# Lab report 3b
# DVA218 Data communication
**Simon Roysson**
**Mareks Ozols**

## The program overview

Our goal with this C language program is to demonstrate safe data transmission between client and server by implementing a protocol stack on UDP. It should be used in Linux environment. The program is built with four program files: client.c, server.c, functions.c and header.h.  Client.c and server.c are main program files that implement socket communication with the help of a state machine which have three main states. First state is connection establishment which allows the client and the server to decide important parameters for communication. Second are sliding window state which ensure effective and secure communication between client and server. It uses the selective repeat protocol.  Third and last state is teardown state which will guarantee disconnection between client and server, then transmission is no longer needed between parties. Last two files header.h and functions.c are also important files in program. Header.h includes all inbuilt C language libraries, define all macros and declare all functions that will be used in program. Functions.c hold all definitions of functions.

## How use it the program

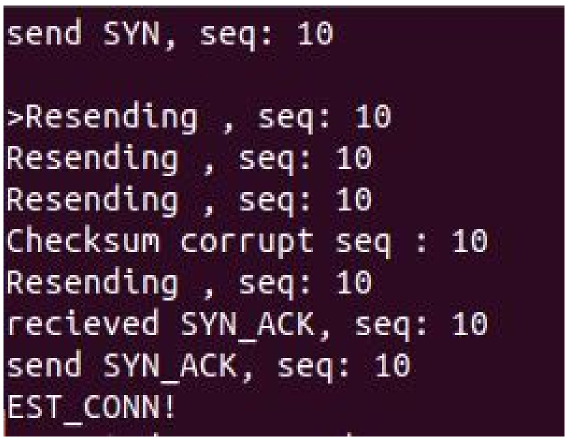
As mentioned before, the program should be used in a Linux environment. User should save all files in one directory. After that one should start two separate command line terminals, move to the directory where the files were saved and execute “make” command. It will enable to start server program in one terminal and client program in second terminal, it should be started in mentioned order.

## Output and details

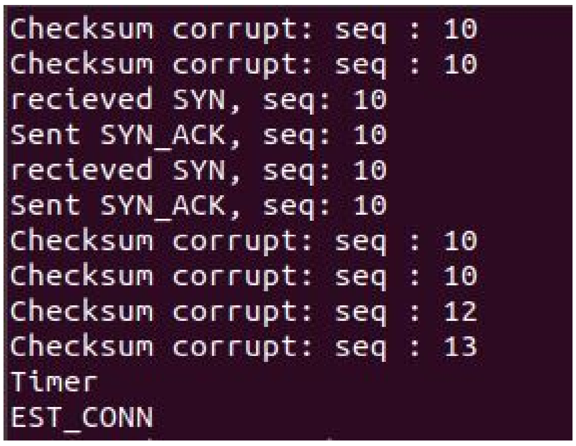
When both client and server programs are started. The demonstration output will follow in both client terminal and server terminal. In the header.h file are some macros that user should feel free to change. ClientWinSize and ServWinSize predefines client respective server sliding window size. By default ClientWinSize is 8 and ServWinSize is 4. These both will be used, then client connects to server. Client will send SYN with the desired window size( ClientWinSize) to server. Server will response to SYN with SYN_ACK and will define sliding window size (ServWinSize), this is the one that eventually will be used in communication. Next macro that is able to be changed is TIMEOUT which by default is set to 5. It is represented in seconds and controls how long time will pass before a new packet will be resent in case some has been lost or corrupt. The last macro that can be changed is Fail_Threshold - by default set to 0. It controls the rate in which different corrupt packets will be generated.

If program is executed with default parameters output will be following:
First it will show the establishment of connection. Client will send SYN and wait for SYN_ACK, when server receives SYN it will send SYN_ACK. Then the client receives the SYN_ACK and it will respond with last SYN_ACK to server and start timer. It can take a few seconds before client goes to state EST_CONN. Reason why we have chose to use timer is, that if last SYN_ACK from client to server is lost it will give time to server to resend SYN_ACK then timer is triggered and client receive it. In that case client will resend last SYN_ACK . It guarantees safe three way handshake and connection is established from both sides.  After that follows inbuild communication between both sides. User will be able to observe smooth packet exchange. There will be output showing sent packets with sequence numbers and expected ACK's sequence numbers on client side. Sending continuously until window is full, then client need wait for expected ACK to be able slide the window and continue sending. On server side the user can notice received packets with included data, sequence number of packet and expected packets sequence number as well as ACK that has been sent.
At the end of inbuilt communication user is able to write something, that will be sent to server and get acknowledgment from server about sent packet. If user choose to write "disconnect" it will trigger teardown and will disconnect client from server. Client sends FIN to server and waits for FIN_ACK. Server receives FIN and will send FIN_ACK and FIN. Client gets FIN_ACK and FIN and after that it will send last FIN_ACK and start timer in case last FIN_ACK have been lost in transaction. As mentioned before it will only happen in case of fail free communication. Next we will describe differences and show screenshots from program when fail occurs, packets get lost or corrupt, in all three client and server main states - establish connection, sliding window and teardown .

**First scenario when fail with packets take place at connection establishment.**



```
send SYN, seq: 10

>Resending , seq: 10
Resending , seq: 10
Resending , seq: 10
Checksum corrupt seq : 10
Resending , seq: 10
recieved SYN_ACK, seq: 10
send SYN_ACK, seq: 10
EST_CONN!
```

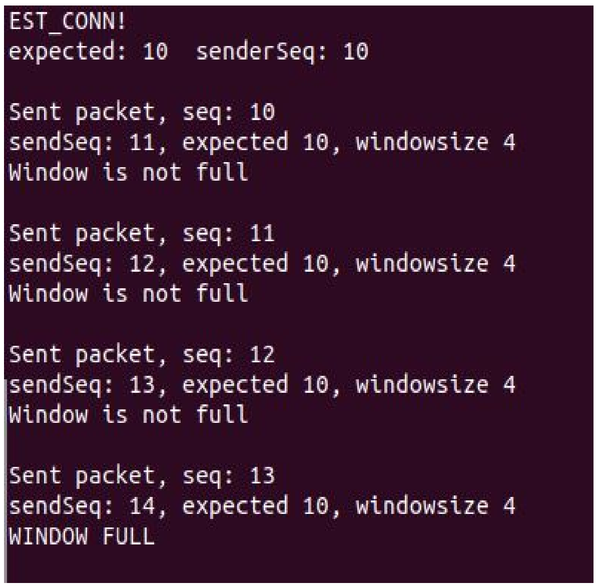


```
Checksum corrupt: seq : 10
Checksum corrupt: seq : 10
recieved SYN, seq: 10
Sent SYN_ACK, seq: 10
recieved SYN, seq: 10
Sent SYN_ACK, seq: 10
Checksum corrupt: seq : 10
Checksum corrupt: seq : 10
Checksum corrupt: seq : 12
Checksum corrupt: seq : 13
Timer
EST_CONN
```

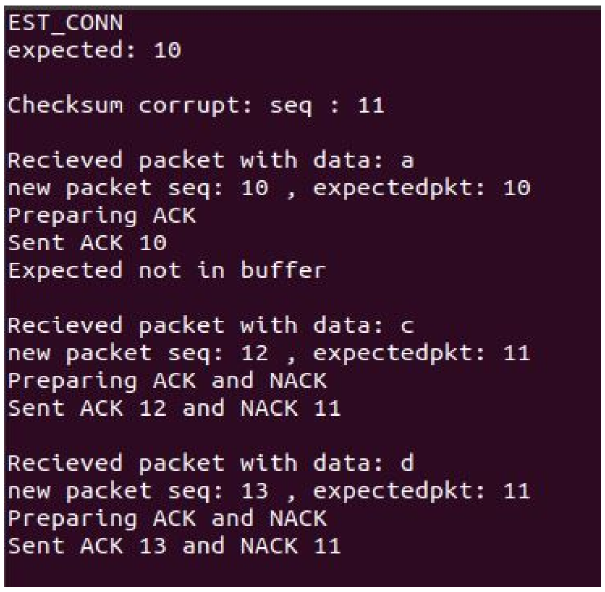
Figure 1. Shows connection establishment between client (left side) and  server(right side).

Client sends SYN to server, server does not accept it because checksum is corrupt. After some time, timeout will be triggered and client will send it again. Server gets it and send back SYN_ACK, but it also seems to be corrupt and client don't get it. Timeout is activated again and client sends another SYN. That will cause the server to send SYN_ACK again. Client receives it this time and is able to change state to established connection. When server had sent last SYN_ACK that was accepted by client, at the same time it started a timer that allows it go in established connection state in case the final SYN_ACK does not arrive at server. We have chose to implement timeouts in case, as it shows this example, client do not get SYN_ACK and sends SYN again.

**Next scenario is then packets disappears or are corrupt at sliding window state.**



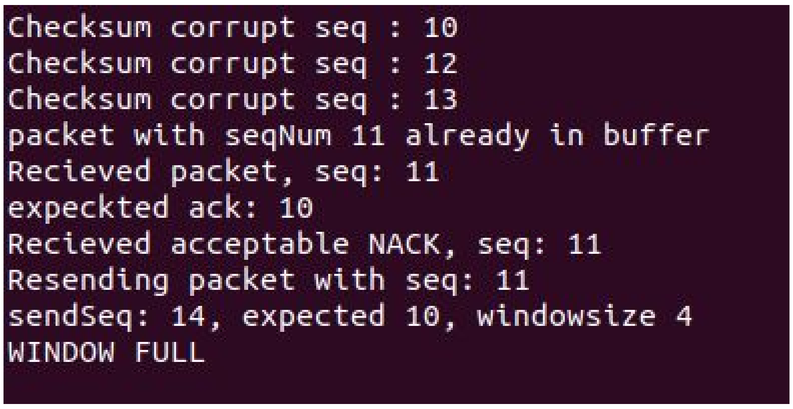
```
EST_CONN!
expected: 10  senderSeq: 10

Sent packet, seq: 10
sendSeq: 11, expected 10, windowsize 4
Window is not full

Sent packet, seq: 11
sendSeq: 12, expected 10, windowsize 4
Window is not full

Sent packet, seq: 12
sendSeq: 13, expected 10, windowsize 4
Window is not full

Sent packet, seq: 13
sendSeq: 14, expected 10, windowsize 4
WINDOW FULL
```



```
EST_CONN
expected: 10

Checksum corrupt: seq : 11

Recieved packet with data: a
new packet seq: 10 , expectedpkt: 10
Preparing ACK
Sent ACK 10
Expected not in buffer

Recieved packet with data: c
new packet seq: 12 , expectedpkt: 11
Preparing ACK and NACK
Sent ACK 12 and NACK 11

Recieved packet with data: d
new packet seq: 13 , expectedpkt: 11
Preparing ACK and NACK
Sent ACK 13 and NACK 11
```

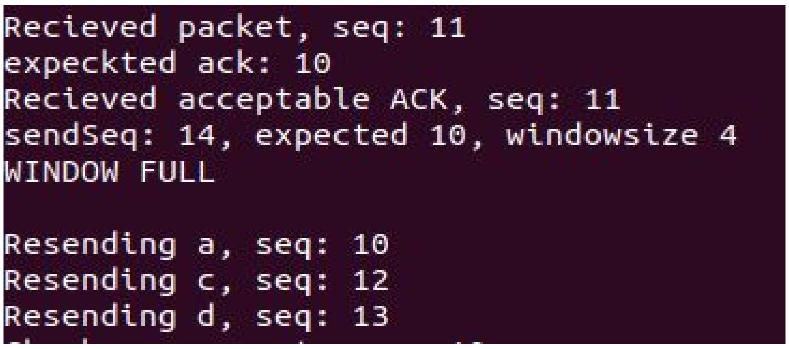
Figure 2. Represents how client(left side) sends packets until there is no more free space in sliding window. Server (right side) get those packet but some of them are corrupt.

As we can see in figure 2 client sends 4 packets to server with sequence numbers from 10 to 13. When server get those packet it shows that packet with sequence number 11 is corrupted and is dropped immediately. Packet with sequence number 10 is accepted and ACK has been sent as it should. In a different way packets with sequence number 12 and 13 have been accepted saved in accepted packets buffer and ACK's have been sent, but in both cases NACKS have been sent too, about packet 11. Reason to that is the expected packet that server wants to get next is 11, but instead have got 12 and 13. It represents selective repeat protocol.
In next figure 3 we can observe how client gets corrupt ACK packet with sequence numbers 10,12 and 13 and NACK on packet 11. Client checks the buffer and is able to find packet 11 that will be resend corresponding to NACK. We can see as well that sliding window is full and expected ACK packet that client is waiting for is with sequence number 10.
What we can read from figure 4 is the following, client gets ACK on packet 11, that was resent. After that we see timeouts have been triggered on packets 10, 12 and 13 so those have been resent as well.



```
Checksum corrupt seq : 10
Checksum corrupt seq : 12
Checksum corrupt seq : 13
packet with seqNum 11 already in buffer
Recieved packet, seq: 11
expeckted ack: 10
Recieved acceptable NACK, seq: 11
Resending packet with seq: 11
sendSeq: 14, expected 10, windowsize 4
WINDOW FULL
```

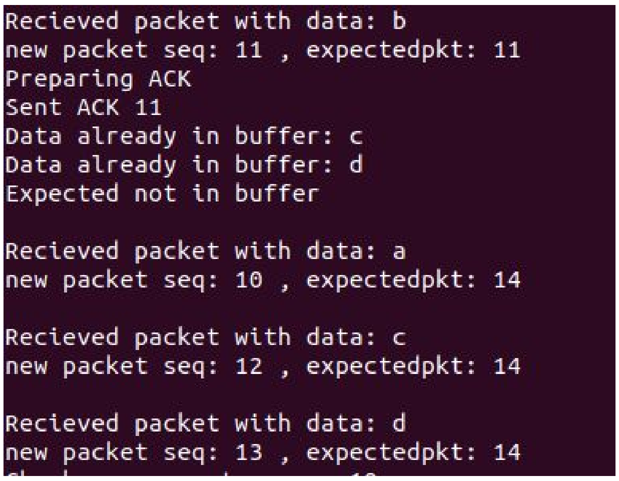
Figure 3. Client receives ACK's and NACK's.



```
Recieved packet, seq: 11
expeckted ack: 10
Recieved acceptable ACK, seq: 11
sendSeq: 14, expected 10, windowsize 4
WINDOW FULL

Resending a, seq: 10
Resending c, seq: 12
Resending d, seq: 13
```

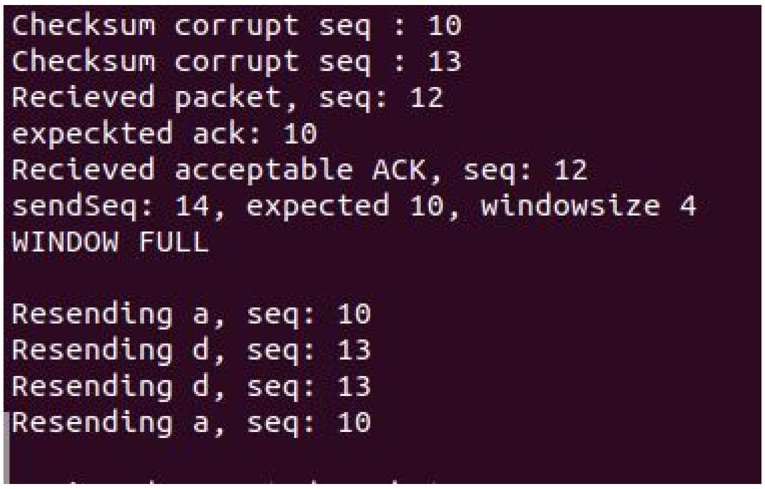
Figure4. Client receives ACK and resends some packets.

As we can see in figure 5, server gets packet with sequence number 11. It is packet that server was waiting for. When server gets that packet, expected packets sequence number will be updated to next and buffer for acceptable packet will be checked. In this particular case, there are two packet in a row, so next expected packet will be updated from 11 to 14. We can also see that there are coming in som packets that server already have. This means that server will drop those packets but ACK's will still be sent to client. It will guarantee that client do not end up on infinit loop.



```
Recieved packet with data: b
new packet seq: 11 , expectedpkt: 11
Preparing ACK
Sent ACK 11
Data already in buffer: c
Data already in buffer: d
Expected not in buffer

Recieved packet with data: a
new packet seq: 10 , expectedpkt: 14

Recieved packet with data: c
new packet seq: 12 , expectedpkt: 14

Recieved packet with data: d
new packet seq: 13 , expectedpkt: 14
```

Figure 5. Server get expected packet and some packet that it already have.


```
Checksum corrupt seq : 10
Checksum corrupt seq : 13
Recieved packet, seq: 12
expeckted ack: 10
Recieved acceptable ACK, seq: 12
sendSeq: 14, expected 10, windowsize 4
WINDOW FULL

Resending a, seq: 10
Resending d, seq: 13
Resending d, seq: 13
Resending a, seq: 10
```

Figur 6. Client receives some corrupt and one acceptable packet.

In figure 6  we can see that client gets corrupt packets with sequence number 10 and 13 but also non-corrupt packet 12 that has been accepted. There are also some resendings, because the timeout of those packets has been triggered. It points to that the fail rate is quite high (at this exact presentation 60 %) and it is quite obvious that most of the packets will be dropped or corrupted.

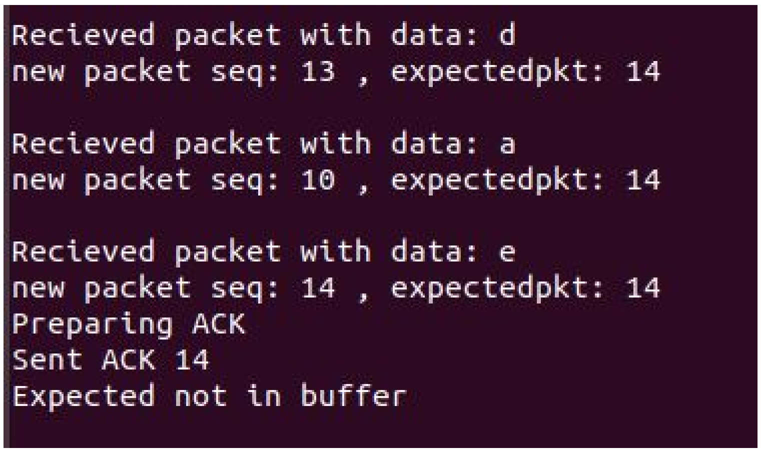
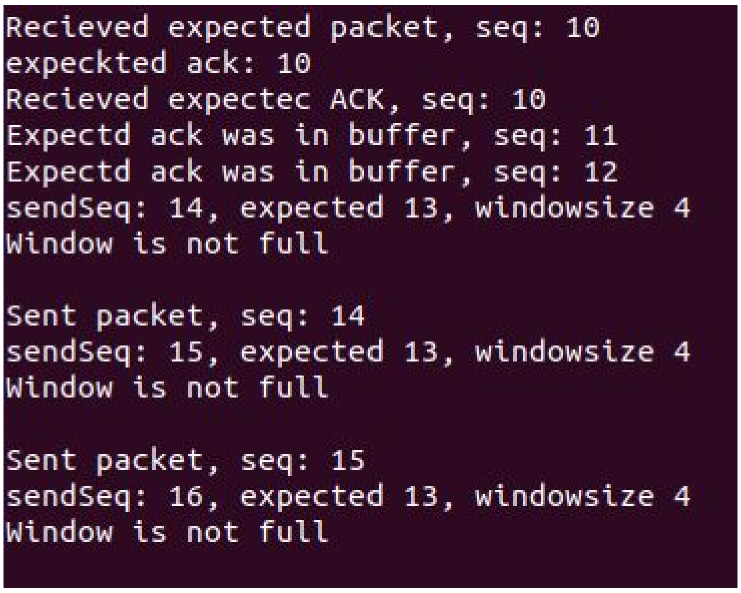

```
Recieved packet with data: d
new packet seq: 13 , expectedpkt: 14

Recieved packet with data: a
new packet seq: 10 , expectedpkt: 14

Recieved packet with data: e
new packet seq: 14 , expectedpkt: 14
Preparing ACK
Sent ACK 14
Expected not in buffer
```

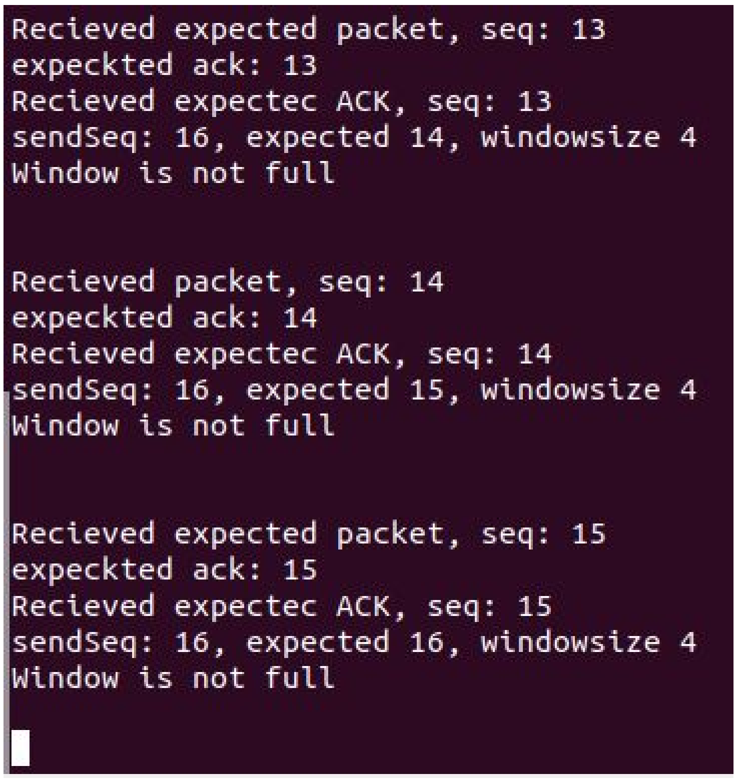
Figure 7. Represents the same pattern of server communication as described before until all packets are received.

As we can see in figure 8 as soon as client will receive expected ACK packet sliding window will start to change. In this case client gets ACK on packet with sequence number 10, that actually is the first packet that client has been sent. Sliding windows have been slided and expected sequence number updated. After that, acceptable packets buffer has been checked and expected sequence number have been updated two more times, because there was already saved ACK's in buffer. Last two packet have been sent.



```
Recieved expected packet, seq: 10
expeckted ack: 10
Recieved expectec ACK, seq: 10
Expectd ack was in buffer, seq: 11
Expectd ack was in buffer, seq: 12
sendSeq: 14, expected 13, windowsize 4
Window is not full

Sent packet, seq: 14
sendSeq: 15, expected 13, windowsize 4
Window is not full

Sent packet, seq: 15
sendSeq: 16, expected 13, windowsize 4
Window is not full
```

Figure 8. Shows how sliding window on client side has been changed from full window to window is not full.

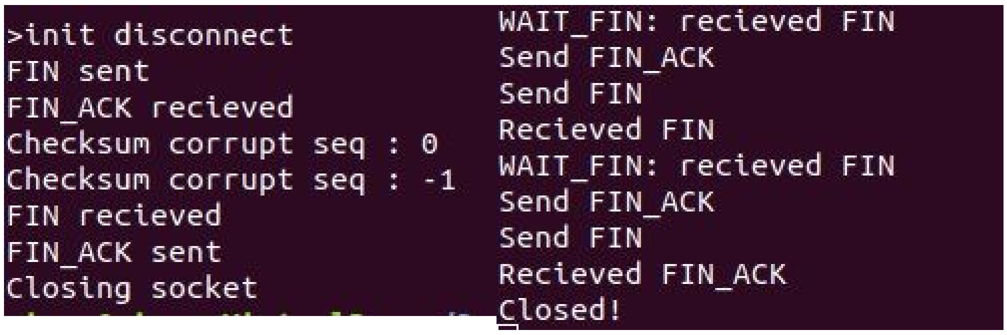


```
Recieved expected packet, seq: 13
expeckted ack: 13
Recieved expectec ACK, seq: 13
sendSeq: 16, expected 14, windowsize 4
Window is not full


Recieved packet, seq: 14
expeckted ack: 14
Recieved expectec ACK, seq: 14
sendSeq: 16, expected 15, windowsize 4
Window is not full


Recieved expected packet, seq: 15
expeckted ack: 15
Recieved expectec ACK, seq: 15
sendSeq: 16, expected 16, windowsize 4
Window is not full
```

Figure 9. Represents how client are receiving the last  ACK’s in the same way as described before.

**Last scenario when packets have been lost or corrupt at teardown state.**

As mentioned before, when inbuild communication is done, user is able to write some sentences and send them to server. In case user write the word “disconnect”, the teardown will take place.
As we can see in figure 10 client change its state to teardown as init disconnect symbolizes and sends FIN to server. Server will change state to teardown as soon it gets FIN packet and will send FIN_ACK and it’s own FIN to client. Client will wait for FIN_ACK to its own FIN and server FIN. In this case first FIN_ACK and FIN is corrupt packets and client does not receive them. After some time, timer will be triggered and client will resend the FIN packet. Server waits for FIN_ACK, but get instead same FIN packet again. That will make server to resend FIN_ACK and FIN to client. This time it is success and client will send last FIN_ACK to server as well as start timer. This timer will ensure that in case the server do not receive last FIN_ACK it can resend FIN_ACK and FIN to client (it is not happening in this particular example). As soon server receives last FIN_ACK it will disconnect immediately. Client will do it after the timer will end.



```
>init disconnect
FIN sent
FIN_ACK recieved
Checksum corrupt seq : 0
Checksum corrupt seq : -1
FIN recieved
FIN_ACK sent
Closing socket
```

```
WAIT_FIN: recieved FIN
Send FIN_ACK
Send FIN
Recieved FIN
WAIT_FIN: recieved FIN
Send FIN_ACK
Send FIN
Recieved FIN_ACK
Closed!
```

Figure 10. Client(left side) and Server(right side) goes into teardown state and ends the communication with each other with help of FIN and FIN_ACK packets.

**Differences from our theoretical state machine and state machine that is implemented in program.**

When we were done with theoretical client and server state machines we decided to make a code skeleton of them. So we wrote them in code in client.c and server.c files. After that we decided to split the code into function.c and header.h files. Reason of that was - client and server uses some same functions to communicate with each other. After that we started to implement functions that have been used in state machines, to simulate client- server communication. We were able to implement the code quite close to our theoretical state machine. The one biggest and only difference is that in our theoretical state machine both the client side and server side, in all three main states, had intended to use a substate called “Read Checksum” which would check if packets were corrupt. Instead we choose to do it where packets come in from socket which is done in a completely separate thread. This way our state machine does not have to check for corrupt packet, all packets have already been checked before reaching the state machine. Otherwise our implemented code acts same as we had designed it in our theoretical state machine.