listen() was successful.
if (listen(SocketFD, MAX_CONNECTIONS) == -1) { // We allow no more than MAX_CONNECTIONS queue connections requests.
    printf("Failed to start listening! -> listen() failed with error code : %d\n", errno);
    close(SocketFD); // close the socket.
    exit(EXIT_FAILURE); // Exit program and return EXIT_FAILURE (defined as 1 in stdlib.h).
}
printf("(=) Waiting for incoming TCP-connections...\n");
```

Get A Connection From The Sender:

We now create a struct sockaddr_in called “clientAddress” to hold the details of the connection Made by the sender client, meaning the ip address and port.
We then reset it using memset() method to wipe it.
After that, we use “accept()” method to accept the TCP connection and hold the socket descriptor for this connection in variable called “clientSocket”.
If we were unsuccessful, we close the sockets. Either way, we print a massage accordingly.

```
//---------------------------------Get TCP Connection From The Sender-------------------------------
// Create sockaddr_in for IPv4 for holding ip address and port of client and cleans it.
struct sockaddr_in clientAddress;
memset(&clientAddress, 0, sizeof(clientAddress));
unsigned int clientAddressLen = sizeof(clientAddress);
int clientSocket = accept(SocketFD, (struct sockaddr*)&clientAddress, &clientAddressLen); // Accept connection.
if (clientSocket == -1) {
    printf("(-) Failed to accept connection. -> accept() failed with error code: %d\n", errno);
    close(SocketFD);
    close(clientSocket);
    exit(EXIT_FAILURE); // Exit program and return EXIT_FAILURE (defined as 1 in stdlib.h).
}
else {
    printf("(=) Connection established.\n\n");
}
```

Allocating Memory For The Times:

We use “malloc()” function to allocate memory for the first 2 rows and then we use it again to allocate memory for the first 5 columns. If an error occurred in allocating memory, we will print it.

```
//---------------------------------Allocate Memory For Times Array---------------------------------
times = (double**) malloc( size: 2 * sizeof(double*));
// Check if malloc was successful.
if (times == NULL) {
    printf("(-) Error in allocating memory!");
}

// Allocate memory for every row.
for (int i = 0; i < 2; i++) {
    times[i] = (double*) malloc( size: columns * sizeof(double));
    // Check if malloc was successful.
    if (times[i] == NULL) {
        printf("(-) Error in allocating memory!");
    }
}
```

Receive Messages From The Sender:

We use while loop to make receiver keep receiving data from the sender as long as the sender wants to send data to the receiver, or in other words, as long as recv_message() doesn’t return -1.

```
// While the user still wants to send file, keep receiving.
while(true) {
    if(recv_message(clientSocket) == -1){
        printf("\nExiting...\n\n");
        break;
    }
}
```

Re-allocate Memory + CC Change + Receiver First Half:

We first check if we need to reallocate memory for “times” array, or in other words, if we need to add columns to our 2D array. This would happen when “numOfTimes” >= “columns”.
Each time, we will add 5 columns to our “times” array.

We now change the CC algorithm in receiver side to “cubic” using setsockopt() method.
We do this to match between the sender’s CC to receiver’s CC,
but it doesn’t matter as we will learn moving forward.
Then we check if the change was successful or not, and print a massage accordingly.
After that, we receive the first half of the message using the method we created “recv_half()”.
If the method returns -1, it means the user wants to exit and so we return -1.
Else, we print that we finished receiving first half and we reset “buffer”.

```
//---------------------------------Receive Messages---------------------------------------
// Method for receiving message from sender.
int recv_message(int clientSocket) {

    //---------------------------------Re-allocate Memory For Times Array-------------------------------
    // Will relocate only if necessary.
    if (numOfTimes >= columns) {
        columns += 5;
        for (int i = 0; i < 2; i++) {
            times[i] = (double *) realloc( ptr: times[i],  size: sizeof(double) * columns);
            // Check if realloc() was successful.
            if (times[i] == NULL) {
                printf("(-) Error in re-allocating memory!");
            }
        }
    }

    //---------------------------------Change CC To Cubic-----------------------------------------------
    if (setsockopt(clientSocket, IPPROTO_TCP, TCP_CONGESTION, "cubic", 5) == -1) {
        printf("(-) Failed to change cc algorithm! -> setsocketopt() failed with error code: %d", errno);
    }
    else {
        printf("(+) Set CC algorithm to: cubic");
    }

    //---------------------------------Receive First Messages---------------------------------
    if (recv_half(clientSocket,  row: 0) == -1) {
        return -1;
    }
    printf("\n--------------------> Finished receiving first half <--------------------\n");
    bzero(buffer, FILE_SIZE + 1); // Clean buffer.
```

Send Authentication:

We first create array named “xor” of size 6 (xor result of size 5 + 1 for ‘\0’)
We then use sprintf to calculate the xor in receiver side and store the result in ‘xor’.
After doing so, we send the the xor result to sender and print if we were successful or not.

```
//---------------------------------Send Authentication-----------------------------------
char xor[6] = {0}; // Array of holding the xor result. It's of size 5. +1 for the \0.
sprintf(xor, "%d", ID1 ^ ID2); // Convert int (xor result) to string and put in "xor" array.
if (send(clientSocket, xor, 5, 0) == -1) { // Send xor result to client.
    printf("(-) Failed to send authentication! -> send() failed with error code: %d\n", errno);
}
else {
    printf("(+) Sent authentication.\n");
}
```

CC Change + Receiver Second Half:

We now change the CC algorithm from “cubic” to “reno” using setsockopt() method.
Then we check if the change was successful or not, and print a massage accordingly.
After that, we receive the second half of the message using the method we created “recv_half()”.
If the method returns -1, it means the user wants to exit and so we return -1.
Else, we print that we finished receiving second half, we reset “buffer” and increase “numOfTimes” by 1 since another round of sending file came to an end. After that, we return 0.

```
//---------------------------------Change CC To Reno-------------------------------------------
if (setsockopt(clientSocket, IPPROTO_TCP, TCP_CONGESTION, "reno", 4) == -1) {
    printf("(-) Failed to change cc algorithm! -> setsocketopt() failed with error code: %d", errno);
}
else {
    printf("(+) Set CC algorithm to: reno");
}

//---------------------------------Receive Second Messages-------------------------------
if(recv_half(clientSocket,  row: 1) == -1) {
    return -1;
}
printf("\n--------------------> Finished receiving second half <--------------------\n\n");
bzero(buffer, FILE_SIZE + 1); // Clean buffer.
numOfTimes++; // Increase "numOfTimes" by 1.
return 0;
```

CC Change + Receiver Second Half:

We first create multiple variables: some will help us measure time (seconds, microsecond, time, begin, end), some will help us know when to start measuring time (startTime) and some will help us keep track of number of bytes received (receivedTotalBytes).
We use a while loop running as long as (totalLengthSent != FILE_SIZE/2) meaning we didn’t got half file or as long as “receivedBytes” isn’t 0 or -1 which means an error or peer closed his socket.
Each time, we clean the buffer and receive data while updating “receivedTotalBytes” using “receiveBytes”. The first time we received anything, we start the timer. At the end of receiving the file, check for an “exit” message, calculate the time and add it to “times”. Eventually, we return 0.

```
int recv_half(int clientSocket, int row) {
    size_t receivedTotalBytes = 0; // Variable for keeping track of number of received bytes.
    double seconds; // Variable for measuring time in seconds.
    double microseconds; // Variable for measuring time in microseconds.
    double time; // Variable for measuring elapsed time.
    bool startTime = false; // Variable for knowing when to start timer.
    struct timeval begin, end; // Struct for measuring time.

    //---------------------------------Receive Half Loop-------------------------------
    while (receivedTotalBytes != FILE_SIZE / 2) {
        bzero(buffer, FILE_SIZE + 1); // Clean buffer.
        ssize_t receivedBytes = recv(clientSocket, buffer, FILE_SIZE / 2 - receivedTotalBytes, 0);
        if (receivedBytes <= 0) { // Break if we got an error (-1) or peer closed half side of the socket (0).
            printf("(-) Error in receiving data or peer closed half side of the socket.");
            break;
        }
        // Start time.
        if (startTime == false) {
            startTime = true;
            gettimeofday(&begin, 0);
        }
        receivedTotalBytes += receivedBytes; // Add the new received bytes to the total bytes received.

//        printf("%s", buffer);

        // Check if we need to exit.
        if (strcmp(buffer, "exit") == 0) {
            return -1;
        }
    }

    //---------------------------------Calculate Time---------------------------------
    gettimeofday(&end, 0); // End time.
    seconds = (double) (end.tv_sec - begin.tv_sec); // Calculate elapsed time in seconds.
    microseconds = (double) (end.tv_usec - begin.tv_usec); // Calculate elapsed time in microseconds.
    time = seconds + microseconds * (1e-6);
    times[row][numOfTimes] = time;

    return 0;
}
```

Print Out The Times:

In case of an “exit” message from the sender, we will print all times for both halves from “times” array.
First half represents the “cubic” CC and the second half represents the “reno” CC.
After that, we create 2 variables: “sum1” for summing all first half’ times and “sum2”
for summing all second half’ times.
Now , we create another 2 variables: “average1” for the average time for the first half and
“average2” for the average time for the second half.
We then print them + print the average time for the whole file.

```
//---------------------------------Print Times --------------------------------
printf("Times in seconds for both halves:\n");
for (int i = 0; i < numOfTimes; i++) {
    printf("%d - First Half: %.12f sec \t | \t Second Half: %.12f sec\t\n", i + 1, times[0][i], times[1][i]);
}

double sum1 = 0.0; // Variable to sum times for the first part.
double sum2 = 0.0; // Variable to sum times for the second part.

for (int i = 0; i < numOfTimes; i++) {
    sum1 += times[0][i];
    sum2 += times[1][i];
}

double average1 = sum1 / numOfTimes; // Average time for the first part.
double average2 = sum2 / numOfTimes; // Average time for the second part.

// Prints averages.
printf("\nAverages:\n");
printf("Average time for the first part: %lf\n", average1);
printf("Average time for the second part: %lf\n", average2);
printf("Average time for the whole file: %lf\n", (average1 + average2) / 2.0);
```

Free Memory:

Since “times” is a dynamic array, and we ended our use with it, we need to free its memory allocation from use in order to avoid memory leak.

```
//---------------------------------Free Allocated Memory---------------------------------
for (int i = 0; i < 2; i++) {
    free(times[i]);
}
free(times);
```

Close Socket:

We are closing the connection using close() method on the “clientSocket”
and print message accordingly.

```
//---------------------------------Close Socket---------------------------------
if(close(clientSocket) == -1) {
    printf("(-) Failed to close connection! -> close() failed with error code: %d\n", errno);
}
else {
    printf("(=) Connection closed!\n");
}
```

Tests On Terminal

0% lost:

Setting the packet loss to 0%:

```
xdz@xdz-QEMU-Virtual-Machine:~/Downloads$ sudo tc qdisc add dev lo root netem loss 0%
```

Sender and Receiver output in Terminal:

Sender

```
xdz@xdz-QEMU-Virtual-Machine:~/Downloads/Networking_Ex3$ ./Sender
(=) File opened successfully.
(=) Socket created successfully.
(=) Connection with server established.

(+) Set CC algorithm to: cubic
(+) Sent the first half of the message.
(+) Authentication was successful...
(+) Set CC algorithm to: reno
(+) Sent the second half of the message.

Do you want to send file? Enter Y for Yes or N for No.
Y
-----------------------------------------------------------------
(+) Set CC algorithm to: cubic
(+) Sent the first half of the message.
(+) Authentication was successful...
(+) Set CC algorithm to: reno
(+) Sent the second half of the message.

Do you want to send file? Enter Y for Yes or N for No.
Y
-----------------------------------------------------------------
(+) Set CC algorithm to: cubic
(+) Sent the first half of the message.
(+) Authentication was successful...
(+) Set CC algorithm to: reno
(+) Sent the second half of the message.

Do you want to send file? Enter Y for Yes or N for No.
Y
-----------------------------------------------------------------
(+) Set CC algorithm to: cubic
(+) Sent the first half of the message.
(+) Authentication was successful...
(+) Set CC algorithm to: reno
(+) Sent the second half of the message.

Do you want to send file? Enter Y for Yes or N for No.
Y
-----------------------------------------------------------------
(+) Set CC algorithm to: cubic
(+) Sent the first half of the message.
(+) Authentication was successful...
(+) Set CC algorithm to: reno
(+) Sent the second half of the message.

Do you want to send file? Enter Y for Yes or N for No.
Y
-----------------------------------------------------------------
(+) Set CC algorithm to: cubic
(+) Sent the first half of the message.
(+) Authentication was successful...
(+) Set CC algorithm to: reno
(+) Sent the second half of the message.

Do you want to send file? Enter Y for Yes or N for No.
N
(=) Exiting...
(=) Connection closed!
```

Receiver

```
(=) Socket created successfully.
(=) Binding was successful!
(=) Waiting for incoming TCP-connections...
(=) Connection established.

(+) Set CC algorithm to: cubic
--------------------> Finished receiving first half <--------------------
(+) Set CC algorithm to: reno
--------------------> Finished receiving second half <--------------------

(+) Set CC algorithm to: cubic
--------------------> Finished receiving first half <--------------------
(+) Set CC algorithm to: reno
--------------------> Finished receiving second half <--------------------

(+) Set CC algorithm to: cubic
--------------------> Finished receiving first half <--------------------
(+) Set CC algorithm to: reno
--------------------> Finished receiving second half <--------------------

(+) Set CC algorithm to: cubic
--------------------> Finished receiving first half <--------------------
(+) Set CC algorithm to: reno
--------------------> Finished receiving second half <--------------------

(+) Set CC algorithm to: cubic
--------------------> Finished receiving first half <--------------------
(+) Set CC algorithm to: reno
--------------------> Finished receiving second half <--------------------

(+) Set CC algorithm to: cubic
--------------------> Finished receiving first half <--------------------
(+) Set CC algorithm to: reno
--------------------> Finished receiving second half <--------------------

(+) Set CC algorithm to: cubic
Exiting...

Times in seconds for both halves:
1 - First Half: 0.000979000000 sec       |       Second Half: 0.000001000000 sec
2 - First Half: 0.000001000000 sec       |       Second Half: 0.000001000000 sec
3 - First Half: 0.000004000000 sec       |       Second Half: 0.000000000000 sec
4 - First Half: 0.000001000000 sec       |       Second Half: 0.000879000000 sec
5 - First Half: 0.000001000000 sec       |       Second Half: 0.000531000000 sec
6 - First Half: 0.000544000000 sec       |       Second Half: 0.000271000000 sec

Averages:
Average time for the first part: 0.000255
Average time for the second part: 0.000280
Average time for the whole file: 0.000268
```

10% lost:

Setting the packet loss to 10%:

```
xdz@xdz-QEMU-Virtual-Machine:~/Downloads/Networking_Ex3$ sudo tc qdisc change dev lo root netem loss 10%
```

Sender and Receiver output in Terminal:

Sender

```
xdz@xdz-QEMU-Virtual-Machine:~/Downloads/Networking_Ex3$ ./Sender
(=) File opened successfully.
(=) Socket created successfully.
(=) Connection with server established.

(+) Set CC algorithm to: cubic
(+) Sent the first half of the message.
(+) Authentication was successful...
(+) Set CC algorithm to: reno
(+) Sent the second half of the message.

Do you want to send file? Enter Y for Yes or N for No.
Y
-------------------------------------------------------------------
(+) Set CC algorithm to: cubic
(+) Sent the first half of the message.
(+) Authentication was successful...
(+) Set CC algorithm to: reno
(+) Sent the second half of the message.

Do you want to send file? Enter Y for Yes or N for No.
Y
-------------------------------------------------------------------
(+) Set CC algorithm to: cubic
(+) Sent the first half of the message.
(+) Authentication was successful...
(+) Set CC algorithm to: reno
(+) Sent the second half of the message.

Do you want to send file? Enter Y for Yes or N for No.
Y
-------------------------------------------------------------------
(+) Set CC algorithm to: cubic
(+) Sent the first half of the message.
(+) Authentication was successful...
(+) Set CC algorithm to: reno
(+) Sent the second half of the message.

Do you want to send file? Enter Y for Yes or N for No.
Y
-------------------------------------------------------------------
(+) Set CC algorithm to: cubic
(+) Sent the first half of the message.
(+) Authentication was successful...
(+) Set CC algorithm to: reno
(+) Sent the second half of the message.

Do you want to send file? Enter Y for Yes or N for No.
N
(=) Exiting...
(=) Connection closed!
```

Receiver

```
xdz@xdz-QEMU-Virtual-Machine:~/Downloads/Networking_Ex3$ ./Receiver
(=) Socket created successfully.
(=) Binding was successful!
(=) Waiting for incoming TCP-connections...
(=) Connection established.

(+) Set CC algorithm to: cubic
-------------------> Finished receiving first half <--------------------
(+) Sent authentication.
(+) Set CC algorithm to: reno
-------------------> Finished receiving second half <--------------------

(+) Set CC algorithm to: cubic
-------------------> Finished receiving first half <--------------------
(+) Sent authentication.
(+) Set CC algorithm to: reno
-------------------> Finished receiving second half <--------------------

(+) Set CC algorithm to: cubic
-------------------> Finished receiving first half <--------------------
(+) Sent authentication.
(+) Set CC algorithm to: reno
-------------------> Finished receiving second half <--------------------

(+) Set CC algorithm to: cubic
-------------------> Finished receiving first half <--------------------
(+) Sent authentication.
(+) Set CC algorithm to: reno
-------------------> Finished receiving second half <--------------------

(+) Set CC algorithm to: cubic
-------------------> Finished receiving first half <--------------------
(+) Sent authentication.
(+) Set CC algorithm to: reno
-------------------> Finished receiving second half <--------------------

(+) Set CC algorithm to: cubic
Exiting...

Times in seconds for both halves:
1 - First Half: 0.000610000000 sec      |       Second Half: 0.212921000000 sec
2 - First Half: 0.000776000000 sec      |       Second Half: 0.001070000000 sec
3 - First Half: 0.018725000000 sec      |       Second Half: 0.000210000000 sec
4 - First Half: 0.000872000000 sec      |       Second Half: 0.000410000000 sec
5 - First Half: 0.000544000000 sec      |       Second Half: 0.060133000000 sec

Averages:
Average time for the first part: 0.004305
Average time for the second part: 0.054949
Average time for the whole file: 0.029627
(=) Connection closed!
```

15% lost:

Setting the packet loss to 15%:

```
xdz@xdz-QEMU-Virtual-Machine:~/Downloads/Networking_Ex3$ sudo tc qdisc change dev lo root netem loss 15%
```

Sender and Receiver output in Terminal:

Sender

```
xdz@xdz-QEMU-Virtual-Machine:~/Downloads/Networking_Ex3$ ./Sender
(=) File opened successfully.
(=) Socket created successfully.
(=) Connection with server established.

(+) Set CC algorithm to: cubic
(+) Sent the first half of the message.
(+) Authentication was successful...
(+) Set CC algorithm to: reno
(+) Sent the second half of the message.

Do you want to send file? Enter Y for Yes or N for No.
Y
-----------------------------------------------------------------
(+) Set CC algorithm to: cubic
(+) Sent the first half of the message.
(+) Authentication was successful...
(+) Set CC algorithm to: reno
(+) Sent the second half of the message.

Do you want to send file? Enter Y for Yes or N for No.
Y
-----------------------------------------------------------------
(+) Set CC algorithm to: cubic
(+) Sent the first half of the message.
(+) Authentication was successful...
(+) Set CC algorithm to: reno
(+) Sent the second half of the message.

Do you want to send file? Enter Y for Yes or N for No.
Y
-----------------------------------------------------------------
(+) Set CC algorithm to: cubic
(+) Sent the first half of the message.
(+) Authentication was successful...
(+) Set CC algorithm to: reno
(+) Sent the second half of the message.

Do you want to send file? Enter Y for Yes or N for No.
Y
-----------------------------------------------------------------
(+) Set CC algorithm to: cubic
(+) Sent the first half of the message.
(+) Authentication was successful...
(+) Set CC algorithm to: reno
(+) Sent the second half of the message.

Do you want to send file? Enter Y for Yes or N for No.
N
(=) Exiting...
(=) Connection closed!
```

Receiver

```
xdz@xdz-QEMU-Virtual-Machine:~/Downloads/Networking_Ex3$ ./Receiver
(=) Socket created successfully.
(=) Binding was successful!
(=) Waiting for incoming TCP-connections...
(=) Connection established.

(+) Set CC algorithm to: cubic
--------------------> Finished receiving first half <--------------------
(+) Sent authentication.
(+) Set CC algorithm to: reno
--------------------> Finished receiving second half <--------------------

(+) Set CC algorithm to: cubic
--------------------> Finished receiving first half <--------------------
(+) Sent authentication.
(+) Set CC algorithm to: reno
--------------------> Finished receiving second half <--------------------

(+) Set CC algorithm to: cubic
--------------------> Finished receiving first half <--------------------
(+) Sent authentication.
(+) Set CC algorithm to: reno
--------------------> Finished receiving second half <--------------------

(+) Set CC algorithm to: cubic
--------------------> Finished receiving first half <--------------------
(+) Sent authentication.
(+) Set CC algorithm to: reno
--------------------> Finished receiving second half <--------------------

(+) Set CC algorithm to: cubic
--------------------> Finished receiving first half <--------------------
(+) Sent authentication.
(+) Set CC algorithm to: reno
--------------------> Finished receiving second half <--------------------

(+) Set CC algorithm to: cubic
Exiting...

Times in seconds for both halves:
1 - First Half: 0.018056000000 sec      |       Second Half: 0.000518000000 sec
2 - First Half: 0.062583000000 sec      |       Second Half: 0.000858000000 sec
3 - First Half: 0.000002000000 sec      |       Second Half: 0.463638000000 sec
4 - First Half: 0.000683000000 sec      |       Second Half: 0.230403000000 sec
5 - First Half: 0.063898000000 sec      |       Second Half: 0.265837000000 sec

Averages:
Average time for the first part: 0.029044
Average time for the second part: 0.192251
Average time for the whole file: 0.110648
(=) Connection closed!
```

20% lost:

Setting the packet loss to 20%:

```
xdz@xdz-QEMU-Virtual-Machine:~/Downloads/Networking_Ex3$ sudo tc qdisc change dev lo root netem loss 20%
```

Sender and Receiver output in Terminal:

Sender

```
xdz@xdz-QEMU-Virtual-Machine:~/Downloads/Networking_Ex3$ ./Sender
(=) File opened successfully.
(=) Socket created successfully.
(=) Connection with server established.

(+) Set CC algorithm to: cubic
(+) Sent the first half of the message.
(+) Authentication was successful...
(+) Set CC algorithm to: reno
(+) Sent the second half of the message.

Do you want to send file? Enter Y for Yes or N for No.
Y
-------------------------------------------------------------------
(+) Set CC algorithm to: cubic
(+) Sent the first half of the message.
(+) Authentication was successful...
(+) Set CC algorithm to: reno
(+) Sent the second half of the message.

Do you want to send file? Enter Y for Yes or N for No.
Y
-------------------------------------------------------------------
(+) Set CC algorithm to: cubic
(+) Sent the first half of the message.
(+) Authentication was successful...
(+) Set CC algorithm to: reno
(+) Sent the second half of the message.

Do you want to send file? Enter Y for Yes or N for No.
Y
-------------------------------------------------------------------
(+) Set CC algorithm to: cubic
(+) Sent the first half of the message.
(+) Authentication was successful...
(+) Set CC algorithm to: reno
(+) Sent the second half of the message.

Do you want to send file? Enter Y for Yes or N for No.
Y
-------------------------------------------------------------------
(+) Set CC algorithm to: cubic
(+) Sent the first half of the message.
(+) Authentication was successful...
(+) Set CC algorithm to: reno
(+) Sent the second half of the message.

Do you want to send file? Enter Y for Yes or N for No.
N
(=) Exiting...
(=) Connection closed!
```

Receiver

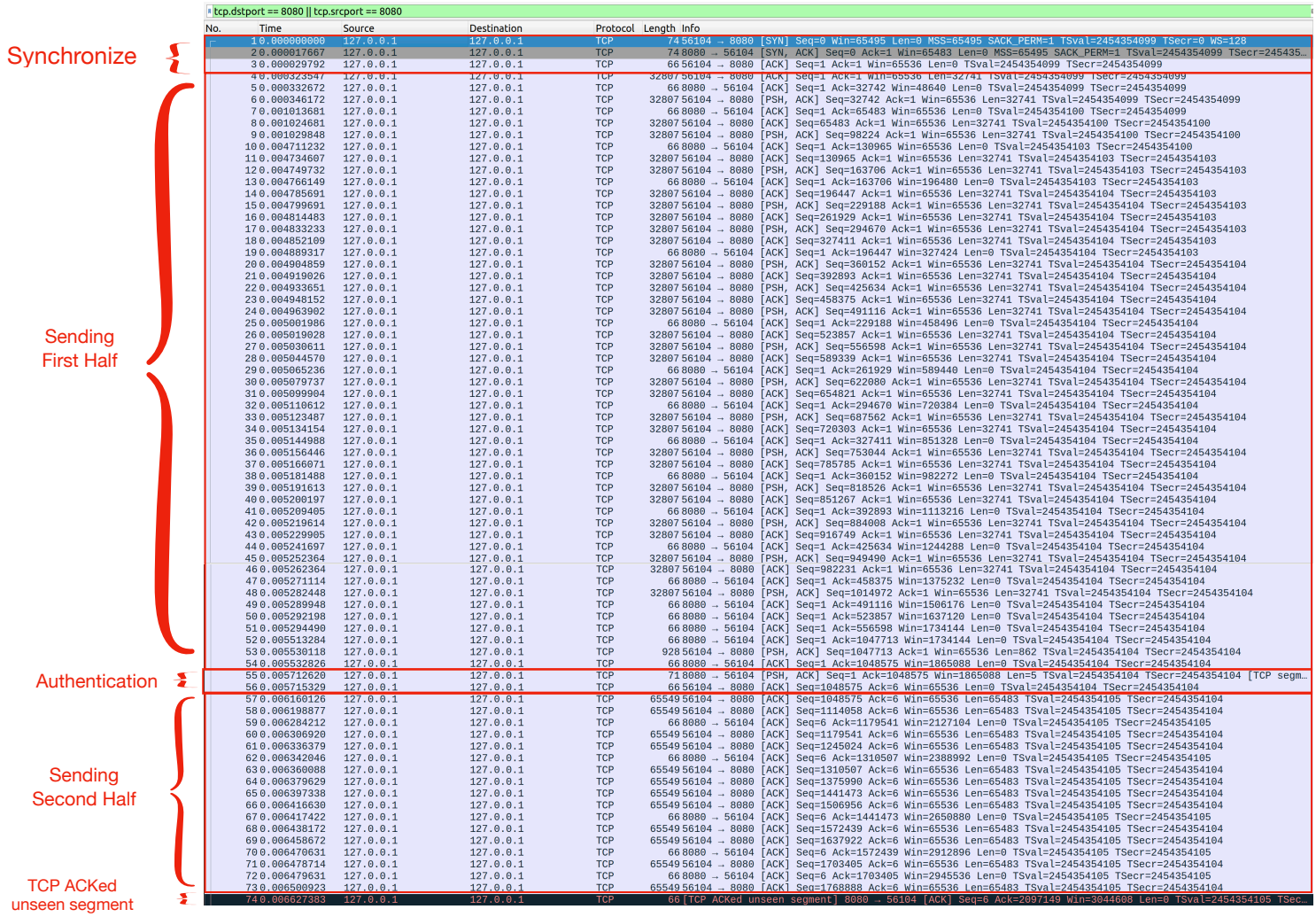
```
xdz@xdz-QEMU-Virtual-Machine:~/Downloads/Networking_Ex3$ ./Receiver
(=) Socket created successfully.
(=) Binding was successful!
(=) Waiting for incoming TCP-connections...
(=) Connection established.

(+) Set CC algorithm to: cubic
--------------------> Finished receiving first half <--------------------
(+) Sent authentication.
(+) Set CC algorithm to: reno
--------------------> Finished receiving second half <--------------------

(+) Set CC algorithm to: cubic
--------------------> Finished receiving first half <--------------------
(+) Sent authentication.
(+) Set CC algorithm to: reno
--------------------> Finished receiving second half <--------------------

(+) Set CC algorithm to: cubic
--------------------> Finished receiving first half <--------------------
(+) Sent authentication.
(+) Set CC algorithm to: reno
--------------------> Finished receiving second half <--------------------

(+) Set CC algorithm to: cubic
--------------------> Finished receiving first half <--------------------
(+) Sent authentication.
(+) Set CC algorithm to: reno
--------------------> Finished receiving second half <--------------------

(+) Set CC algorithm to: cubic
--------------------> Finished receiving first half <--------------------
(+) Sent authentication.
(+) Set CC algorithm to: reno
--------------------> Finished receiving second half <--------------------

(+) Set CC algorithm to: cubic
Exiting...

Times in seconds for both halves:
1 - First Half: 0.655165000000 sec      |       Second Half: 27.240476000000 sec
2 - First Half: 55.292401000000 sec     |       Second Half: 3.633876000000 sec
3 - First Half: 1.100234000000 sec      |       Second Half: 0.000817000000 sec
4 - First Half: 99.162833000000 sec     |       Second Half: 260.107512000000 sec
5 - First Half: 0.000002000000 sec      |       Second Half: 1.932559000000 sec

Averages:
Average time for the first part: 31.242127
Average time for the second part: 58.583048
Average time for the whole file: 44.912588
(=) Connection closed!
```

Tests On WireShark

- Since sending the file over and over will only make the process repeat itself, we will explain it widely only for the first iteration and it would pretty much be identical to the other times we sent the file.
- Switching to higher and higher x% packet loss test may Include different TCP packet like (TCP Retransmission, TCP Dup ACK, TCP Out-Of-Order…) on which we will explain.
- Going higher and higher with x% packet loss will make it more common for those messages to appear so we will try and show it with screenshots.
- Since we seek for all communication done on port 8080, this is what we will filter our Wireshark display by.
- Our Wireshark sometimes confuses and recognize [PSH] messages as [ACK].
  This is easily can be spotted by checking the length of the packet.
- To keep track, the receiver port is 8080 and the sender is assigned a different port number at each connection.

0% lost:

First time sending file:



Synchronize:

Here, the sender is trying to initialize and establish a connection with the receiver.
He sends a [SYN] (Synchronize) message to the receiver, the receiver send him a [SYN, ACK] message to also ask to create a connection with him + acknowledgement for the request he sent.
Eventually, the sender send back [ACK] message to acknowledge he got the receiver request.
This process is called **3 Way-Handshake** and will reappear at the beginning of each pcapng file, since before any communication we need to establish the connection itself between the client and sever.

Sending First Half:

Here, we sent the first file.

- Since the maximum length of a TCP segment is limited to 64kB, each time we send a half of the file, it splits that data into segments of length <= 64kB until it reaches FILE_SIZE/2 which in our case is 1,048,574. Most of the segments here are of length 32,807.
  If we sum them up, we will see it equals exactly 1,048,574, meaning we sent exactly half file.

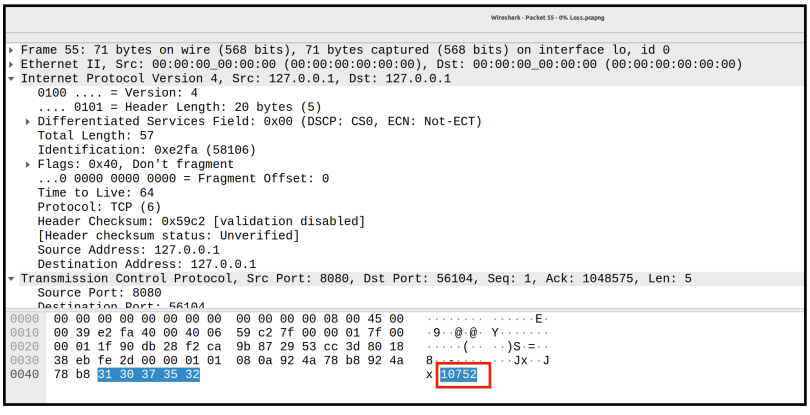
Each time we send data to the receiver, we actually send [PSH, ACK] (Push) message containing the data itself, followed by [ACK] message from the receiver he got the date. For example:

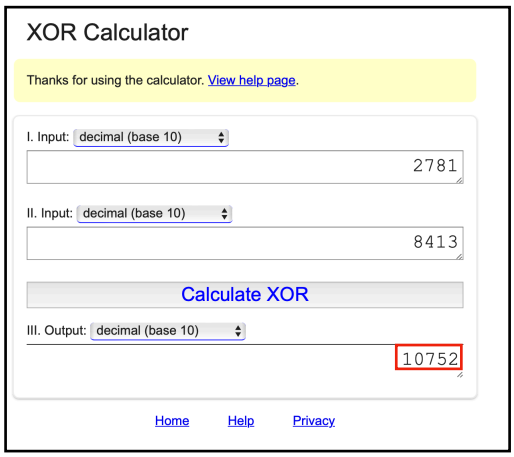
```
48 0.005282448   127.0.0.1       127.0.0.1       TCP    32807 56104 → 8080 [PSH, ACK] Seq=1014972 Ack=1 Win=65536 Len=32741 TSval=2454354104 TSecr=2454354104
49 0.005289948   127.0.0.1       127.0.0.1       TCP       66 8080 → 56104 [ACK] Seq=1 Ack=491116 Win=1506176 Len=0 TSval=2454354104 TSecr=2454354104
```

Notice that some of the ACK's may not come right after each [PSH] message since it all happens fast and the ACK messages may take time to appear.

Authentication:

As we now, the server sends an authentication which is the xor result of the last 4 digits of our ID's.
The receiver (server) sends it through [PSH, ACK] message to the sender (client) and the sender sends back an [ACK] message to acknowledge he got the authentication.
If we open the packet details, we can see the content of the data, meaning the xor result the receiver calculated which is exactly the xor result we would expect (10752):





Sending Second Half:

Here, we sent the second file.

This time, the process is much shorter and requires less TCP packet since now we send at each time a segment of size 65,549 bytes which is approximately 64kB.

TCP Special Messages That Occurred During Communication:

TCP ACKed unseen segment:

```
TCP        66 [TCP ACKed unseen segment] 8080 → 56104 [ACK]
```

This is a way of Wireshark to inform us that we got ACK's that were not seen by Wireshark,
or in other words, they are not in the capture but the data has been received by the sender of the ACK's,
Which In this case is the receiver.
A typical cause for this is a poor capture, meaning the server where we are capturing is not capable of capturing all packets to/from the server.

TCP Previous segment not captured:

```
TCP      65549 [TCP Previous segment not captured] 56104 → 8080 [ACK]
```

This message means that Wireshark is seeing an ACK for a packet it didn't really capture.
This can appear when the computer running Wireshark can't keep up or when we have packet loss.
Some errors like this are to be expected, even in in a normal network.
When we increase the x% of packet loss, we should expect seeing more of these messages.
The reason why we don't see "retransmission" messages here, or in other words, the reason why we don't try to send any of those lost packets are because we run on 0% packet loss, so the CC doesn't act.

Closing Connection:

After finishing sending the file for the last time, the sender enters "N" and so the sender sends "exit message" with [PSH, ACK]. After that, he sends [FIN, ACK] (Finish) message to the receiver (server) To close the connection. This Indicates no more data will be transmitted from the sender.
After that, the receiver send back to the sender the same [FIN, ACK] message.
Lastly, the sender send [ACK] to acknowledge he got the message from sender and then the TCP connection is closed.

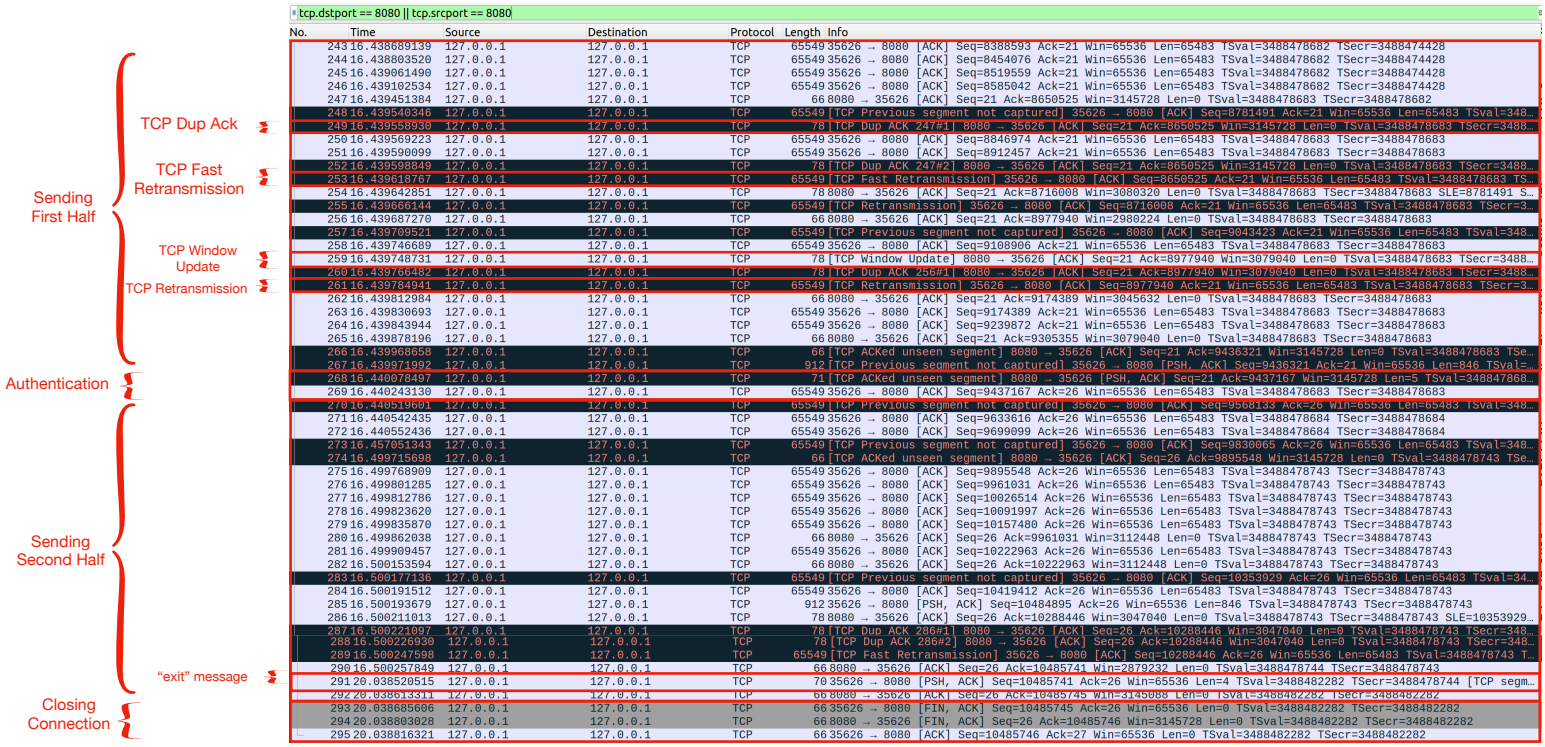
```
271 10.284170737  127.0.0.1  127.0.0.1  TCP  70 56104 → 8080 [PSH, ACK] Seq=12582889 Ack=31 Win=65536 Len=4 TSval=2454364383 TSecr=2454362586 [TCP segm…
272 10.284302822  127.0.0.1  127.0.0.1  TCP  66 56104 → 8080 [FIN, ACK] Seq=12582893 Ack=31 Win=65536 Len=0 TSval=2454364383 TSecr=2454362586
273 10.284503574  127.0.0.1  127.0.0.1  TCP  66 8080 → 56104 [FIN, ACK] Seq=31 Ack=12582894 Win=3144704 Len=0 TSval=2454364383 TSecr=2454364383
274 10.284516200  127.0.0.1  127.0.0.1  TCP  66 56104 → 8080 [ACK] Seq=12582894 Ack=32 Win=65536 Len=0 TSval=2454364383 TSecr=2454364383
```

```
▸ Frame 271: 70 bytes on wire (560 bits), 70 bytes captured (560 bits) on interface lo, id 0
▸ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
▸ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
▾ Transmission Control Protocol, Src Port: 56104, Dst Port: 8080, Seq: 12582889, Ack: 31, Len: 4
    Source Port: 56104
    Destination Port: 8080
    [Stream index: 0]
    [Conversation completeness: Complete, WITH_DATA (31)]
    [TCP Segment Len: 4]
    Sequence Number: 12582889    (relative sequence number)
    Sequence Number (raw): 704891943
    [Next Sequence Number: 12582893    (relative sequence number)]
    Acknowledgment Number: 31    (relative ack number)
    Acknowledgment number (raw): 4073364389
    1000 .... = Header Length: 32 bytes (8)
  ▸ Flags: 0x018 (PSH, ACK)
    Window: 512
    [Calculated window size: 65536]
    [Window size scaling factor: 128]
0000  00 00 00 00 00 00 00 00  00 00 00 00 08 00 45 00   ........ ......E·
0010  00 38 92 db 40 00 40 06  a9 e2 7f 00 00 01 7f 00   ·8··@·@· ........
0020  00 01 db 28 1f 90 2a 03  cc 27 f2 ca 9b a5 80 18   ···(··*· ·'......
0030  02 00 fe 2c 00 00 01 01  08 0a 92 4a a0 df 92 4a   ···,···· ···J···J
0040  99 da 65 78 69 74                                 ··exit
```

10% lost:

Here, we increase the packet loss to 10%.
Thus, we may see more packet get lost,
meaning more segments who are not captured by the wireshark.
We would focus on what happens here when this happens and how the CC helps us recover and resend the lost segments.

Last time sending file:



Since we already explained about the process of sending the message including opening/ closing the connection, sending authentication, TCP ACKed unseen segment message and TCP previous segment not captured message, we will not explain it again.

However, there are a few TCP new messages we haven't seen before in the 0% loss and which we will explain now.
As said before, since now we use 10% packet loss, we will see retransmission of packet which were lost during the communication.

TCP Special Messages That Occurred During Communication:

TCP Retransmission

TCP Retransmission occurs when a packet is lost and its ACK timer expires,
which causes the sender to retransmits the packet.
This is often happens after "previous segment not captured" message.

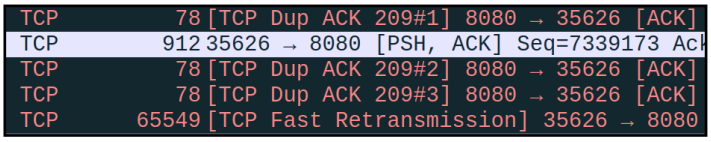
TCP Dup Ack:

This is a stage towards the use of TCP Fast Retransmission.
This happens when one or more packets have been lost in the stream and the connection is attempting to recover. They are a common symptom of packet loss.
A duplicate acknowledgment is sent when a receiver receives out-of-order packets.
When this is recognized, the receiver starts sending duplicate ACKs so the sender would start the fast-retransmit process.

TCP Fast Retransmission

TCP Fast Retransmission occurs right after TCP Dup Ack message.
This is called “Fast Retransmission” since it doesn't wait for the ACK timer to expire in order to retransmit the packet, but it sends it the moment it recognizes some packets didn’t arrived where they should have arrived (since it must be in order).
In other words, we received packets which sequence number are bigger than the ACK packets.
Fast Retransmission should be used upon receipt of 3 duplicate ACKs.

Example:

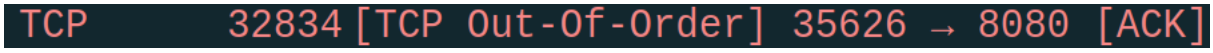


Here we received 3 duplicate ACKs and then used Fast Retransmission.

[TCP Window Update]:

This occurs when the receiver has consumed the received data from the buffer causing the TCP to send a Window Update message to the sender to indicate that there is now more space available in the buffer.

TCP Out-Of-Order:

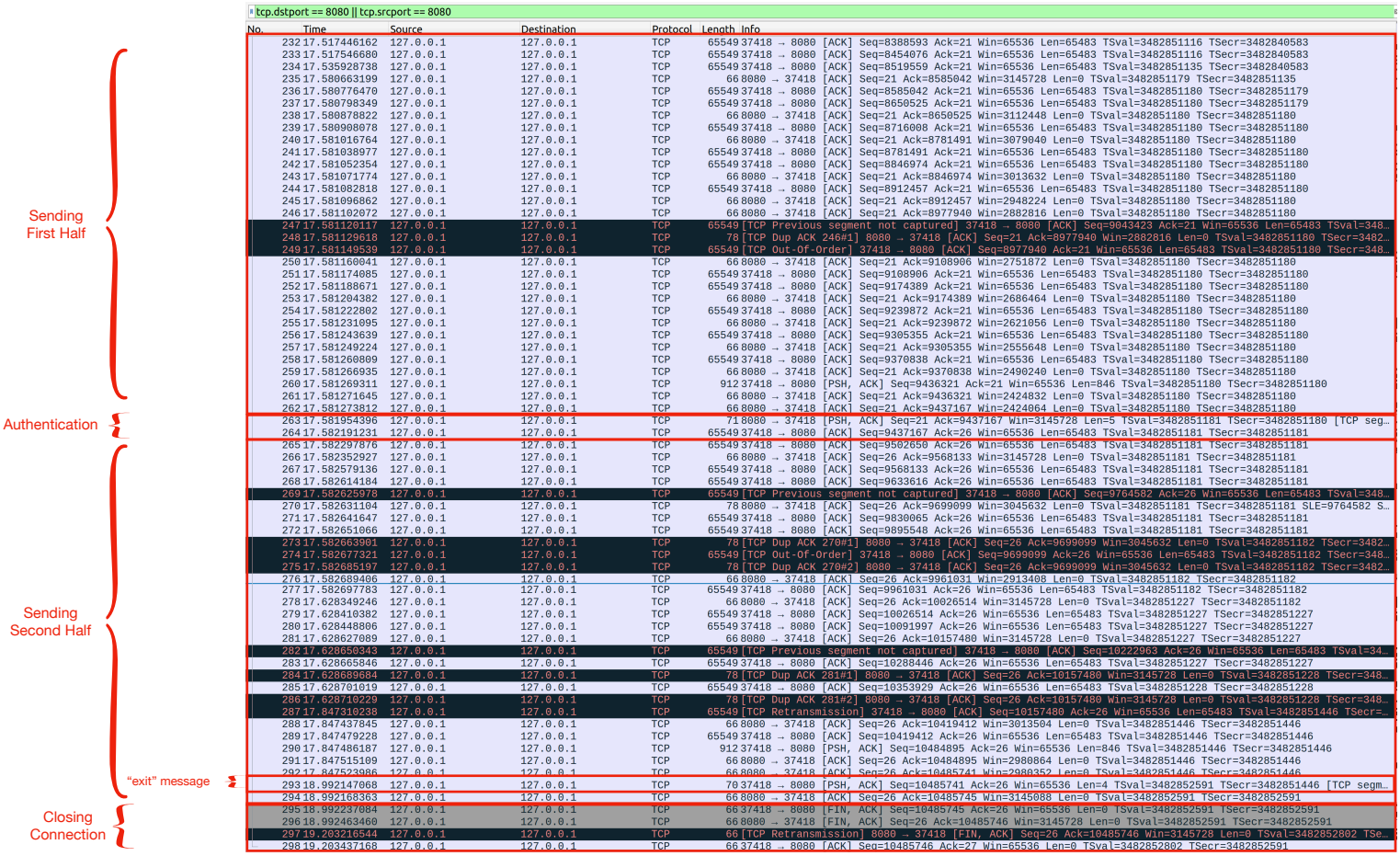


TCP Out-Of-Order occurs when a packet is seen with a sequence number that is lower than the previously received packet on that that connection.

15% lost:

Here, we increase the packet loss to 15%.
Thus, we may see more packet get lost:

Last time sending file:

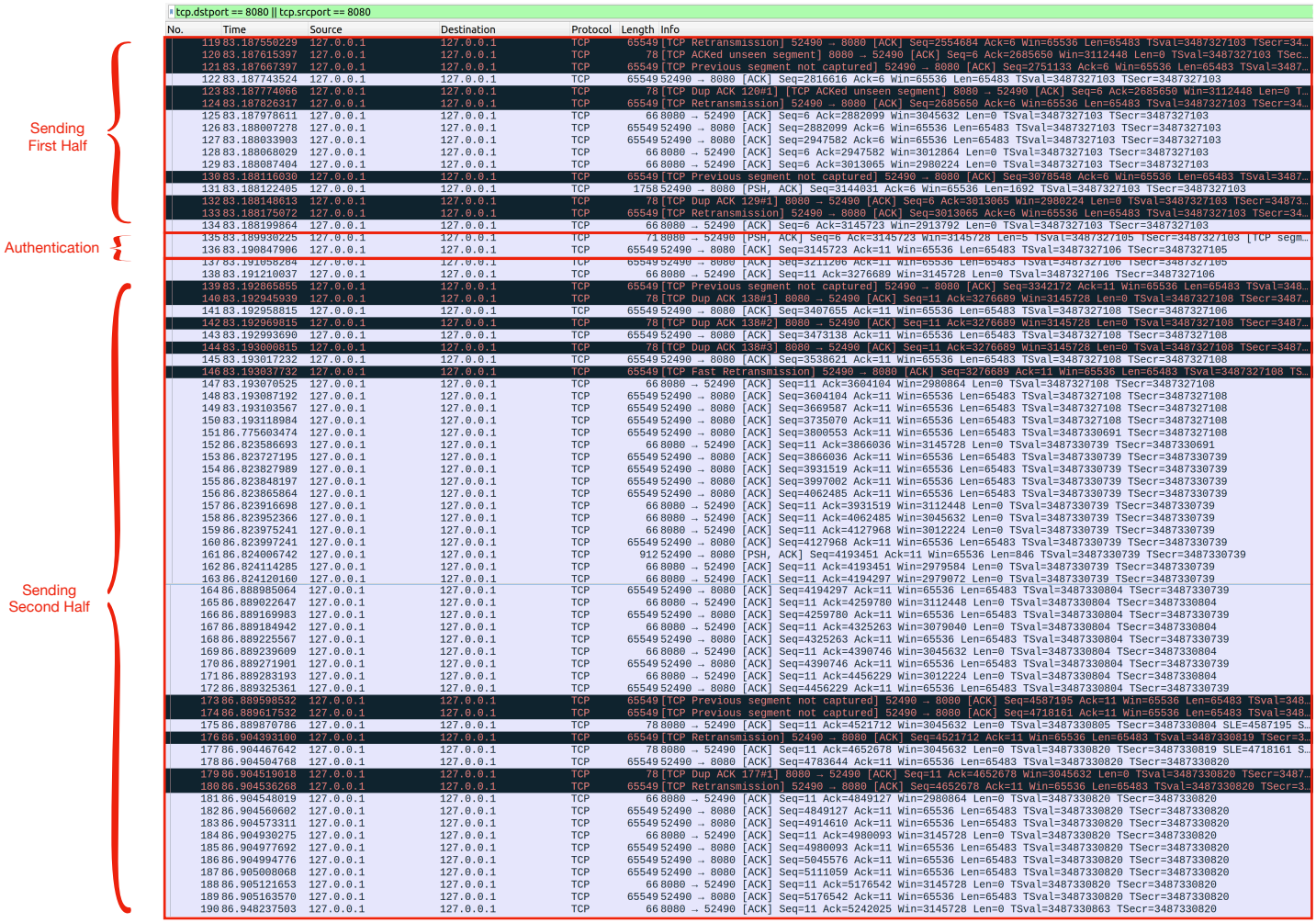


Since we already explained about the process of sending the message including opening/ closing the connection, sending authentication, TCP ACKed unseen segment message, TCP previous segment not captured message, TCP Dup ACK, TCP Retransmission, TCP Fast Retransmission and TCP Window Update will not explain it again.

20% lost:

Here, we increase the packet loss to 20%.
Thus, we should see a lot more packet get lost:

Second time sending file:



tcp.dstport == 8080 || tcp.srcport == 8080

No.	Time	Source	Destination	Protocol	Length	Info
119	83.187550229	127.0.0.1	127.0.0.1	TCP	65549	[TCP Retransmission] 52490 → 8080 [ACK] Seq=2554684 Ack=6 Win=65536 Len=65483 TSval=3487327103 TSecr=34…
120	83.187615397	127.0.0.1	127.0.0.1	TCP	78	[TCP ACKed unseen segment] 8080 → 52490 [ACK] Seq=6 Ack=2685650 Win=3112448 Len=0 TSval=3487327103 TSec…
121	83.187667397	127.0.0.1	127.0.0.1	TCP	65549	[TCP Previous segment not captured] 52490 → 8080 [ACK] Seq=2751133 Ack=6 Win=65536 Len=65483 TSval=3487…
122	83.187743524	127.0.0.1	127.0.0.1	TCP	65549	52490 → 8080 [ACK] Seq=2816616 Ack=6 Win=65536 Len=65483 TSval=3487327103 TSecr=3487327103
123	83.187774066	127.0.0.1	127.0.0.1	TCP	78	[TCP Dup ACK 120#1] [TCP ACKed unseen segment] 8080 → 52490 [ACK] Seq=6 Ack=2685650 Win=3112448 Len=0 T…
124	83.187826317	127.0.0.1	127.0.0.1	TCP	65549	[TCP Retransmission] 52490 → 8080 [ACK] Seq=2685650 Ack=6 Win=65536 Len=65483 TSval=3487327103 TSecr=34…
125	83.187978611	127.0.0.1	127.0.0.1	TCP	66	8080 → 52490 [ACK] Seq=6 Ack=2882099 Win=3045632 Len=0 TSval=3487327103 TSecr=3487327103
126	83.188007278	127.0.0.1	127.0.0.1	TCP	65549	52490 → 8080 [ACK] Seq=2882099 Ack=6 Win=65536 Len=65483 TSval=3487327103 TSecr=3487327103
127	83.188033903	127.0.0.1	127.0.0.1	TCP	65549	52490 → 8080 [ACK] Seq=2947582 Ack=6 Win=65536 Len=65483 TSval=3487327103 TSecr=3487327103
128	83.188068029	127.0.0.1	127.0.0.1	TCP	66	8080 → 52490 [ACK] Seq=6 Ack=2947582 Win=3012864 Len=0 TSval=3487327103 TSecr=3487327103
129	83.188087404	127.0.0.1	127.0.0.1	TCP	66	8080 → 52490 [ACK] Seq=6 Ack=3013065 Win=2980224 Len=0 TSval=3487327103 TSecr=3487327103
130	83.188116030	127.0.0.1	127.0.0.1	TCP	65549	[TCP Previous segment not captured] 52490 → 8080 [ACK] Seq=3078548 Ack=6 Win=65536 Len=65483 TSval=3487…
131	83.188122405	127.0.0.1	127.0.0.1	TCP	1758	52490 → 8080 [PSH, ACK] Seq=3144031 Ack=6 Win=65536 Len=1692 TSval=3487327103 TSecr=3487327103
132	83.188148613	127.0.0.1	127.0.0.1	TCP	78	[TCP Dup ACK 129#1] 8080 → 52490 [ACK] Seq=6 Ack=3013065 Win=2980224 Len=0 TSval=3487327103 TSecr=34873…
133	83.188175072	127.0.0.1	127.0.0.1	TCP	65549	[TCP Retransmission] 52490 → 8080 [ACK] Seq=3013065 Ack=6 Win=65536 Len=65483 TSval=3487327103 TSecr=34…
134	83.188199864	127.0.0.1	127.0.0.1	TCP	66	8080 → 52490 [ACK] Seq=6 Ack=3145723 Win=2913792 Len=0 TSval=3487327103 TSecr=3487327103
135	83.189930225	127.0.0.1	127.0.0.1	TCP	71	8080 → 52490 [PSH, ACK] Seq=6 Ack=3145723 Win=3145728 Len=5 TSval=3487327105 TSecr=3487327103 [TCP segm…
136	83.190847906	127.0.0.1	127.0.0.1	TCP	65549	52490 → 8080 [ACK] Seq=3145723 Ack=11 Win=65536 Len=65483 TSval=3487327106 TSecr=3487327105
137	83.191058284	127.0.0.1	127.0.0.1	TCP	65549	52490 → 8080 [ACK] Seq=3211206 Ack=11 Win=65536 Len=65483 TSval=3487327106 TSecr=3487327105
138	83.191210037	127.0.0.1	127.0.0.1	TCP	66	8080 → 52490 [ACK] Seq=11 Ack=3276689 Win=3145728 Len=0 TSval=3487327106 TSecr=3487327106
139	83.192865855	127.0.0.1	127.0.0.1	TCP	65549	[TCP Previous segment not captured] 52490 → 8080 [ACK] Seq=3342172 Ack=11 Win=65536 Len=65483 TSval=348…
140	83.192945939	127.0.0.1	127.0.0.1	TCP	78	[TCP Dup ACK 138#1] 8080 → 52490 [ACK] Seq=11 Ack=3276689 Win=3145728 Len=0 TSval=3487327108 TSecr=3487…
141	83.192958815	127.0.0.1	127.0.0.1	TCP	65549	52490 → 8080 [ACK] Seq=3407655 Ack=11 Win=65536 Len=65483 TSval=3487327108 TSecr=3487327106
142	83.192959815	127.0.0.1	127.0.0.1	TCP	78	[TCP Dup ACK 138#2] 8080 → 52490 [ACK] Seq=11 Ack=3276689 Win=3145728 Len=0 TSval=3487327108 TSecr=3487…
143	83.192993690	127.0.0.1	127.0.0.1	TCP	65549	52490 → 8080 [ACK] Seq=3473138 Ack=11 Win=65536 Len=65483 TSval=3487327108 TSecr=3487327108
144	83.193000815	127.0.0.1	127.0.0.1	TCP	78	[TCP Dup ACK 138#3] 8080 → 52490 [ACK] Seq=11 Ack=3276689 Win=3145728 Len=0 TSval=3487327108 TSecr=3487…
145	83.193017232	127.0.0.1	127.0.0.1	TCP	65549	52490 → 8080 [ACK] Seq=3538621 Ack=11 Win=65536 Len=65483 TSval=3487327108 TSecr=3487327108
146	83.193037732	127.0.0.1	127.0.0.1	TCP	65549	[TCP Fast Retransmission] 52490 → 8080 [ACK] Seq=3276689 Ack=11 Win=65536 Len=65483 TSval=3487327108 TS…
147	83.193070525	127.0.0.1	127.0.0.1	TCP	66	8080 → 52490 [ACK] Seq=11 Ack=3604104 Win=2980864 Len=0 TSval=3487327108 TSecr=3487327108
148	83.193087192	127.0.0.1	127.0.0.1	TCP	65549	52490 → 8080 [ACK] Seq=3604104 Ack=11 Win=65536 Len=65483 TSval=3487327108 TSecr=3487327108
149	83.193103567	127.0.0.1	127.0.0.1	TCP	65549	52490 → 8080 [ACK] Seq=3669587 Ack=11 Win=65536 Len=65483 TSval=3487327108 TSecr=3487327108
150	83.193118984	127.0.0.1	127.0.0.1	TCP	65549	52490 → 8080 [ACK] Seq=3735070 Ack=11 Win=65536 Len=65483 TSval=3487327108 TSecr=3487327108
151	86.775603474	127.0.0.1	127.0.0.1	TCP	65549	52490 → 8080 [ACK] Seq=3800553 Ack=11 Win=65536 Len=65483 TSval=3487330691 TSecr=3487327108
152	86.823586693	127.0.0.1	127.0.0.1	TCP	66	8080 → 52490 [ACK] Seq=11 Ack=3866036 Win=3145728 Len=0 TSval=3487330739 TSecr=3487330691
153	86.823727195	127.0.0.1	127.0.0.1	TCP	65549	52490 → 8080 [ACK] Seq=3866036 Ack=11 Win=65536 Len=65483 TSval=3487330739 TSecr=3487330739
154	86.823827989	127.0.0.1	127.0.0.1	TCP	65549	52490 → 8080 [ACK] Seq=3931519 Ack=11 Win=65536 Len=65483 TSval=3487330739 TSecr=3487330739
155	86.823848197	127.0.0.1	127.0.0.1	TCP	65549	52490 → 8080 [ACK] Seq=3997002 Ack=11 Win=65536 Len=65483 TSval=3487330739 TSecr=3487330739
156	86.823865864	127.0.0.1	127.0.0.1	TCP	65549	52490 → 8080 [ACK] Seq=4062485 Ack=11 Win=65536 Len=65483 TSval=3487330739 TSecr=3487330739
157	86.823916698	127.0.0.1	127.0.0.1	TCP	66	8080 → 52490 [ACK] Seq=11 Ack=3931519 Win=3112448 Len=0 TSval=3487330739 TSecr=3487330739
158	86.823952366	127.0.0.1	127.0.0.1	TCP	66	8080 → 52490 [ACK] Seq=11 Ack=4062485 Win=3045632 Len=0 TSval=3487330739 TSecr=3487330739
159	86.823975241	127.0.0.1	127.0.0.1	TCP	66	8080 → 52490 [ACK] Seq=11 Ack=4127968 Win=3012224 Len=0 TSval=3487330739 TSecr=3487330739
160	86.823997241	127.0.0.1	127.0.0.1	TCP	65549	52490 → 8080 [ACK] Seq=4127968 Ack=11 Win=65536 Len=65483 TSval=3487330739 TSecr=3487330739
161	86.824006742	127.0.0.1	127.0.0.1	TCP	912	52490 → 8080 [PSH, ACK] Seq=4193451 Ack=11 Win=65536 Len=846 TSval=3487330739 TSecr=3487330739
162	86.824114285	127.0.0.1	127.0.0.1	TCP	66	8080 → 52490 [ACK] Seq=11 Ack=4193451 Win=2979584 Len=0 TSval=3487330739 TSecr=3487330739
163	86.824120160	127.0.0.1	127.0.0.1	TCP	66	8080 → 52490 [ACK] Seq=11 Ack=4194297 Win=2979072 Len=0 TSval=3487330739 TSecr=3487330739
164	86.888985064	127.0.0.1	127.0.0.1	TCP	65549	52490 → 8080 [ACK] Seq=4194297 Ack=11 Win=65536 Len=65483 TSval=3487330804 TSecr=3487330739
165	86.889022647	127.0.0.1	127.0.0.1	TCP	66	8080 → 52490 [ACK] Seq=11 Ack=4259780 Win=3112448 Len=0 TSval=3487330804 TSecr=3487330804
166	86.889169983	127.0.0.1	127.0.0.1	TCP	65549	52490 → 8080 [ACK] Seq=4259780 Ack=11 Win=65536 Len=65483 TSval=3487330804 TSecr=3487330739
167	86.889184942	127.0.0.1	127.0.0.1	TCP	66	8080 → 52490 [ACK] Seq=11 Ack=4325263 Win=3079040 Len=0 TSval=3487330804 TSecr=3487330804
168	86.889225567	127.0.0.1	127.0.0.1	TCP	65549	52490 → 8080 [ACK] Seq=4325263 Ack=11 Win=65536 Len=65483 TSval=3487330804 TSecr=3487330739
169	86.889239609	127.0.0.1	127.0.0.1	TCP	66	8080 → 52490 [ACK] Seq=11 Ack=4390746 Win=3045632 Len=0 TSval=3487330804 TSecr=3487330804
170	86.889271901	127.0.0.1	127.0.0.1	TCP	65549	52490 → 8080 [ACK] Seq=4390746 Ack=11 Win=65536 Len=65483 TSval=3487330804 TSecr=3487330739
171	86.889283193	127.0.0.1	127.0.0.1	TCP	66	8080 → 52490 [ACK] Seq=11 Ack=4456229 Win=3012224 Len=0 TSval=3487330804 TSecr=3487330804
172	86.889325361	127.0.0.1	127.0.0.1	TCP	65549	52490 → 8080 [ACK] Seq=4456229 Ack=11 Win=65536 Len=65483 TSval=3487330804 TSecr=3487330739
173	86.889598532	127.0.0.1	127.0.0.1	TCP	65549	[TCP Previous segment not captured] 52490 → 8080 [ACK] Seq=4587195 Ack=11 Win=65536 Len=65483 TSval=348…
174	86.889617532	127.0.0.1	127.0.0.1	TCP	65549	[TCP Previous segment not captured] 52490 → 8080 [ACK] Seq=4718161 Ack=11 Win=65536 Len=65483 TSval=348…
175	86.889870786	127.0.0.1	127.0.0.1	TCP	78	8080 → 52490 [ACK] Seq=11 Ack=4521712 Win=3045632 Len=0 TSval=3487330805 TSecr=3487330804 SLE=4587195 S…
176	86.904393100	127.0.0.1	127.0.0.1	TCP	65549	[TCP Retransmission] 52490 → 8080 [ACK] Seq=4521712 Ack=11 Win=65536 Len=65483 TSval=3487330819 TSecr=3…
177	86.904467642	127.0.0.1	127.0.0.1	TCP	78	8080 → 52490 [ACK] Seq=11 Ack=4652678 Win=3045632 Len=0 TSval=3487330820 TSecr=3487330819 SLE=4718161 S…
178	86.904504768	127.0.0.1	127.0.0.1	TCP	65549	52490 → 8080 [ACK] Seq=4783644 Ack=11 Win=65536 Len=65483 TSval=3487330820 TSecr=3487330820
179	86.904519018	127.0.0.1	127.0.0.1	TCP	78	[TCP Dup ACK 177#1] 8080 → 52490 [ACK] Seq=11 Ack=4652678 Win=3045632 Len=0 TSval=3487330820 TSecr=3487…
180	86.904536268	127.0.0.1	127.0.0.1	TCP	65549	[TCP Retransmission] 52490 → 8080 [ACK] Seq=4652678 Ack=11 Win=65536 Len=65483 TSval=3487330820 TSecr=3…
181	86.904548019	127.0.0.1	127.0.0.1	TCP	66	8080 → 52490 [ACK] Seq=11 Ack=4849127 Win=2980864 Len=0 TSval=3487330820 TSecr=3487330820
182	86.904560602	127.0.0.1	127.0.0.1	TCP	65549	52490 → 8080 [ACK] Seq=4849127 Ack=11 Win=65536 Len=65483 TSval=3487330820 TSecr=3487330820
183	86.904573311	127.0.0.1	127.0.0.1	TCP	65549	52490 → 8080 [ACK] Seq=4914610 Ack=11 Win=65536 Len=65483 TSval=3487330820 TSecr=3487330820
184	86.904930275	127.0.0.1	127.0.0.1	TCP	66	8080 → 52490 [ACK] Seq=11 Ack=4980093 Win=3145728 Len=0 TSval=3487330820 TSecr=3487330820
185	86.904977692	127.0.0.1	127.0.0.1	TCP	65549	52490 → 8080 [ACK] Seq=4980093 Ack=11 Win=65536 Len=65483 TSval=3487330820 TSecr=3487330820
186	86.904994776	127.0.0.1	127.0.0.1	TCP	65549	52490 → 8080 [ACK] Seq=5045576 Ack=11 Win=65536 Len=65483 TSval=3487330820 TSecr=3487330820
187	86.905008068	127.0.0.1	127.0.0.1	TCP	65549	52490 → 8080 [ACK] Seq=5111059 Ack=11 Win=65536 Len=65483 TSval=3487330820 TSecr=3487330820
188	86.905121653	127.0.0.1	127.0.0.1	TCP	66	8080 → 52490 [ACK] Seq=11 Ack=5176542 Win=3145728 Len=0 TSval=3487330820 TSecr=3487330820
189	86.905163570	127.0.0.1	127.0.0.1	TCP	65549	52490 → 8080 [ACK] Seq=5176542 Ack=11 Win=65536 Len=65483 TSval=3487330820 TSecr=3487330820
190	86.948237503	127.0.0.1	127.0.0.1	TCP	66	8080 → 52490 [ACK] Seq=11 Ack=5242025 Win=3145728 Len=0 TSval=3487330863 TSecr=3487330820

Since we already explained about the process of sending the message including opening/ closing the connection, sending authentication, TCP ACKed unseen segment message, TCP previous segment not captured message, TCP Dup ACK, TCP Retransmission, TCP Fast Retransmission and TCP Window Update will not explain it again.

- Notice that a lot of packet may have not been captured by the wireshark (which is fine).

Example for how much there is in the 20% loss:

223	104.860116520	127.0.0.1	127.0.0.1	TCP	65549	[TCP Previous segment not captured] 52490 → 8080 [ACK] Seq=6553377 Ack=16 Win=65536 Len=65483 TSval=348…
224	104.860126270	127.0.0.1	127.0.0.1	TCP	78	[TCP Window Update] 8080 → 52490 [ACK] Seq=16 Ack=6487894 Win=3112448 Len=0 TSval=3487348775 TSecr=3487…
225	104.860155479	127.0.0.1	127.0.0.1	TCP	65549	52490 → 8080 [ACK] Seq=6618860 Ack=16 Win=65536 Len=65483 TSval=3487348775 TSecr=3487332415
226	104.860166729	127.0.0.1	127.0.0.1	TCP	78	[TCP Dup ACK 222#1] 8080 → 52490 [ACK] Seq=16 Ack=6487894 Win=3112448 Len=0 TSval=3487348775 TSecr=3487…
227	104.860402357	127.0.0.1	127.0.0.1	TCP	65549	52490 → 8080 [ACK] Seq=6684343 Ack=16 Win=65536 Len=65483 TSval=3487348776 TSecr=3487348775
228	104.860411649	127.0.0.1	127.0.0.1	TCP	78	[TCP Dup ACK 222#2] 8080 → 52490 [ACK] Seq=16 Ack=6487894 Win=3112448 Len=0 TSval=3487348776 TSecr=3487…
229	104.860437150	127.0.0.1	127.0.0.1	TCP	65549	[TCP Previous segment not captured] 52490 → 8080 [ACK] Seq=6815309 Ack=16 Win=65536 Len=65483 TSval=348…
230	104.860447150	127.0.0.1	127.0.0.1	TCP	86	[TCP Dup ACK 222#3] 8080 → 52490 [ACK] Seq=16 Ack=6487894 Win=3112448 Len=0 TSval=3487348776 TSecr=3487…
231	104.860464442	127.0.0.1	127.0.0.1	TCP	65549	[TCP Fast Retransmission] 52490 → 8080 [ACK] Seq=6487894 Ack=16 Win=65536 Len=65483 TSval=3487348776 TS…
232	104.860502109	127.0.0.1	127.0.0.1	TCP	65549	52490 → 8080 [ACK] Seq=6880792 Ack=16 Win=65536 Len=65483 TSval=3487348776 TSecr=3487348776
233	104.860511734	127.0.0.1	127.0.0.1	TCP	78	8080 → 52490 [ACK] Seq=16 Ack=6749826 Win=2978944 Len=0 TSval=3487348776 TSecr=3487348776 SLE=6815309 S…
234	104.860526984	127.0.0.1	127.0.0.1	TCP	65549	[TCP Previous segment not captured] 52490 → 8080 [ACK] Seq=7011758 Ack=16 Win=65536 Len=65483 TSval=348…
235	104.860542401	127.0.0.1	127.0.0.1	TCP	65549	52490 → 8080 [ACK] Seq=7077241 Ack=16 Win=65536 Len=65483 TSval=3487348776 TSecr=3487348776
236	104.860590527	127.0.0.1	127.0.0.1	TCP	86	[TCP Dup ACK 233#1] 8080 → 52490 [ACK] Seq=16 Ack=6749826 Win=2978944 Len=0 TSval=3487348776 TSecr=3487…
237	104.860609319	127.0.0.1	127.0.0.1	TCP	65549	[TCP Retransmission] 52490 → 8080 [ACK] Seq=6749826 Ack=16 Win=65536 Len=65483 TSval=3487348776 TSecr=3…
238	113.909512076	127.0.0.1	127.0.0.1	TCP	65549	[TCP Retransmission] 52490 → 8080 [ACK] Seq=6749826 Ack=16 Win=65536 Len=65483 TSval=3487357825 TSecr=3…
239	125.684637152	127.0.0.1	127.0.0.1	TCP	65549	[TCP Retransmission] 52490 → 8080 [ACK] Seq=6749826 Ack=16 Win=65536 Len=65483 TSval=3487369600 TSecr=3…
240	150.776066495	127.0.0.1	127.0.0.1	TCP	65549	[TCP Retransmission] 52490 → 8080 [ACK] Seq=6749826 Ack=16 Win=65536 Len=65483 TSval=3487394691 TSecr=3…
241	150.776112204	127.0.0.1	127.0.0.1	TCP	86	8080 → 52490 [ACK] Seq=16 Ack=6946275 Win=3079040 Len=0 TSval=3487394691 TSecr=3487348776 SLE=6749826 S…
242	150.776165539	127.0.0.1	127.0.0.1	TCP	65549	[TCP Previous segment not captured] 52490 → 8080 [ACK] Seq=7208207 Ack=16 Win=65536 Len=65483 TSval=348…
243	150.776219664	127.0.0.1	127.0.0.1	TCP	65549	52490 → 8080 [ACK] Seq=7273690 Ack=16 Win=65536 Len=65483 TSval=3487394691 TSecr=3487394691
244	150.776224956	127.0.0.1	127.0.0.1	TCP	912	52490 → 8080 [PSH, ACK] Seq=7339173 Ack=16 Win=65536 Len=846 TSval=3487394691 TSecr=3487394691
245	150.776243332	127.0.0.1	127.0.0.1	TCP	86	[TCP Dup ACK 241#1] 8080 → 52490 [ACK] Seq=16 Ack=6946275 Win=3079040 Len=0 TSval=3487394691 TSecr=3487…
246	150.776255790	127.0.0.1	127.0.0.1	TCP	86	[TCP Dup ACK 241#2] 8080 → 52490 [ACK] Seq=16 Ack=6946275 Win=3079040 Len=0 TSval=3487394691 TSecr=3487…
247	150.776258623	127.0.0.1	127.0.0.1	TCP	86	[TCP Dup ACK 241#3] 8080 → 52490 [ACK] Seq=16 Ack=6946275 Win=3079040 Len=0 TSval=3487394691 TSecr=3487…
248	150.776277332	127.0.0.1	127.0.0.1	TCP	65549	[TCP Retransmission] 52490 → 8080 [ACK] Seq=7142724 Ack=16 Win=65536 Len=65483 TSval=3487394691 TSecr=3…
249	150.776294374	127.0.0.1	127.0.0.1	TCP	78	[TCP Dup ACK 241#4] 8080 → 52490 [ACK] Seq=16 Ack=6946275 Win=3079040 Len=0 TSval=3487394691 TSecr=3487…
250	204.021629837	127.0.0.1	127.0.0.1	TCP	65549	[TCP Retransmission] 52490 → 8080 [ACK] Seq=6946275 Ack=16 Win=65536 Len=65483 TSval=3487447937 TSecr=3…

CC Cubic vs. CC Reno

Congestion control algorithms are made to handle the large amount of traffic TCP carries.
The CC algorithms prevents the Internet from slowing and entering “congestion collapse”,
which occurs, for example, if the network is clogged with unnecessary retransmissions of lost packets.
Since TCP CC is very essential, it is continuously being improved and modified.
That’s because the Internet changes, and so TCP congestion control must evolve with it

TCP Cubic and TCP Reno are both CC (congestion control) algorithms.
In this exercise we focused on both of them, and so we will compare between them and do different tests to determine which one is better, and under what conditions:

TCP Reno - A congestion control algorithm which uses the traditional linear increase function for the window size ($W = W + 1$). The stages are:
1 - The transmission rate will gradually increase until some packet loss occurs.
2 - The increase will be made by linearly increasing the congestion window, i.e. by adding a value.
3 - If congestion is inferred, meaning packet loss occurs (which is recognized by 3 duplicate ACKs), the CWND (congestion window) will be decreased by half, and then grow linearly.
Pay attention that decreasing the CWND is like decreasing the transmutation rate.

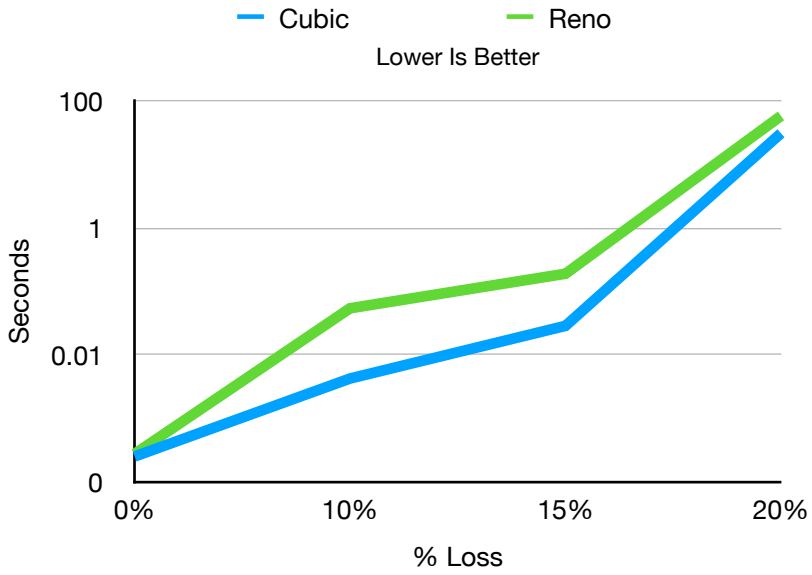
TCP Cubic - A congestion control algorithm which implements a binary search increase function for the windows size. The stages are:
1 - When experiencing congestion event, the window size for that instant will be recorded as W_{max} or the maximum window size, where W_{max} is the sending rate at which congestion loss was detected.
2 - The transmission will then be restarted with a half the window value and, if no congestion occurs, this value will increase according to the cubic function (fast).
3 - As the window approaches W_{max} the increments will slow down.
4 - Once we reached W_{max}, the value of the window will continue to increase discreetly.
5 - Finally, if the network is still not experiencing any congestion, the window size will continue to increase according to the cubic function.

Thus, on paper, Cubic is better then reno since it can reach the available bandwidth much faster.
While TCP Cubic and Reno are made for high throughput, they cause a lot of queuing delays since they reduce their CWND only when they experience a packet loss, like when the queue is full.

As we investigated and researched more and more,
we came to the believe that Cubic is definitely better then Reno - no questions asked.
However, when doing real-time testing, the timing of both of them was very close and sometimes we got that reno is better. So what was better and by how much in the end result?

We will try to answer the question of which one was actually better by using the result we collected.

CC \ % LOSS	0%	10%	15%	20%
Cubic	0.000255	0.004305	0.029044	31.242127
Reno	0.00028	0.054949	0.192251	58.583048

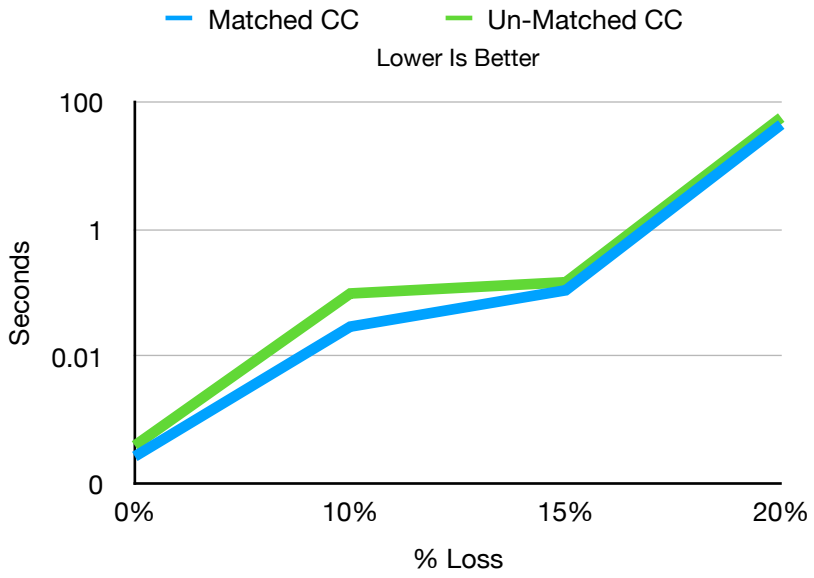


Thus , we see that Cubic is actually better as we expected.

Also, there is another important question:
Is it better to match between the sender's CC and receiver's CC, or not to match? Does it even matter?
We tested this also, and we will show you the results in a moment.
The matched CC test is the test above since in our experiment we matched between the sender and receiver (first used "Cubic" on Both ends, and then used "Reno" on both ends).
In the un-matched CC test, the first half is sent where "Cubic" is set on sender side, and "Reno" on receiver side, and second half is sent where "Reno" is set on sender side, and "Cubic" on receiver side.

- For each % Loss, we will take the average time it took for the whole file.
- Though we calculated wrong the time for the whole size (divided by 2 the average itself), it would not affect this testing since the proportions are the same.

Matched \ % LOSS	0%	10%	15%	20%
Matched CC	0.000268	0.029627	0.110648	44.912588
Un-Matched CC	0.000397	0.098525	0.148197	57.921178



Thus , we see that Matched CC is better then Un-Matched CC.
This is why we chose Matched CC in this exercise.
However, when we the difference is minimal non the less when we look at higher % Loss.

Conclusions

After completing this research, we have learned a lot of things.
First, we have increased our skill in the C programming language,
and specifically in the socket programming field.

Not only that, but we actually learned a few methods and structs that can help us moving forward as programmers like measuring time methods, reseting an array to a specific value or 0 and dynamically allocate memory.

This research also help us understand more about the TCP communication and about the different congestion control (CC) algorithms in TCP .
We took it one step forward and compared and test between “Reno” and “Cubic” CC algorithms, and check what is better: matching between receiver and sender or to have them on different CC algorithms.

Lastly, we learned more about how to use the WireShark program,
how to recognize packet, and the different TCP messages and their meaning.

To sum things up, this was a hard project which took time and resources,
But non the less - we are very grateful and happy with the end result.

Bibliography

- This is an hyper-linked supported bibliography.

- *CR Ferreira . 8 Ways to Measure Execution Time in C/C++: Level-Gitconnected, 2020.*
- *Fraida Fund . TCP congestion control: Witestlab, 2017.*
- *Pandora FMS team . Why does CUBIC take us back to TCP congestion control?: Pandora FMS, 2021.*
- *Rolou Lyn && Ronald && Alrence . Design and Implementation of Client-Server Based Application Using Socket Programming in a Distributed Computing Environment: ResearchGate, 2017.*
- *psionic12 . Can a client and server use different congestion algorithms when communicating?: StackOverflow, 2021.*
- *Unknown . What is the best way to find out what is causing TCP acked unseen segment.: WireShark.org, 2018.*
- *Unknown . TCP ACKed unseen segment.: WireShark.org, 2018.*
- *Unknown . TCP DUP ACK/Out-of-Order/Previous segment not captured/Retransmission flooded on wireshark logs: serverfault, 2022.*
- *Mike .What is duplicate ACK when does it occur?: StackOverflow, 2018.*
- *Unknown .TCP_Analyze_Sequence_Numbers: WireShark, 2020.*
- *Xor Calculator: xor.pw.*