**PKS project 2**

**Communication through UPD protocol**

**Final documentation**

Márk Bartalos

Seminars with**: Ing. Lukáš Mastiľak**Seminars on**: Thu 14:00-15:50**

Table of content

[Introduction 4](#_Toc121409302)

[Header 4](#_Toc121409303)

[1. Communication type (2B) 4](#_Toc121409304)

[2. Packet ID 4](#_Toc121409305)

[3. Data size 4](#_Toc121409306)

[4. Data type 4](#_Toc121409307)

[5. Data 5](#_Toc121409308)

[6. CRC 5](#_Toc121409309)

[Connection management 5](#_Toc121409310)

[Connection establishment 5](#_Toc121409311)

[Connection termination 6](#_Toc121409312)

[Successful data transfer 7](#_Toc121409313)

[Corrupted data transfer 9](#_Toc121409314)

[Data transfer timeout (Advanced ARQ) 11](#_Toc121409315)

[The receiver is not responding 12](#_Toc121409316)

[Keepalive 13](#_Toc121409317)

[Switch tasks 16](#_Toc121409318)

[CRC method 17](#_Toc121409319)

[Simulating Error 17](#_Toc121409320)

[Changes from the original design 18](#_Toc121409321)

[Header design 18](#_Toc121409322)

[Connection establishment 19](#_Toc121409323)

[Connection termination 19](#_Toc121409324)

[Data transfer timeout 20](#_Toc121409325)

[Switch task 21](#_Toc121409326)

[Activity chart 23](#_Toc121409327)

[Wireshark communication analysis – connection establishment 23](#_Toc121409328)

# Introduction

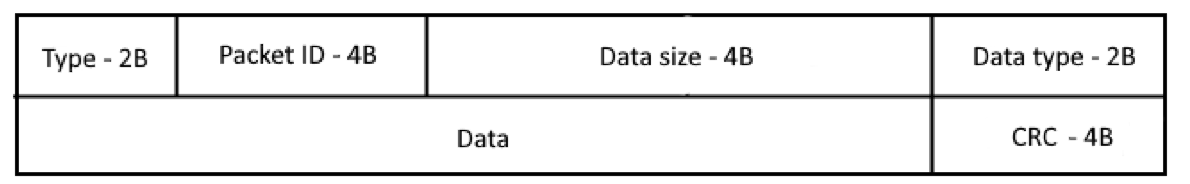
We have to design an application that uses our custom protocol under UDP. That application has to deliver text messages and files. The transfer of data will happen in binary form.

When we start the application, the application will ask if we want to use this instance as receiver or as sender. These states can be change while the program is running.

To be able to communicate and make sure the data is not corrupt we have to design a custom protocol header.

# Header

The protocol header will consist of 6 parts. Each part will have its appropriate bytes (size).



## Communication type (2B)

* 0 - request for synchronization (SYN)
* 1 – accept synchronization (SYN+ACK)
* 2 – OK response for keep-alive, response for package received and other types of requests
* 3 – end connection (FIN)
* 4 – resend data
* 5 – request for keep-alive (keep alive check)
* 6 – request for data transfer initialization
* 7 – data send
* 8 – switch tasks

## Packet ID

Will contain the unique identification number of the packet.

If the data is split into fragments (almost all the time), this will be used to track the number of fragments and to be used to reconstruct the data.

## Data size

This will contain the size of the data (or the fragments size) payload.

## Data type

* 0 – Text
* 1 – File transfer

## Data

This will contain the data (complete or fragment) itself.

## CRC

This will contain the Cyclic Redundancy Check code for the data to verify the correctness.

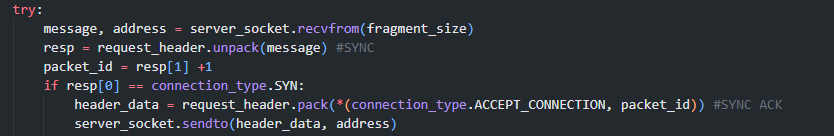
# Connection management

There will be different connection scenarios which will be defined in this section.

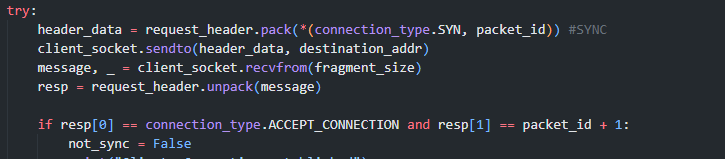
## Connection establishment

Successful connection establishment will consist of two (or maybe three, another ACK for security) requests. First the sender will send a synchronization request to the receiver, by initializing the request this will define the first device as the sender. The second device will have to reply with a SYNCHRONIZATION ACCEPT response for the connection to be successfully established. If the connection is established the server will now the clients IP address and port, that way it will be able to communicate with the client. More concretely the client will send the server a request for synchronization (SYNC), here the packet id is created by the client, then the server will reply to this request with SYNCHRONIZATION\_ACCEPT with the received packet id incremented by one. If no answer is received then the client will continue sending the requests.

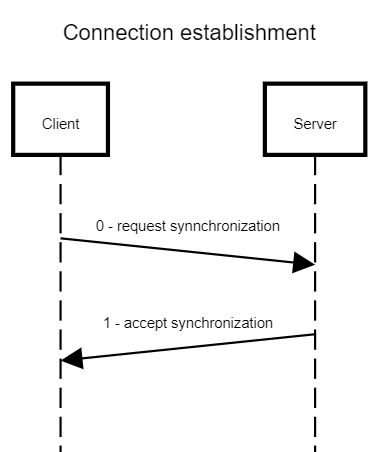
**Server**



**Client**

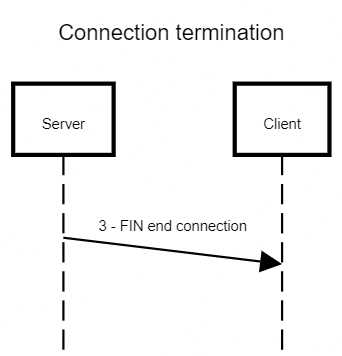


The client will wait for the SYNCHRONIZATION\_ ACCEPT packet and compare the packet ids.



## Connection termination

Connection termination similarly to establishment will consist of one request. The server will send a FIN request to the client and exit. The client wont reply with OK or anything, because its not necessary to both shut down at the same time. Either if the FIN arrives at the destination or lost, the connection will time out if either of the sides shut down.



## Successful data transfer

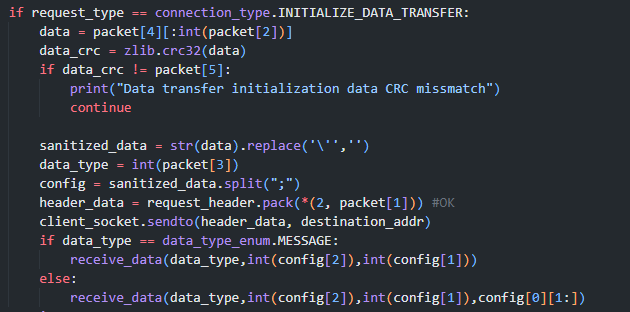
For a successful data transfer, every transfer has to be started with an initialize data transfer request and has to be accepted with an OK request, then all data fragment responses has to be followed with an OK request that the data is received and the checksum is ok. With the data its fragment number is sent and the OK request has too have the same fragment number to be counted as accepted packet.

**Server**

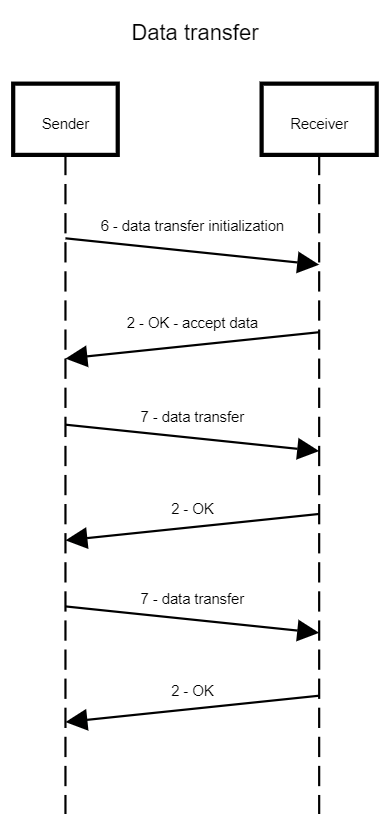


Here the server will send the initialization data as string with a 200B data header this will contain the file path of the file on the receivers end (in case of message it will be ‘\null\’), the total number of data fragments and the data fragment size. The client will extract this data and respond with an OK request if everything is correct and prepare for the file/message reception.

**Client**



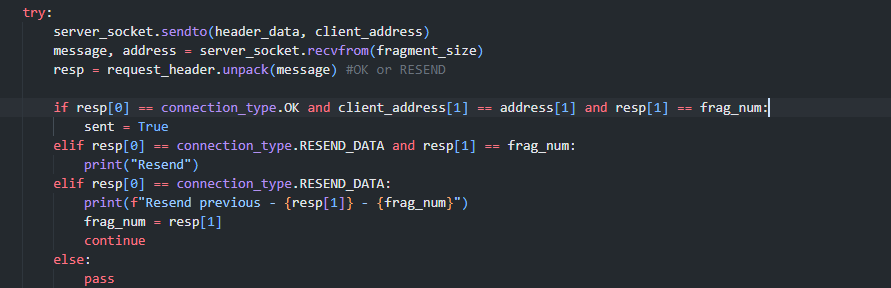
The client will do a CRC check on the data in the request, if there is a mismatch it will wait for the next INITIALIZE\_DATA\_TRANSFER request.



## Corrupted data transfer

If corrupted data is received, the checksum doesn’t match, then the receiver will send a RESEND DATA request back to the sender and the sender will send the data fragment again. This corruption will be simulated by randomly by manually changing the CRC value. Here with the RESEND DATA packet the id of the corrupted packet is send. If the corrupted packet was one of the previous packets than the server will go back and send that part again.

**Server**

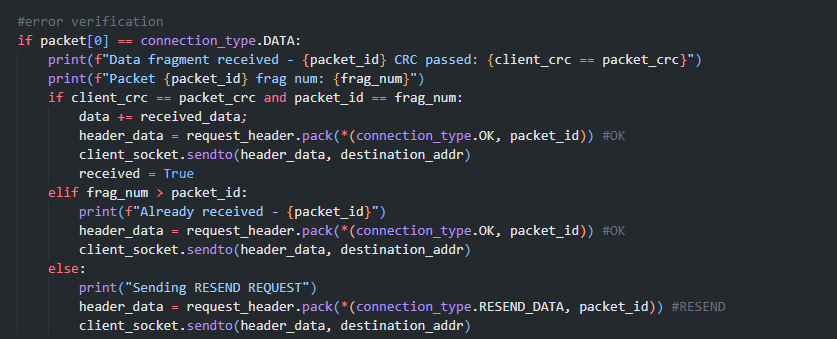


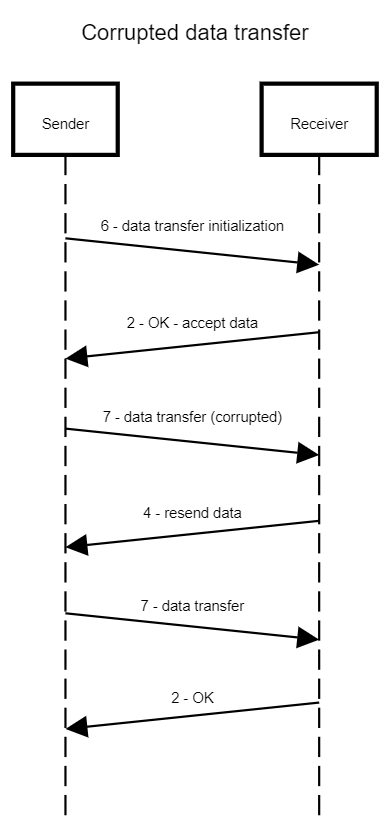
Here the server will check the clients port and the frag num if everything is Ok than the packet has been received successfully and goes to the next.

If the request is RESEND DATA with the current fragment id than it will resend the current fragment again.

If the request is RESEND DATA with previous fragment id than it will go back to that fragment and send it again.

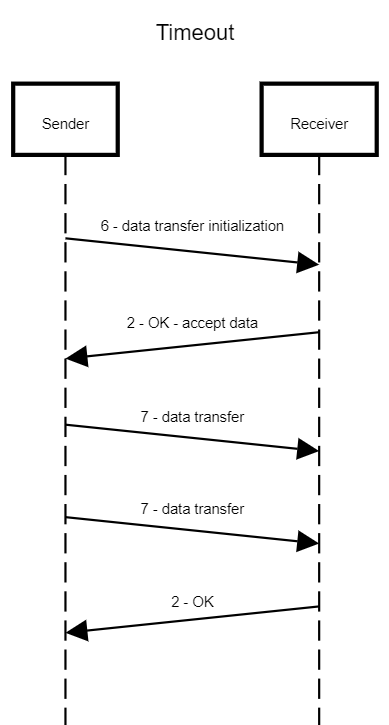
**Client**

****



## Data transfer timeout (Advanced ARQ)

If the sender doesn’t receive OK response after the data is sent again, the sender will wait the timeout amount and send the data again, this will have a limit. exists. The OK responses packet id has to be the data’s fragment number to confirm it (as you can see it the Corrupted data transfer section). Here the sending mechanism can go back and send again already sent packets if they are not received in the same way the client can send again an OK request if the data is already received but the OK request has been lost.



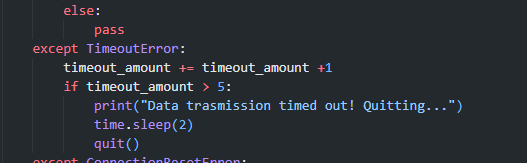
## The receiver is not responding

If the receiver doesn’t respond after several responses the server will time out and quit. After 15 packets receive timeout or the connection has been reset the program quits as the connection probably no longer exist. The data transfer will be aborted.

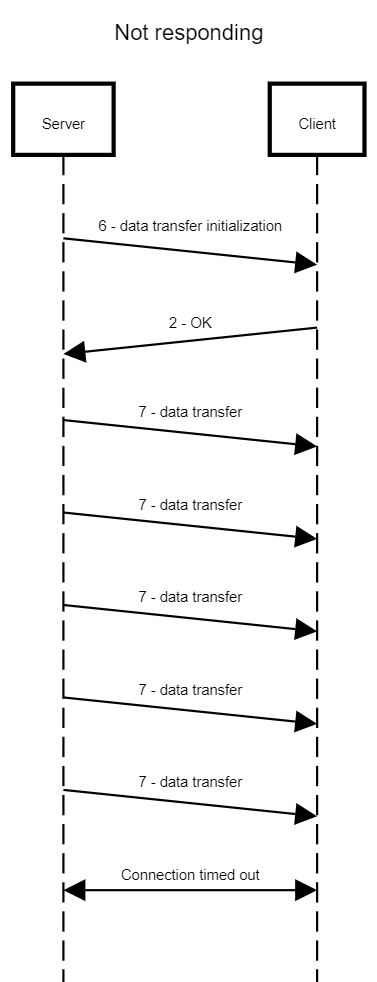
This will also apply for the client if data packets are not sent the client will eventually time out and quit.

During data transfer and initialization KEEPALIVE requests are not sent.

**Server**



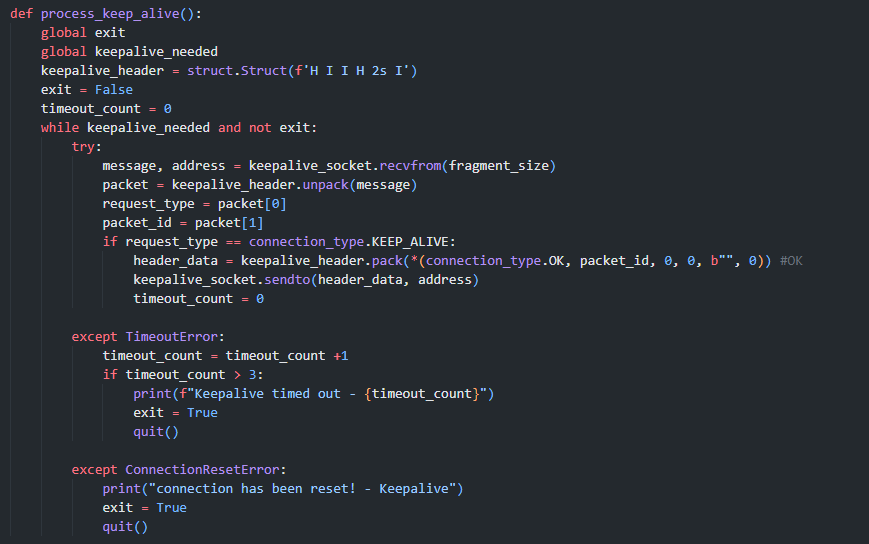
Basically, the same thing is used on the client side.

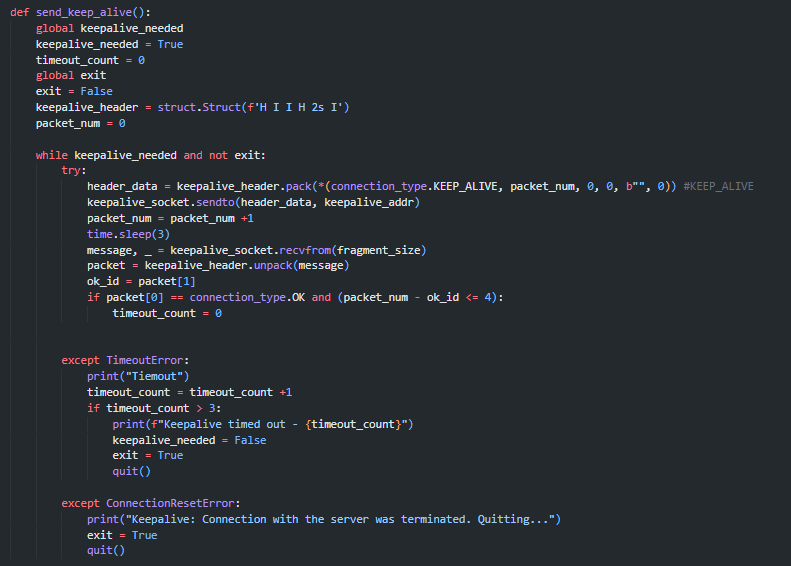


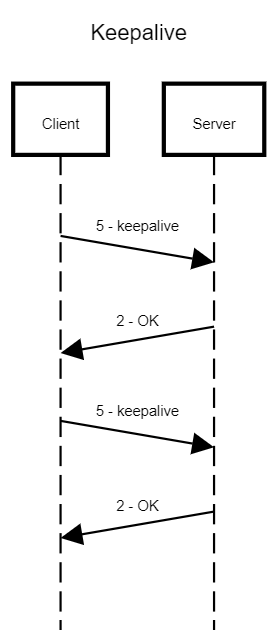
## Keepalive

Keepalive is solved using a separate socket on each side. The client socket will send a keepalive request to the server and the server will reply to that request with an OK. If the keepalive request doesn’t receive and OK request more than 3 times in a row the keepalive fails and the connection is terminated. Keepalive only runs if there is no data transfer happening, because if a data transfer is in progress there is already data being sent and we can track how many packet doesn’t receive a response and if the connection has timed out. Keepalive thread is stopped before data transfer and restarted after data transfer ended.

**Server**

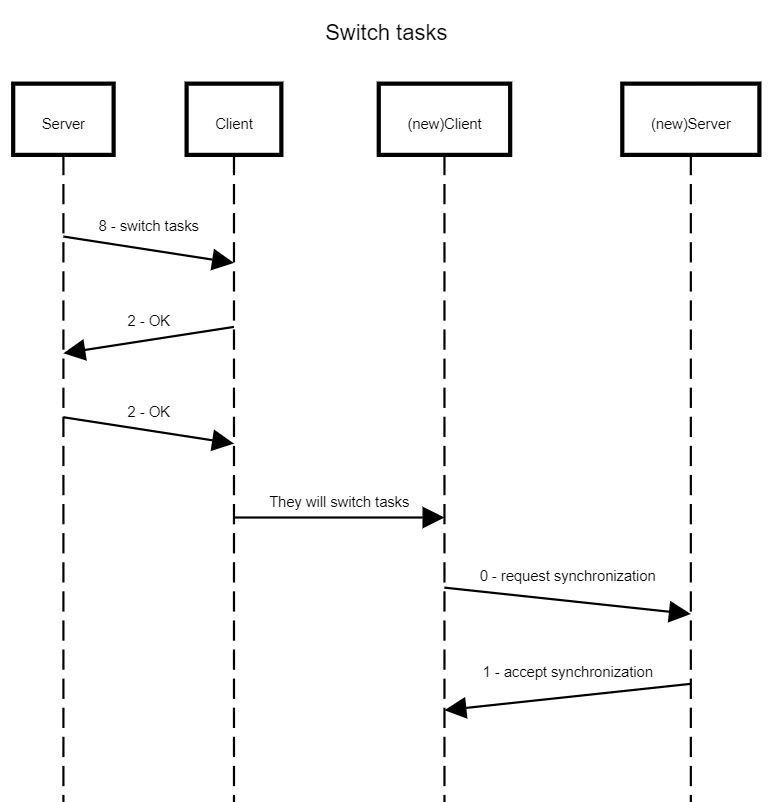


**Client**

****

## Switch tasks

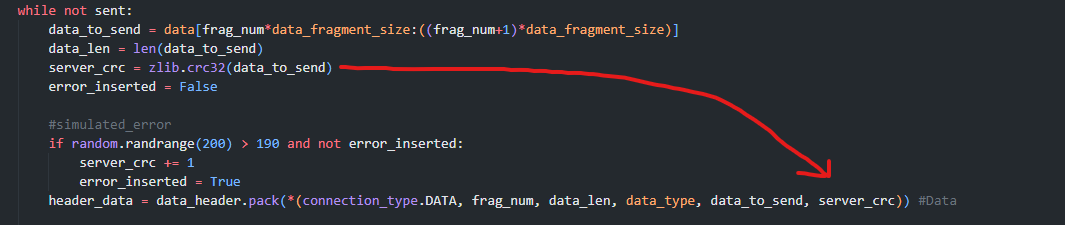
Switching task will start with a SWITCH TASK request. The SWITCH TASK request has to be accepted with an OK response and after that the server has to also respond with an OK request to be able to initialize the task switch. After the tasks have been switched, we begin an establishment of the connection again now as the roles switched. As part of SWITCH TASK request the server will send the client its own Ip address on which the new server (previously client) will listen on.



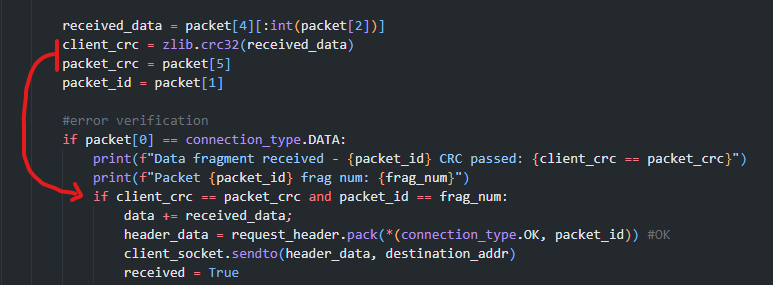
# CRC method

For CRC error correction I will use the **zlib.crc32** function on the data which will generate a CRC code based on the data in the packet. This CRC checksum will be sent and when the data arrives it will be checked against the data. This method of CRC will be able to spot 2-bit changes opposed to the traditional checksum method. The data will be compared bit by bit to generate the CRC checksum.

CRC is used when an important data/fragment is received a CRC checksum is generated on the server side which is included in the request header. Then the client will also generate a CRC checksum for the received data and these checksums will be compared.  
**Server**

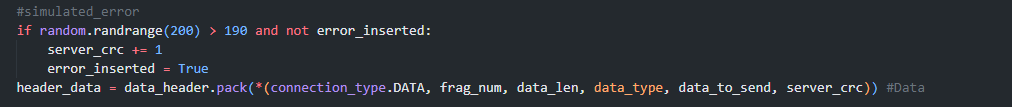


**Client**



### Simulating Error

CRC error simulation is done by manually incrementing the CRC code on the server side and sending it in the header. Each packets CRC can be incremented by only one.

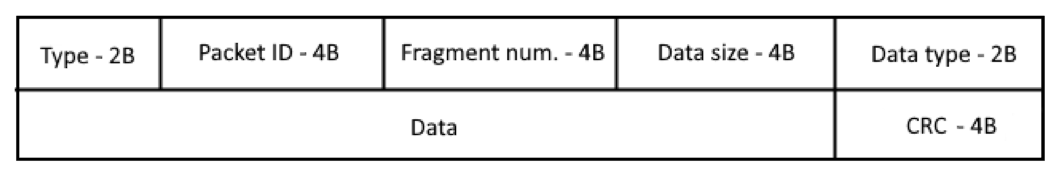


# Changes from the original design

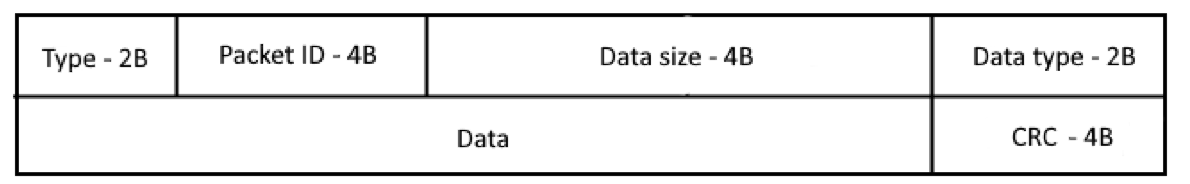
## Header design

The header has been simplified as there is no need for separate fragment num and packet id. Instead, a common packet id is used.

**OLD**



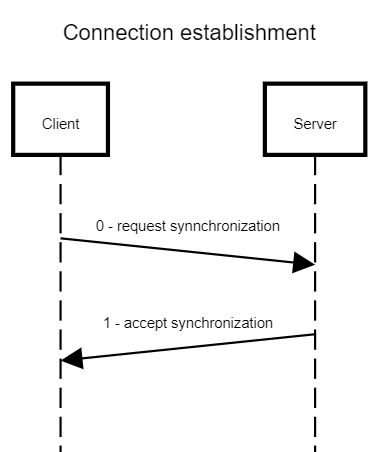
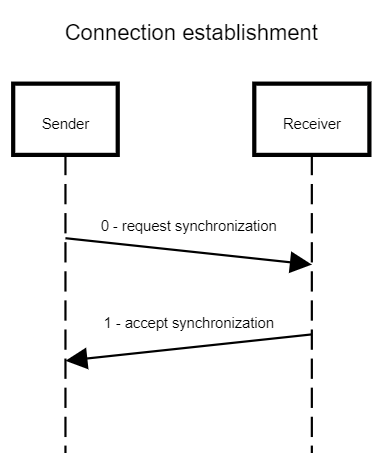
**NEW**



## Connection establishment

Now the client will initialize the connection

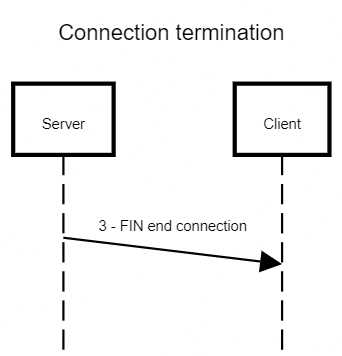
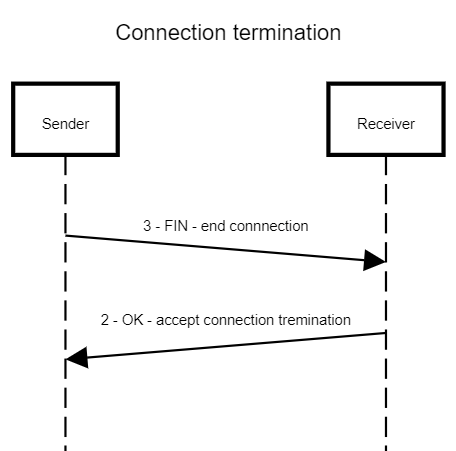
**OLD NEW**



## Connection termination

No need to accept connection termination.

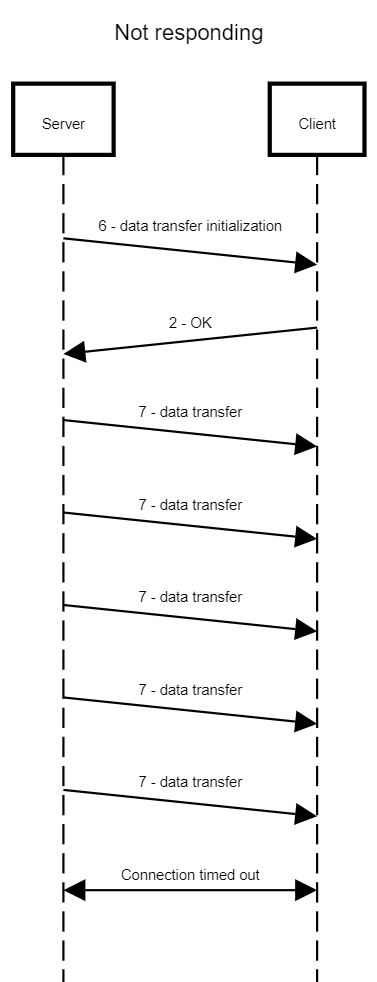
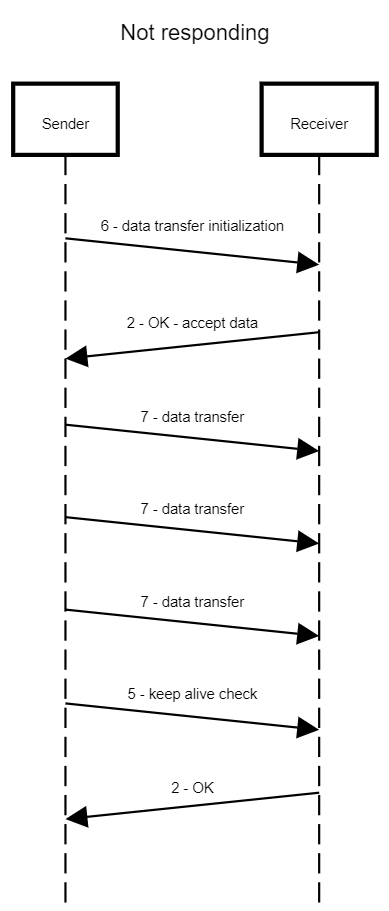
**OLD NEW**



## Data transfer timeout

No keepalive request is sent, timeout happens after 5 requests without OK pair.

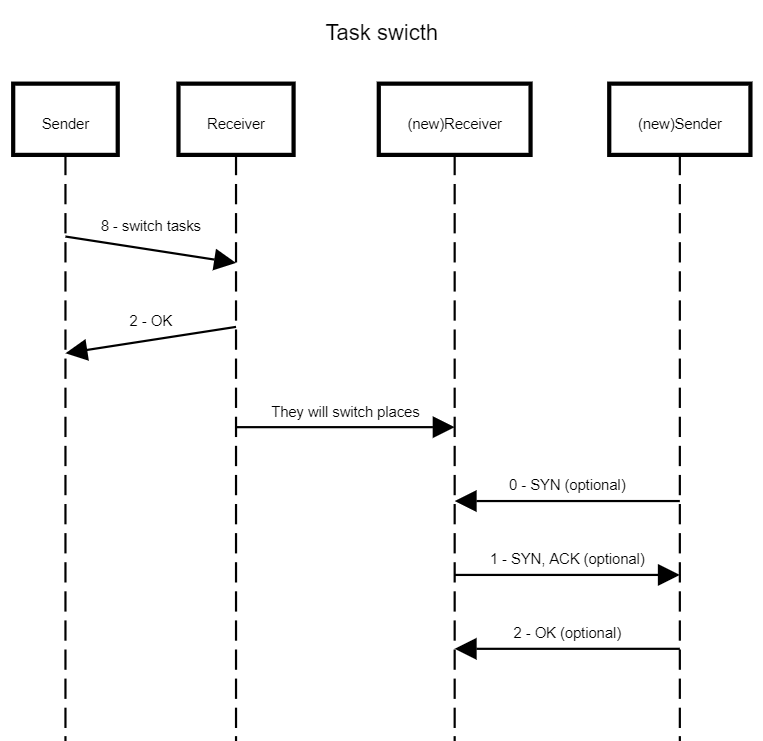
**OLD NEW**



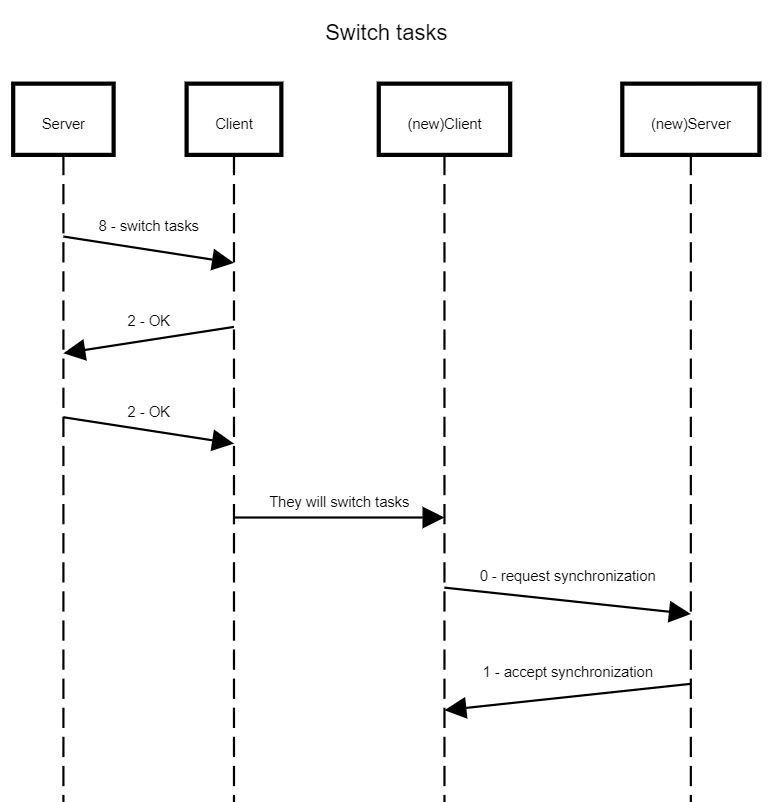
## Switch task

Re-synchronization after switch task won’t be optional but required. This will be done automatically. Another change that the server also has to confirm that the client got the task switch request. The synchronization process is also simpler.

**OLD**



**NEW**

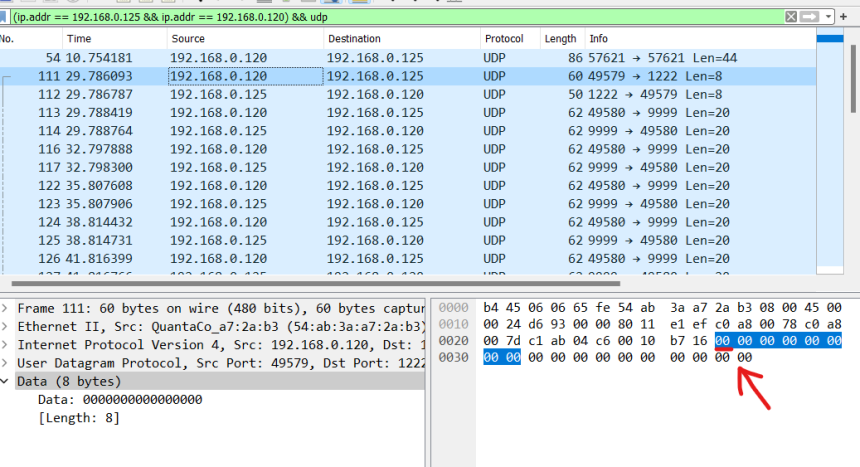


# Activity chart

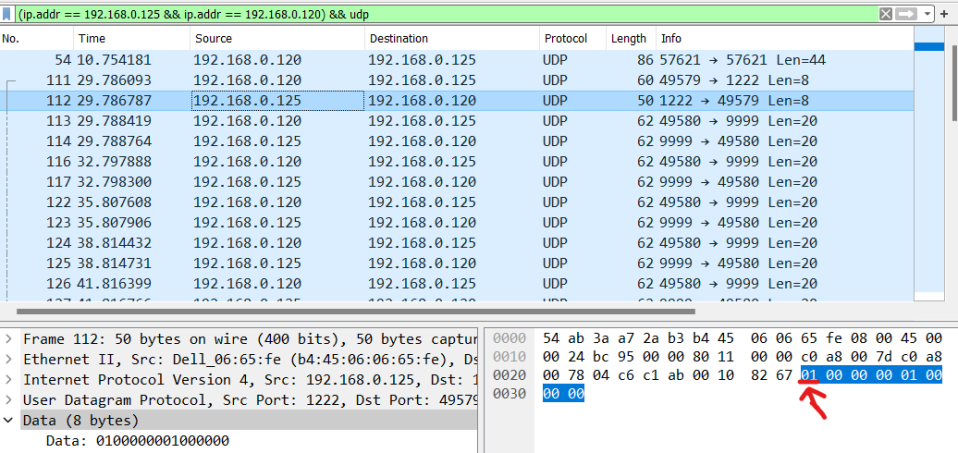


# Wireshark communication analysis – connection establishment

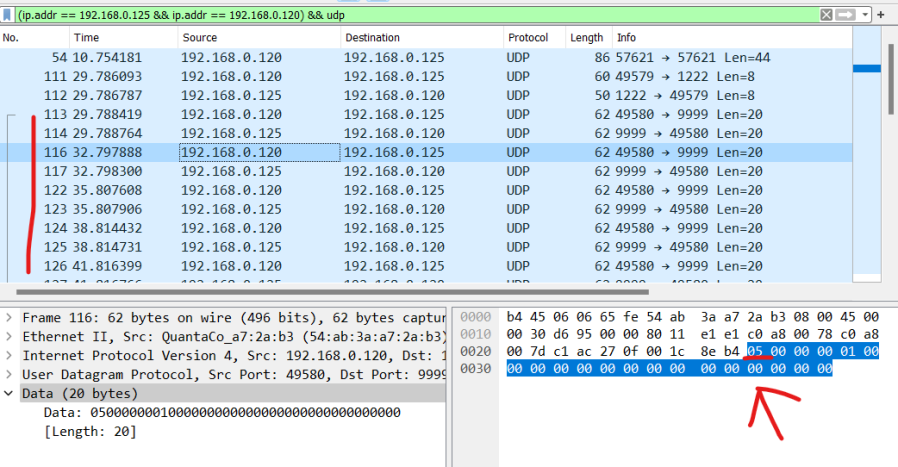
1. The client sends a SYNC request to the server. Here the server has the IP address 192.168.0.125 and is listening on port 1222. The request begins with a 0, which means SYNC

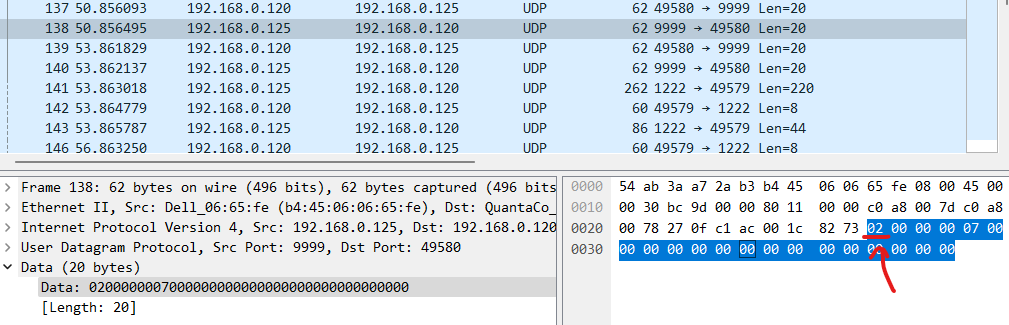


1. The server responds to the client with a CONNECTION ACCEPT request, which is code 1. After that the connection is established.

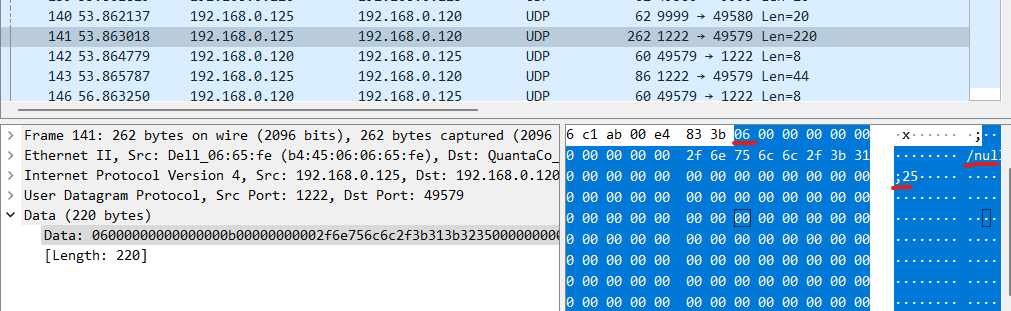


1. Start the keepalive process. The client will start sending keepalive packets and the server will reply to them with OK.

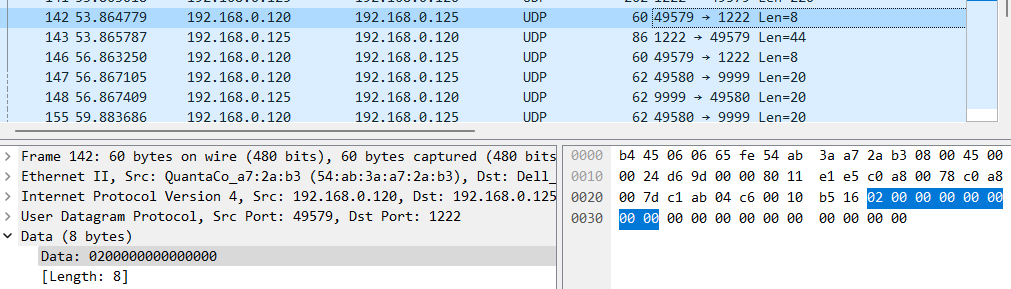




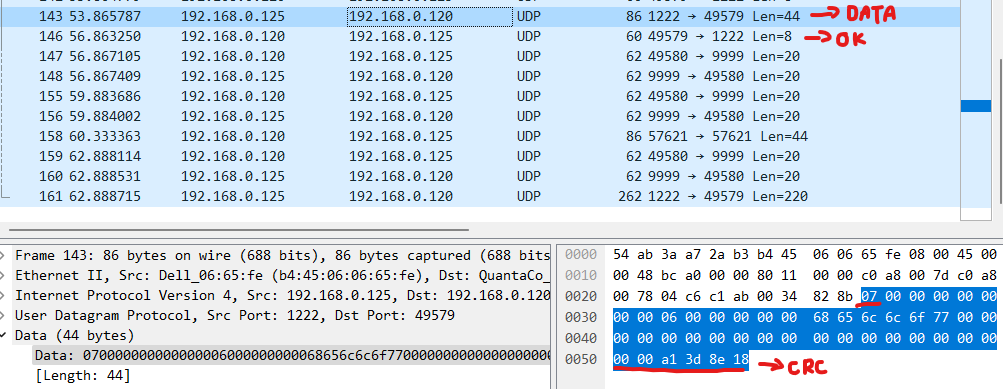
1. Initialize data transfer. Send important information about the data to be sent.



1. OK from the client.



1. Sending of the data packets. CRC is included in the packet. Server responds with OK.



1. End connection – exit program

