Task 4

Sources:

https://www.noip.com/blog/2019/10/30/doh-pros-cons-dns-https

https://www.efficientip.com/why-using-doh-is-questionable

https://www.samknows.com/blog/dns-over-https-performance

https://idiotdeveloper.com/file-transfer-using-tcp-socket-in-c

https://stackoverflow.com/questions/45747973/in-c-how-to-indicate-eof-after-writing-data-without-close-shutdown-a-socket

Part A:

1.So the biggest advantage of the DoH is of course the security and privacy that he can give by encrypting the queries that we are asking the DNS, but another advantage is that because DoH centralize DNS traffic to a few DoH servers, the load time performance is typically improve, well it is clear because there are more chances to get the Ip addresses while checking with a few DNS.

2.Two disadvantages are:

* It overrides any sort of DNS filtering your network is doing to provide insight into security and your network info.

*It weakens cyber-security. By encrypting DNS queries, companies using DNS monitoring for cybersecurity measures will lose visibility into data such as query type, response and originating IP that are used to determine bad actors.

3.We will observe the second disadvantage that I just present. A solution to moderate the problem is to limit the ways of encrypting or to add a sign to the way that this query was encrypted inside the query and to give a cyber-security that were authorized the way to decrypting the encrypting.

4.a. **Advantage** -faster because it is immediately searching and mor secure as we discussed above.

Disadvantage- we have no filtering at all as the we have in the DNS.

b. **Advantage** - with the proxy we can filter the queries as the DNS is doing if anyone wishes to block or control DNS traffic contained inside a DoH transaction.

Disadvantage – we are more vulnerable as if we send our queries to an external proxy we can be tracked, and someone can follow all by searches it is also taking more time to send int to the proxy server.

C. **Advantage** - with the proxy we can filter the queries as the DNS is doing if anyone wishes to block or control DNS traffic contained inside a DoH transaction.

Disadvantage – it is taking more time it must go throw proxy server.

D. **Advantage** – the default will be DoH you will be more secure you can add filtering and you don't need to deal with a proxy which will be slower.

Disadvantage – changing the settings, require from the customer to install.

In my opinion the best implementation is the last one. Today most of the people are looking for the fastest way to get things, the proxy ways will slower the process and the first one has no filtering which is important, computers are always required to install different programs, different updates that are changing the setting are always happening so it can be part of it. Because of that I don't think that the requirements from the client are exaggerated, and the disadvantages are not so big, but you are getting the advantages.

5. DNS-over-HTTPS operates over TCP, which can retransmit data very quickly in the case of packet losses, whereas traditional DNS clients use UDP and wait for a fixed time before retrying. So, in lossy networks, DoH *may* outperform UDP-based DNS. However, DoH-over-TCP has head of line blocking (if there is significant packet loss then all requests over that TCP connection slow down).

Part B:

First, this is my code for the Task:

For sender.c:

```c
#include <stdio.h>

#if defined _WIN32

// link with Ws2_32.lib
#pragma comment(lib,"Ws2_32.lib")
#include <winsock2.h>
#include <ws2tcpip.h>

#else
#include <netinet/tcp.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <unistd.h>
#endif
#define SIZE 1024

void send_file(FILE *fp, int sockfd){
  int n;
  char data[SIZE] = {0};

  while(fgets(data, SIZE, fp) != NULL) {
    if (send(sockfd, data, sizeof(data), 0) == -1) {
      perror("[-]Error in sending file.");
      exit(1);
    }
    bzero(data, SIZE);
  }
}

int main(){
  char *ip = "127.0.0.1";
  int port = 5060;
  int e;

  int sockfd;
  struct sockaddr_in server_addr;
  FILE *fp;
  char *filename = "forTask.txt";
  for(int i = 0; i < 6; i++){


    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if(sockfd < 0) {
        perror("[-]Error in socket");
        exit(1);
    }
    printf("[+]Server socket created successfully.\n");

    server_addr.sin_family = AF_INET;
    server_addr.sin_port = port;
    server_addr.sin_addr.s_addr = inet_addr(ip);

    e = connect(sockfd, (struct sockaddr*)&server_addr,
sizeof(server_addr));
    if(e == -1) {
        perror("[-]Error in socket");
        exit(1);
    }
    printf("[+]Connected to Server.\n");

    char buf[256];
    fp = fopen(filename, "r");
    if (fp == NULL) {
        perror("[-]Error in reading file.");
        exit(1);
    }
            send_file(fp, sockfd);
            printf("[+]File data sent successfully.\n");
            printf("The %d Time in cubic\n", i+1);


        strcpy(buf, "reno");
        socklen_t len = strlen(buf);

        if (setsockopt(sockfd, IPPROTO_TCP, TCP_CONGESTION, buf, len)
!= 0) {
            perror("setsockopt");
            return -1;
        }

            send_file(fp, sockfd);
            fclose(fp);
            printf("[+]File data sent successfully.\n");
            printf("The %d Time in reno\n", i+1);
    printf("[+]Closing the connection.\n");
    close(sockfd);
}
  return 0;
}
```

For Measure.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <arpa/inet.h>
#include <time.h>
#define SIZE 1024



int main(){
  char *ip = "127.0.0.1";
  int port = 5060;
  int e;

  int sockfd, new_sock;
  struct sockaddr_in server_addr, new_addr;
  socklen_t addr_size;
  char buffer[SIZE];

  sockfd = socket(AF_INET, SOCK_STREAM, 0);
  if(sockfd < 0) {
    perror("[-]Error in socket");
    exit(1);
  }
  printf("[+]Server socket created successfully.\n");

  server_addr.sin_family = AF_INET;
  server_addr.sin_port = port;
  server_addr.sin_addr.s_addr = inet_addr(ip);

  e = bind(sockfd, (struct sockaddr*)&server_addr,
sizeof(server_addr));
  if(e < 0) {
    perror("[-]Error in bind");
    exit(1);
  }
  printf("[+]Binding successfull.\n");
 time_t start,end;
 int counter = 1;
 double ans = 0.0;
 while(counter <= 5){
     start=clock();
    if(listen(sockfd, 10) == 0){
    printf("[+]Listening....\n");
    }else{
    perror("[-]Error in listening");
        exit(1);
    }

    addr_size = sizeof(new_addr);
    new_sock = accept(sockfd, (struct sockaddr*)&new_addr, &addr_size);



    end=clock();
    double t=(end-start);
    ans += t;
    printf("The %d massege took %lf to be accepted \n",counter,t);
    counter++;
 }
    ans = ans/5;
    printf("The average sending 5 times with 20%% packet loste
is:  %lf\n",ans);
  return 0;
}
```

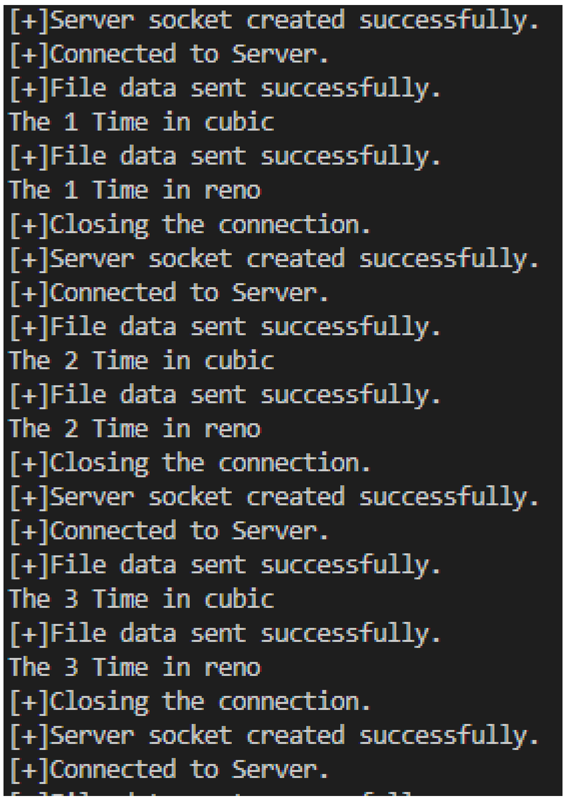
Now I built a table which present the average run time for every packet lost (5 repeats) as required.

Note: the run time is present by clock ticking (using the function clock()) as we can see in the code(included <time.h> library) when trying to show that in seconds the number is too small.

In addition, after the table you can find explanations and images that showing the time for each repeat including graphs that simulate the movement of the packets and showing the lost to each precent that was required in this task and more explanations.

Packet lost(precents)	Average time to send the (clocktick) file
10%	63.400000
15%	55.400000
20%	44.800000
25%	33.800000
30%	41.200000

Regarding to every run code with different percentage, I repeated every send 5 times with the CC changes as requested so actually I sent the file 10 time 5 as "cubic" and five a "reno" as required. We can see in in the code, but I printed it that we can see, this is what the sender printed:



```
[+]Server socket created successfully.
[+]Connected to Server.
[+]File data sent successfully.
The 1 Time in cubic
[+]File data sent successfully.
The 1 Time in reno
[+]Closing the connection.
[+]Server socket created successfully.
[+]Connected to Server.
[+]File data sent successfully.
The 2 Time in cubic
[+]File data sent successfully.
The 2 Time in reno
[+]Closing the connection.
[+]Server socket created successfully.
[+]Connected to Server.
[+]File data sent successfully.
The 3 Time in cubic
[+]File data sent successfully.
The 3 Time in reno
[+]Closing the connection.
[+]Server socket created successfully.
[+]Connected to Server.
```

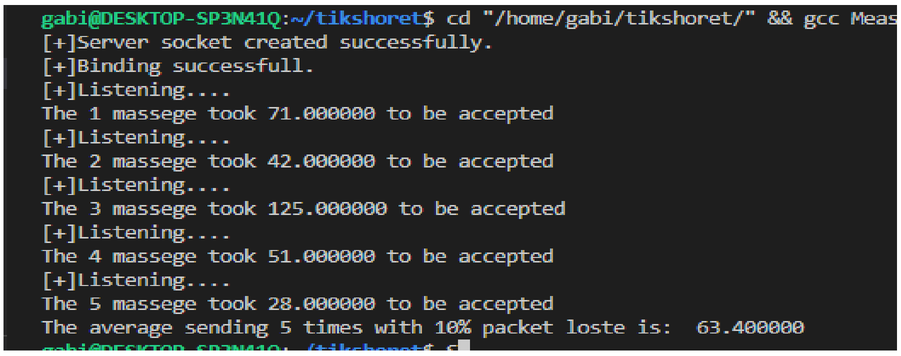
The picture is also sending it 4 and 5 times I am showing you here an example and as we can see the program sending it with "cubic" and with "reno".

We can also see all the printing that pointing that we created the server and connected etc.

10% packet lost:

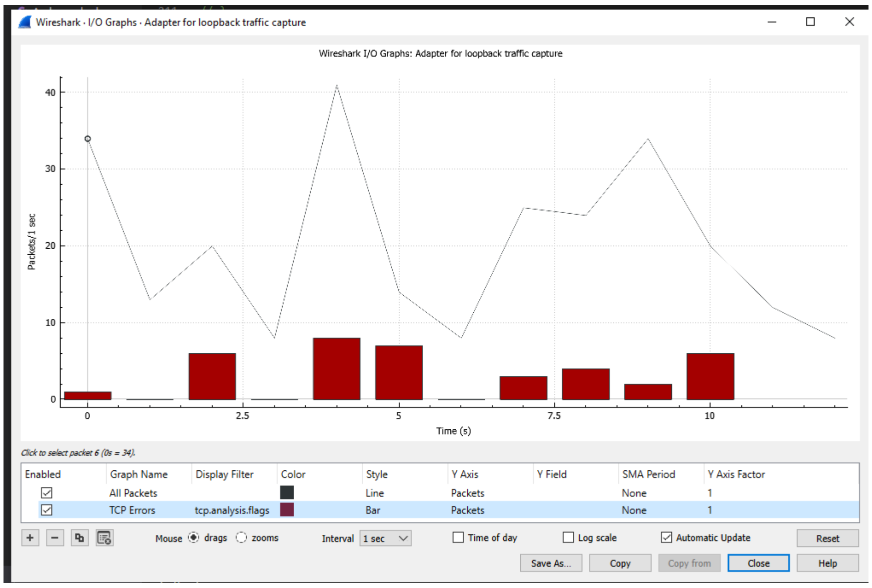
In the following image we can see the run time to each repeat (as we can see in the code above inside sender.c line 255. I used the setsockopt() to set the CC from "cubic" to "reno" also as explained above.

The last line showing the average time for all this massages with the correct precent.



```
gabi@DESKTOP-SP3N41Q:~/tikshoret$ cd "/home/gabi/tikshoret/" && gcc Meas
[+]Server socket created successfully.
[+]Binding successfull.
[+]Listening....
The 1 massege took 71.000000 to be accepted
[+]Listening....
The 2 massege took 42.000000 to be accepted
[+]Listening....
The 3 massege took 125.000000 to be accepted
[+]Listening....
The 4 massege took 51.000000 to be accepted
[+]Listening....
The 5 massege took 28.000000 to be accepted
The average sending 5 times with 10% packet loste is:  63.400000
```

In the following picture we can see the graph showing us the situation with the 10% packet lost. As we can see the red bins are showing us the packet error which is a direct result from the lost. We can predict that when the percentage lost will be higher the amount of the bins will grow (graphicly) we can see in in the next pictures.

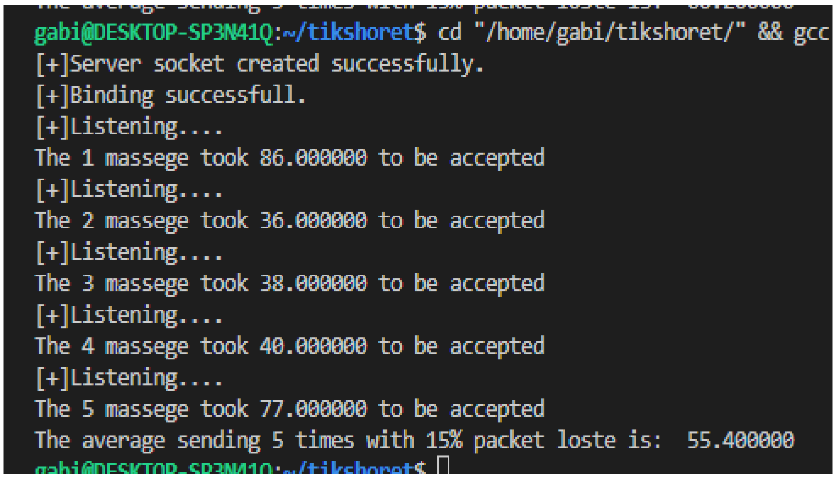


Enabled	Graph Name	Display Filter	Color	Style	Y Axis	Y Field	SMA Period	Y Axis Factor
☑	All Packets		■	Line	Packets		None	1
☑	TCP Errors	tcp.analysis.flags	■	Bar	Packets		None	1

15% packet lost:

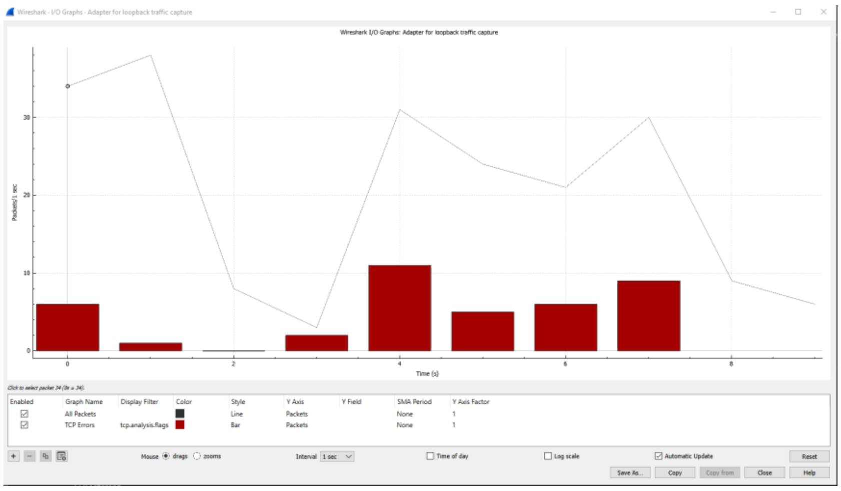
In the following image we can see the run time to each repeat (as we can see in the code above inside sender.c line 255. I used the setsockopt() to set the CC from "cubic" to "reno" also as explained above.

The last line showing the average time for all this massages with the correct precent.



```
gabi@DESKTOP-SP3N41Q:~/tikshoret$ cd "/home/gabi/tikshoret/" && gcc
[+]Server socket created successfully.
[+]Binding successfull.
[+]Listening....
The 1 massege took 86.000000 to be accepted
[+]Listening....
The 2 massege took 36.000000 to be accepted
[+]Listening....
The 3 massege took 38.000000 to be accepted
[+]Listening....
The 4 massege took 40.000000 to be accepted
[+]Listening....
The 5 massege took 77.000000 to be accepted
The average sending 5 times with 15% packet loste is:  55.400000
gabi@DESKTOP-SP3N41Q:~/tikshoret$
```

The next graph showing us the packet movement of the 15 percent packet lost, generally the graph showing us the packet movement that the wire shark captured.

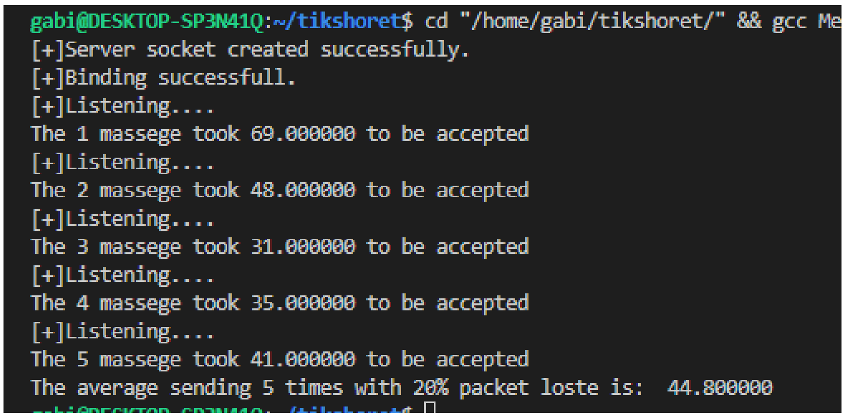


Enabled	Graph Name	Display Filter	Color	Style	Y Axis	Y Field	SMA Period	Y Axis Factor
✓	All Packets		⬛	Line	Packets		None	1
✓	TCP Errors	tcp.analysis.flags	🟥	Bar	Packets		None	1

20% packet lost:

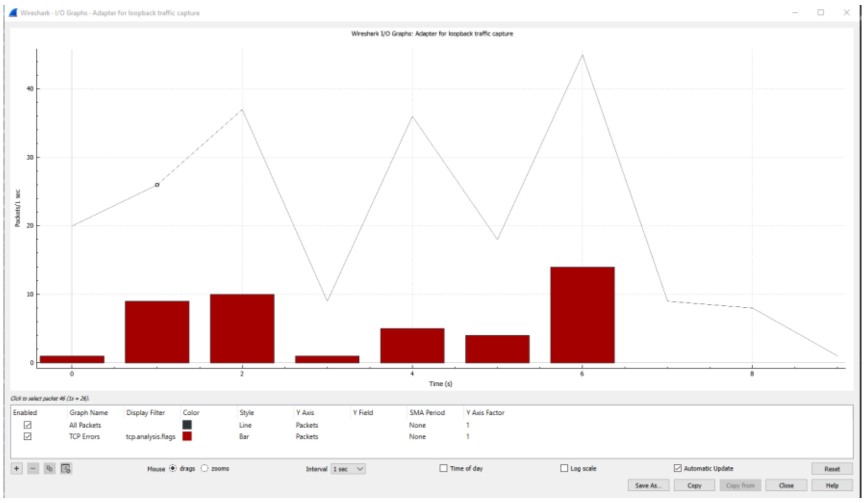
In the following image we can see the run time to each repeat (as we can see in the code above inside sender.c line 255. I used the setsockopt() to set the CC from "cubic" to "reno" also as explained above.

The last line showing the average time for all this massages with the correct precent.



```
gabi@DESKTOP-SP3N41Q:~/tikshoret$ cd "/home/gabi/tikshoret/" && gcc Me
[+]Server socket created successfully.
[+]Binding successfull.
[+]Listening....
The 1 massege took 69.000000 to be accepted
[+]Listening....
The 2 massege took 48.000000 to be accepted
[+]Listening....
The 3 massege took 31.000000 to be accepted
[+]Listening....
The 4 massege took 35.000000 to be accepted
[+]Listening....
The 5 massege took 41.000000 to be accepted
The average sending 5 times with 20% packet loste is:  44.800000
```

The following graph is showing the packet movement and the errors for 20 percent packet lost. Here already in a way that is clear to see the packet lost is bigger than the 10 percent packet lost at the biggening, we can also understand where we had a "break" that we stop to send (probably when we switched the CC in the run time).



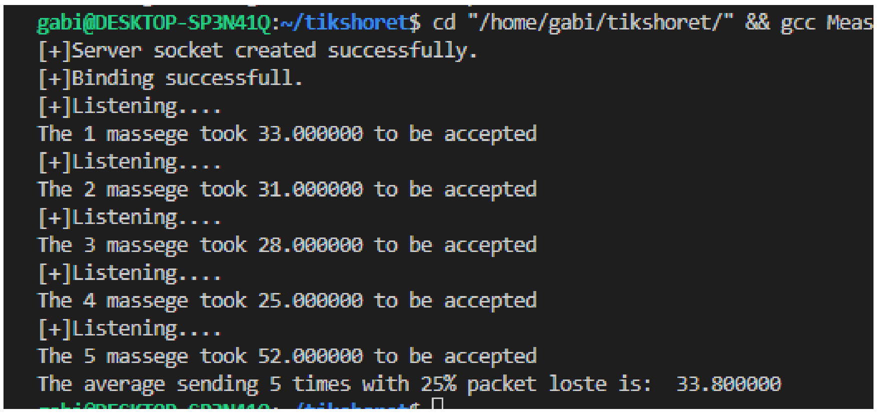
Enabled	Graph Name	Display Filter	Color	Style	Y Axis	Y Field	SMA Period	Y Axis Factor
✓	All Packets		■	Line	Packets		None	1
✓	TCP Errors	tcp.analysis.flags	🟥	Bar	Packets		None	1

Mouse (●) drags ( ) zooms Interval 1 sec ☐ Time of day ☐ Log scale ☑ Automatic Update

25% packet lost:

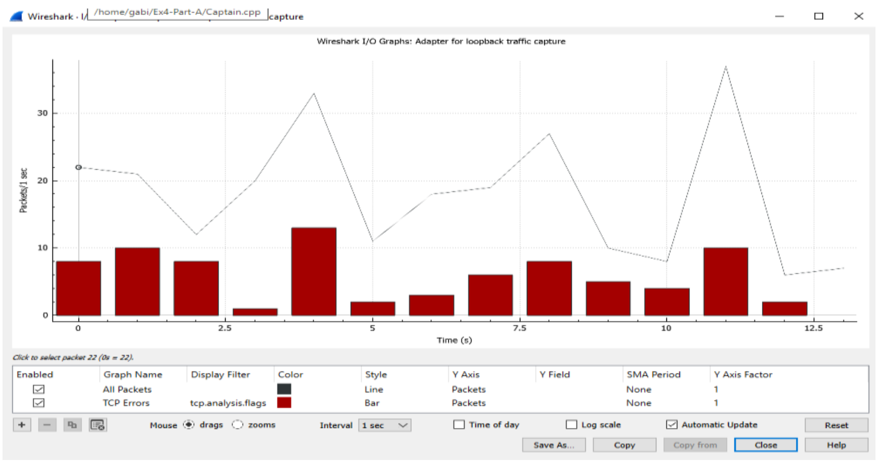
In the following image we can see the run time to each repeat (as we can see in the code above inside sender.c line 255. I used the setsockopt() to set the CC from "cubic" to "reno" also as explained above.

The last line showing the average time for all this massages with the correct precent.



```
gabi@DESKTOP-SP3N41Q:~/tikshoret$ cd "/home/gabi/tikshoret/" && gcc Meas
[+]Server socket created successfully.
[+]Binding successfull.
[+]Listening....
The 1 massege took 33.000000 to be accepted
[+]Listening....
The 2 massege took 31.000000 to be accepted
[+]Listening....
The 3 massege took 28.000000 to be accepted
[+]Listening....
The 4 massege took 25.000000 to be accepted
[+]Listening....
The 5 massege took 52.000000 to be accepted
The average sending 5 times with 25% packet loste is:  33.800000
```

This graph is showing already 25% packet lost which is already a big lost. We also can see already that the spreading of the red bars here is much higher. It is of course understandable because the lost is much higher so clearly we can see that we are losing much more even when we are sending less.



Click to select packet 22 (0s = 22).

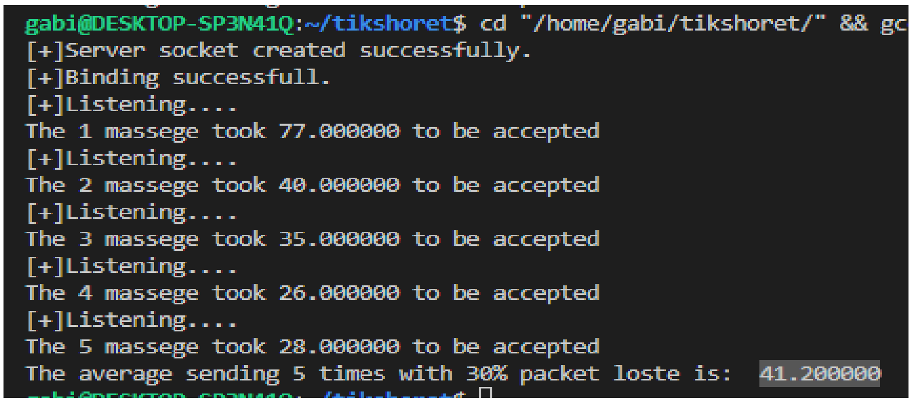
Enabled	Graph Name	Display Filter	Color	Style	Y Axis	Y Field	SMA Period	Y Axis Factor
✓	All Packets		■	Line	Packets		None	1
✓	TCP Errors	tcp.analysis.flags	🟥	Bar	Packets		None	1

Interval 1 sec; ☐ Time of day; ☐ Log scale; ☑ Automatic Update

30% packet lost:

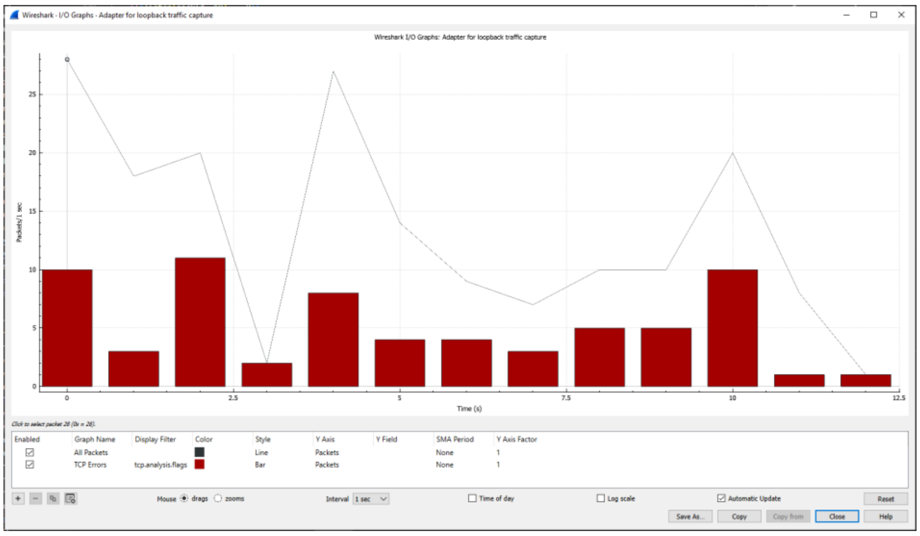
In the following image we can see the run time to each repeat (as we can see in the code above inside sender.c line 255. I used the setsockopt() to set the CC from "cubic" to "reno" also as explained above.

The last line showing the average time for all this massages with the correct precent.



```
gabi@DESKTOP-SP3N41Q:~/tikshoret$ cd "/home/gabi/tikshoret/" && gc
[+]Server socket created successfully.
[+]Binding successfull.
[+]Listening....
The 1 massege took 77.000000 to be accepted
[+]Listening....
The 2 massege took 40.000000 to be accepted
[+]Listening....
The 3 massege took 35.000000 to be accepted
[+]Listening....
The 4 massege took 26.000000 to be accepted
[+]Listening....
The 5 massege took 28.000000 to be accepted
The average sending 5 times with 30% packet loste is:  41.200000
```

This graph is showing the 30% packet lost. Clearly, we can see that the spreading is much bigger and the lost is much higher.

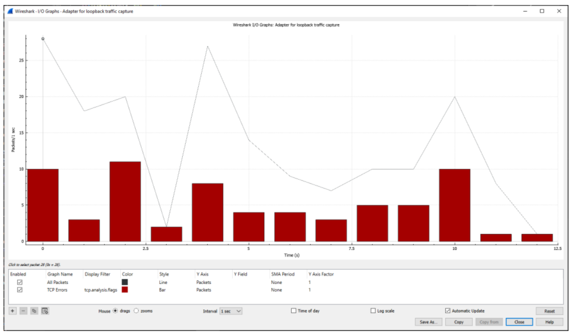


Click to select packet 28 (0s = 28).

Enabled	Graph Name	Display Filter	Color	Style	Y Axis	Y Field	SMA Period	Y Axis Factor
✓	All Packets		■	Line	Packets		None	1
✓	TCP Errors	tcp.analysis.flags	🟥	Bar	Packets		None	1

<u>Just to compere visually the packet lost I am showing the 10% packet lost near the 30-packet lost:</u>

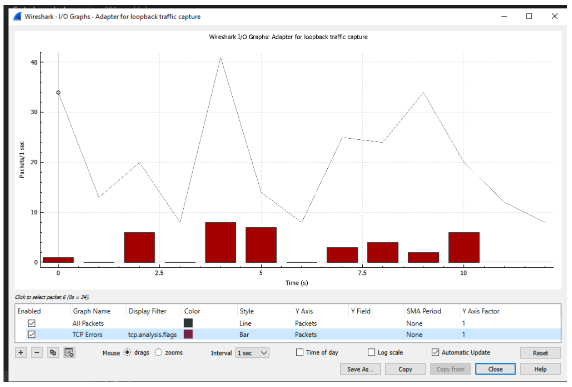
<u>30% packet lost:</u>


Wireshark · I/O Graphs · Adapter for loopback traffic capture
Wireshark I/O Graphs: Adapter for loopback traffic capture
Packets/1 sec
Time (s)

Click to select packet 28 (0s = 28).

Enabled	Graph Name	Display Filter	Color	Style	Y Axis	Y Field	SMA Period	Y Axis Factor
✓	All Packets		■	Line	Packets		None	1
✓	TCP Errors	tcp.analysis.flags	🟥	Bar	Packets		None	1

<u>10% packet lost:</u>



Click to select packet 6 (0s = 34).

Enabled	Graph Name	Display Filter	Color	Style	Y Axis	Y Field	SMA Period	Y Axis Factor
✓	All Packets		■	Line	Packets		None	1
✓	TCP Errors	tcp.analysis.flags	🟥	Bar	Packets		None	1

Clearly, we can see the differences visually, for bigger pictures and more explanations you can find above.