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

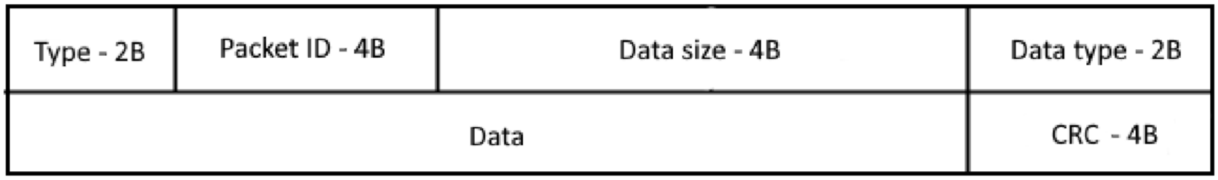
We have to design an application that uses our custom protocol under UDP. That application has to deliver text messages and files. The transfer of data will happen in binary form.

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).



Type - 2B	Packet ID - 4B	Data size - 4B	Data type - 2B
Data			CRC - 4B

1. 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

2. 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.

3. Data size

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

4. Data type

- 0 – Text
- 1 – File transfer

5. Data

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

6. CRC

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

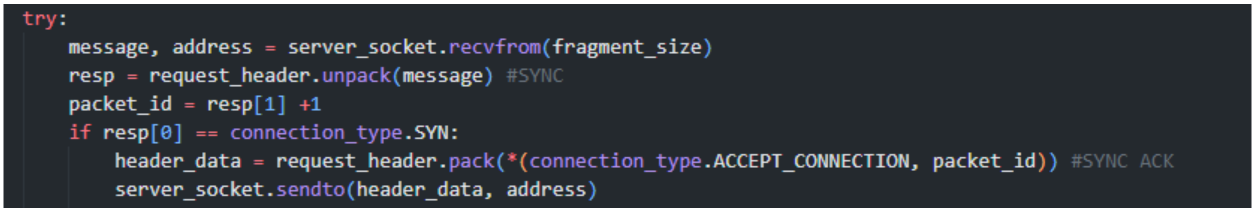
Connection management

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

Connection establishment

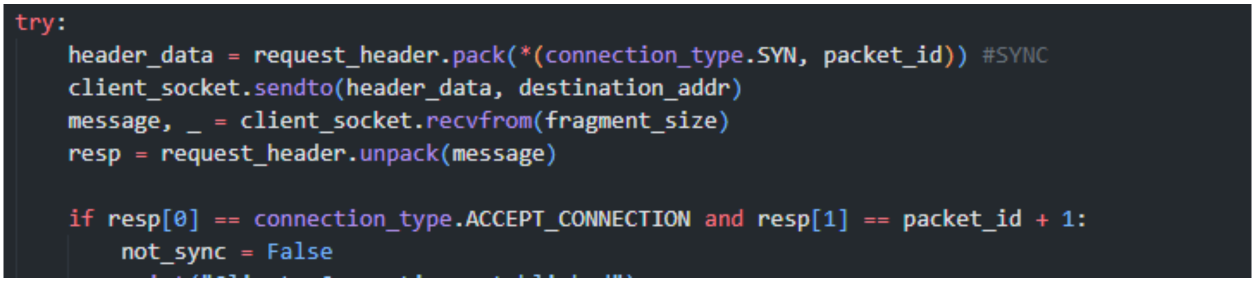
Successful connection establishment will consist of two (or maybe three, another ACK for security) requests. First the sender will send a synchronization request to the receiver, by initializing the request this will define the first device as the sender. The second device will have to reply with a SYNCHRONIZATION 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



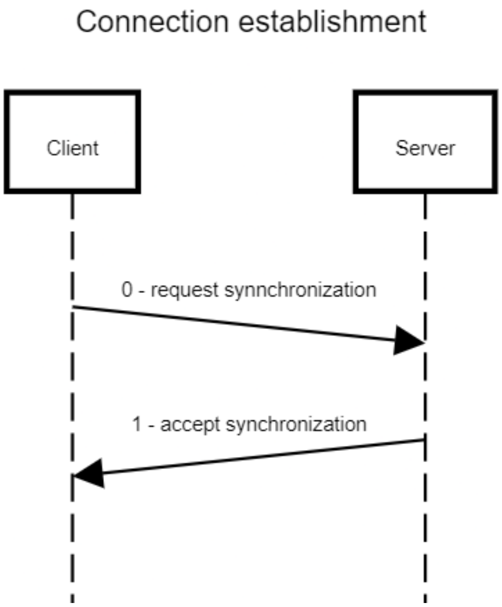
```
try:
    message, address = server_socket.recvfrom(fragment_size)
    resp = request_header.unpack(message) #SYNC
    packet_id = resp[1] +1
    if resp[0] == connection_type.SYN:
        header_data = request_header.pack(*(connection_type.ACCEPT_CONNECTION, packet_id)) #SYNC ACK
        server_socket.sendto(header_data, address)
```

Client



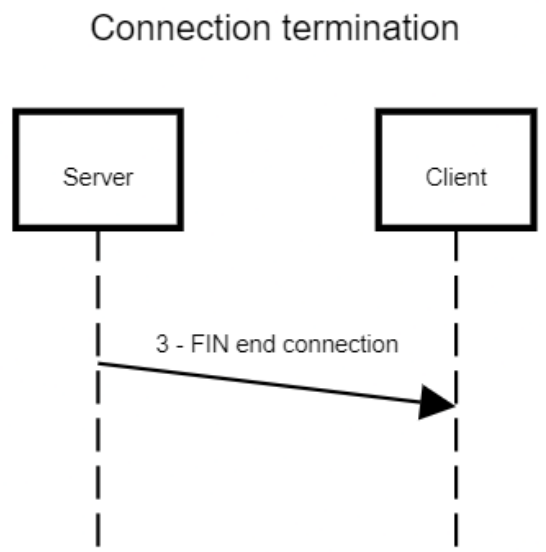
```
try:
    header_data = request_header.pack(*(connection_type.SYN, packet_id)) #SYNC
    client_socket.sendto(header_data, destination_addr)
    message, _ = client_socket.recvfrom(fragment_size)
    resp = request_header.unpack(message)

    if resp[0] == connection_type.ACCEPT_CONNECTION and resp[1] == packet_id + 1:
        not_sync = False
```

The client will wait for the SYNCHRONIZATION_ ACCEPT packet and compare the packet ids.



Connection termination

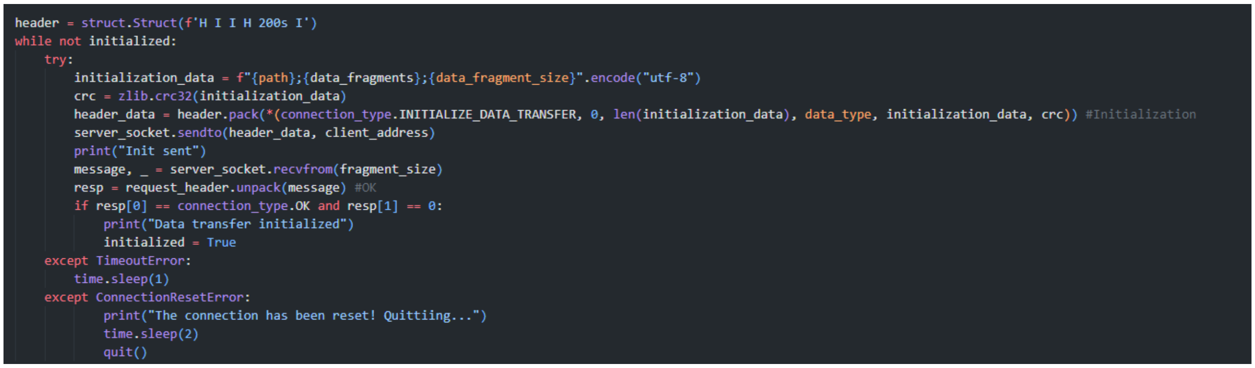
Connection termination similarly to establishment will consist of one request. The server will send a FIN request to the client and exit. The client wont reply with OK or anything, because its not necessary to both shut down at the same time. Either if the FIN arrives at the destination or lost, the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

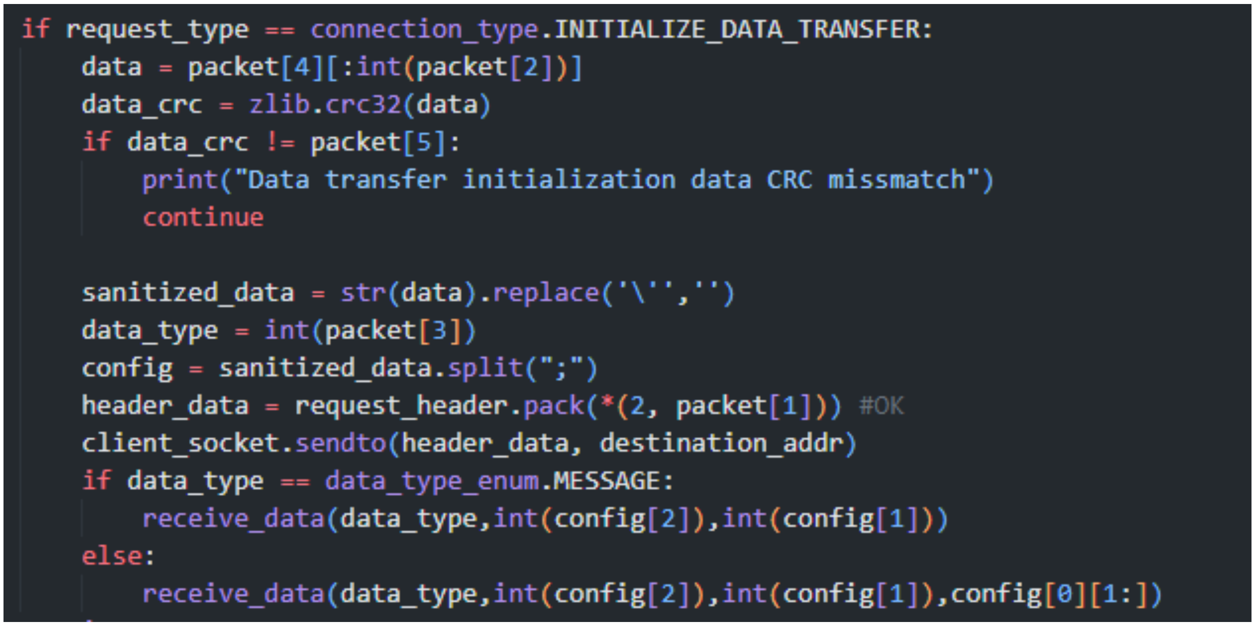
Server



```
header = struct.Struct(f'H I I H 200s I')
while not initialized:
    try:
        initialization_data = f"{path};{data_fragments};{data_fragment_size}".encode("utf-8")
        crc = zlib.crc32(initialization_data)
        header_data = header.pack(*(connection_type.INITIALIZE_DATA_TRANSFER, 0, len(initialization_data), data_type, initialization_data, crc)) #Initialization
        server_socket.sendto(header_data, client_address)
        print("Init sent")
        message, _ = server_socket.recvfrom(fragment_size)
        resp = request_header.unpack(message) #OK
        if resp[0] == connection_type.OK and resp[1] == 0:
            print("Data transfer initialized")
            initialized = True
    except TimeoutError:
        time.sleep(1)
    except ConnectionResetError:
            print("The connection has been reset! Quittiing...")
            time.sleep(2)
            quit()
```

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.

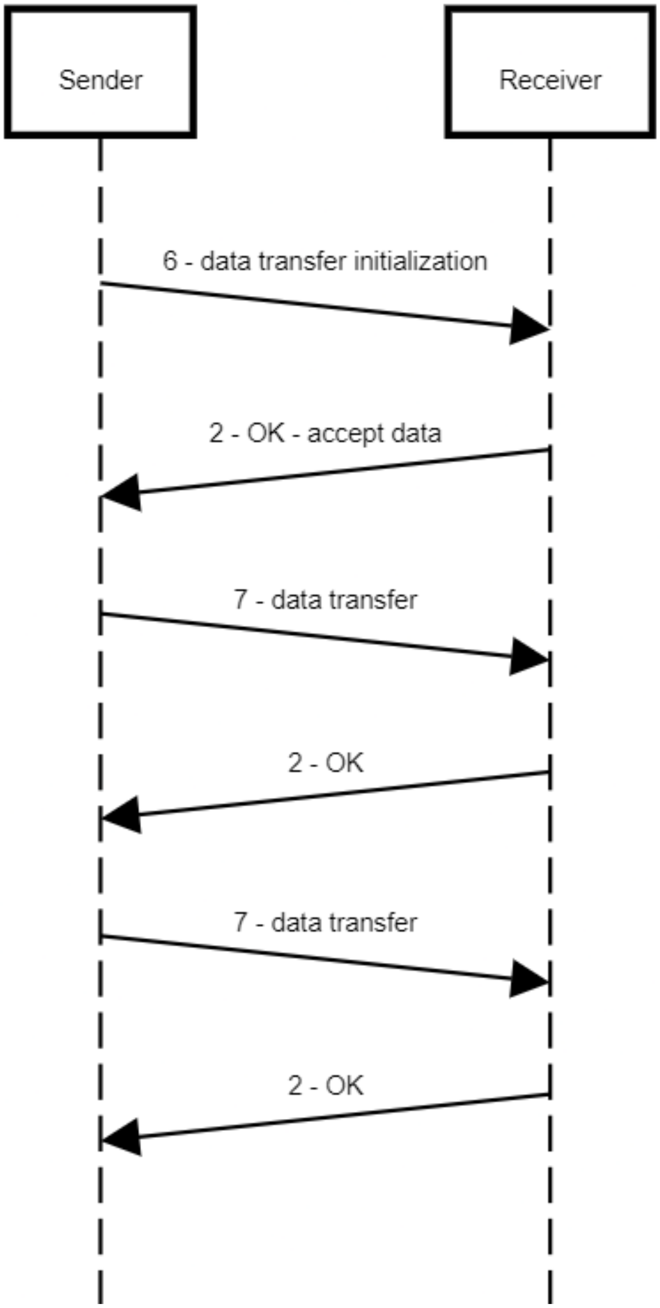
Client



```
if request_type == connection_type.INITIALIZE_DATA_TRANSFER:
    data = packet[4][:int(packet[2])]
    data_crc = zlib.crc32(data)
    if data_crc != packet[5]:
        print("Data transfer initialization data CRC missmatch")
        continue

    sanitized_data = str(data).replace('\'','')
    data_type = int(packet[3])
    config = sanitized_data.split(";")
    header_data = request_header.pack(*(2, packet[1])) #OK
    client_socket.sendto(header_data, destination_addr)
    if data_type == data_type_enum.MESSAGE:
        receive_data(data_type,int(config[2]),int(config[1]))
    else:
        receive_data(data_type,int(config[2]),int(config[1]),config[0][1:])
```

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.

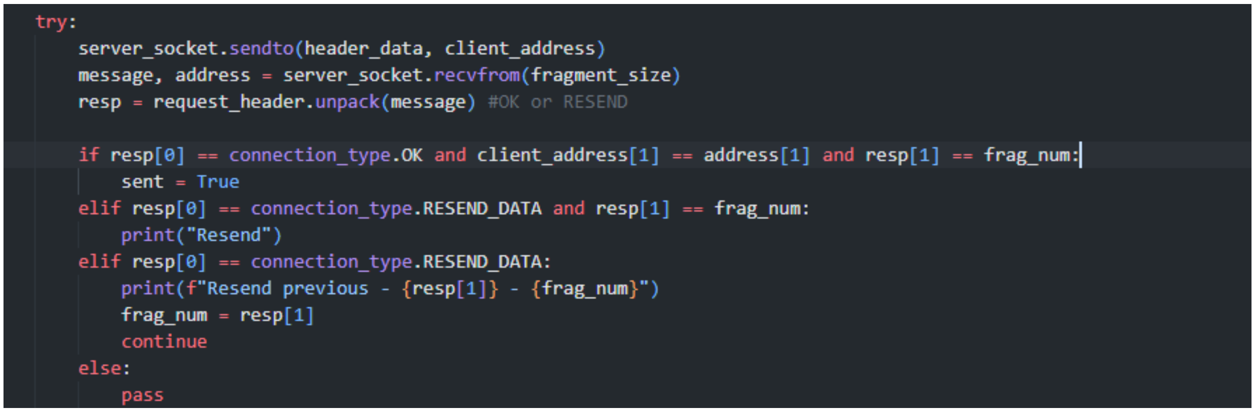
Data transfer



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



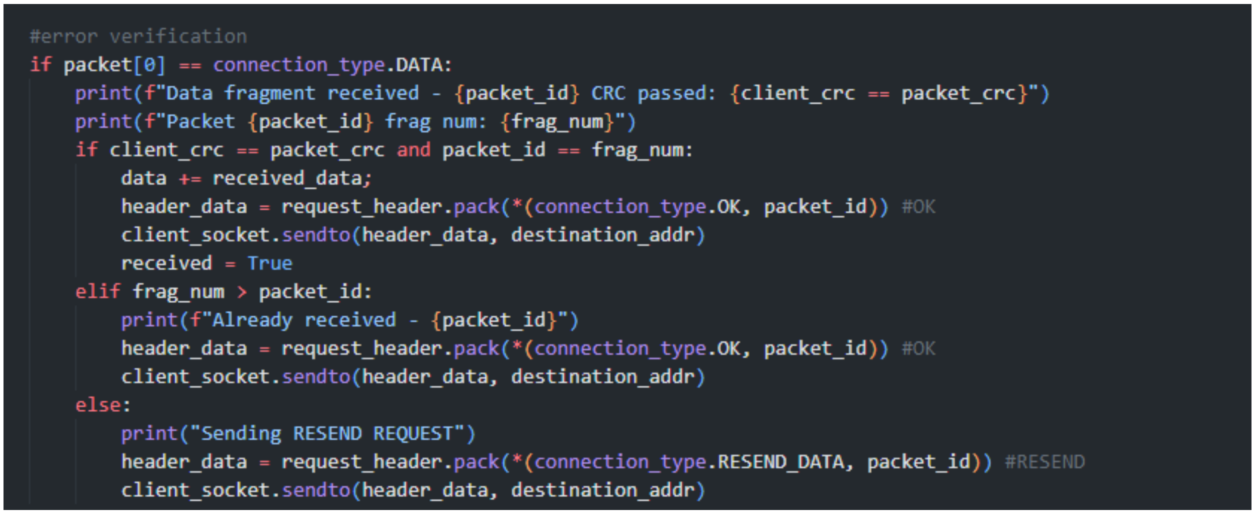
```
try:
    server_socket.sendto(header_data, client_address)
    message, address = server_socket.recvfrom(fragment_size)
    resp = request_header.unpack(message) #OK or RESEND

    if resp[0] == connection_type.OK and client_address[1] == address[1] and resp[1] == frag_num:
        sent = True
    elif resp[0] == connection_type.RESEND_DATA and resp[1] == frag_num:
        print("Resend")
    elif resp[0] == connection_type.RESEND_DATA:
        print(f"Resend previous - {resp[1]} - {frag_num}")
        frag_num = resp[1]
        continue
    else:
        pass
```

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.

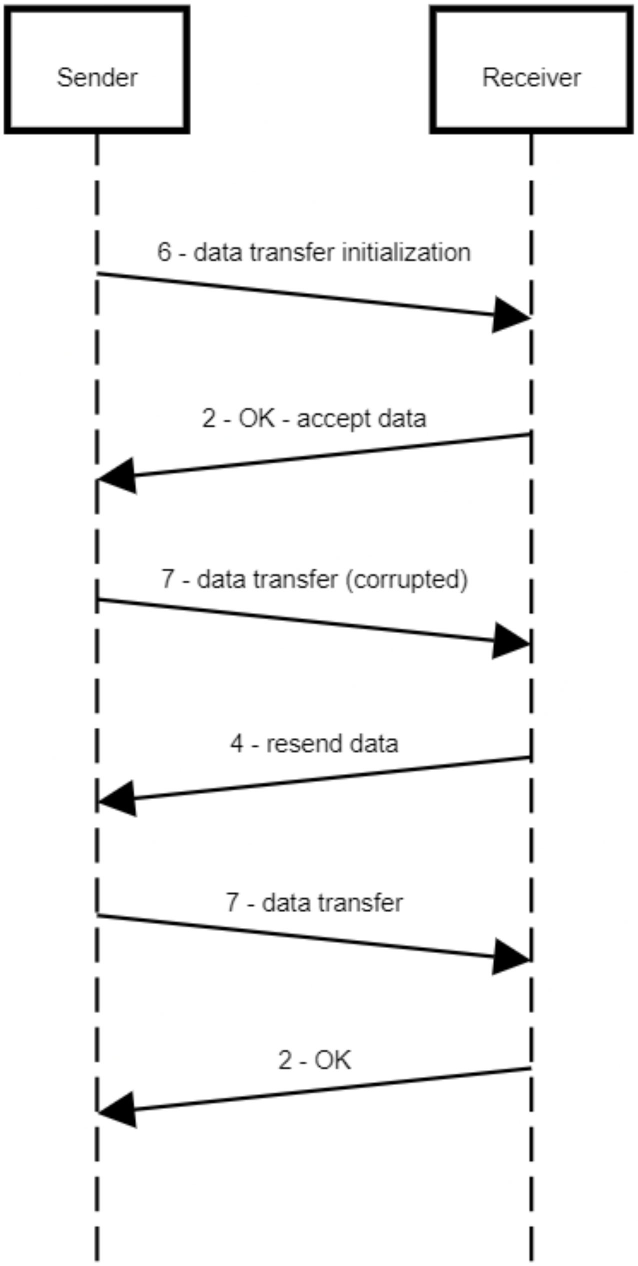
If the request is RESEND DATA with previous fragment id than it will go back to that fragment and send it again.

Client



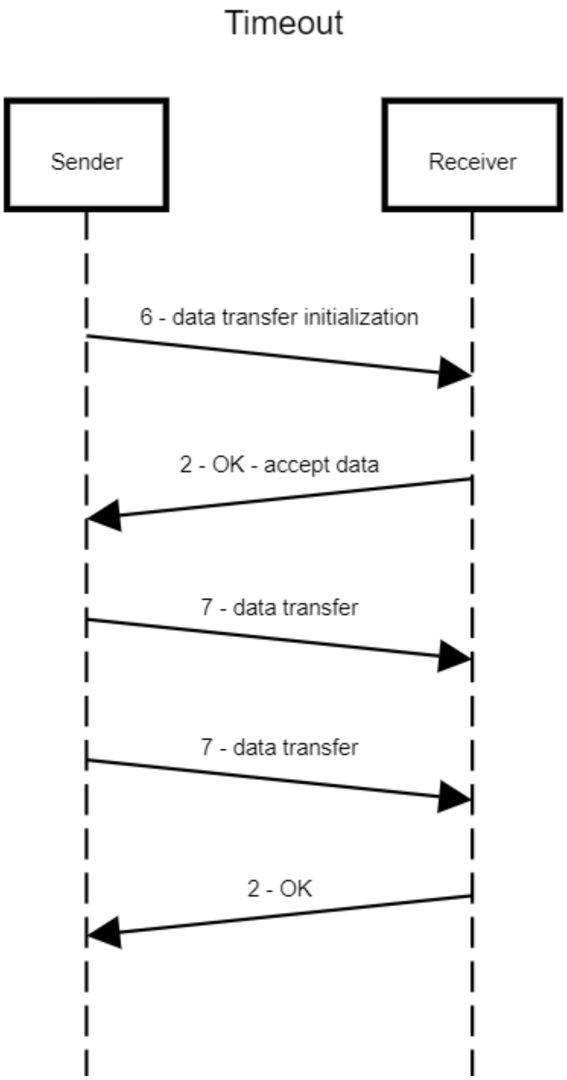
```
#error verification
if packet[0] == connection_type.DATA:
    print(f"Data fragment received - {packet_id} CRC passed: {client_crc == packet_crc}")
    print(f"Packet {packet_id} frag num: {frag_num}")
    if client_crc == packet_crc and packet_id == frag_num:
        data += received_data;
        header_data = request_header.pack(*(connection_type.OK, packet_id)) #OK
        client_socket.sendto(header_data, destination_addr)
        received = True
    elif frag_num > packet_id:
        print(f"Already received - {packet_id}")
        header_data = request_header.pack(*(connection_type.OK, packet_id)) #OK
        client_socket.sendto(header_data, destination_addr)
    else:
        print("Sending RESEND REQUEST")
        header_data = request_header.pack(*(connection_type.RESEND_DATA, packet_id)) #RESEND
        client_socket.sendto(header_data, destination_addr)
```

Corrupted data transfer



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.



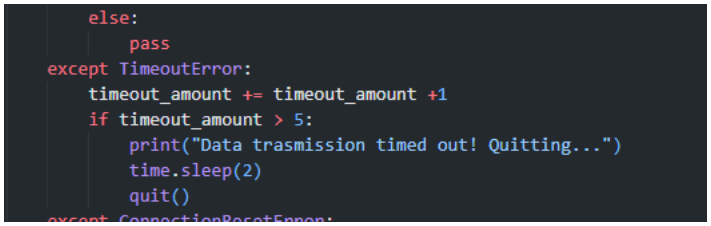
The receiver is not responding

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.

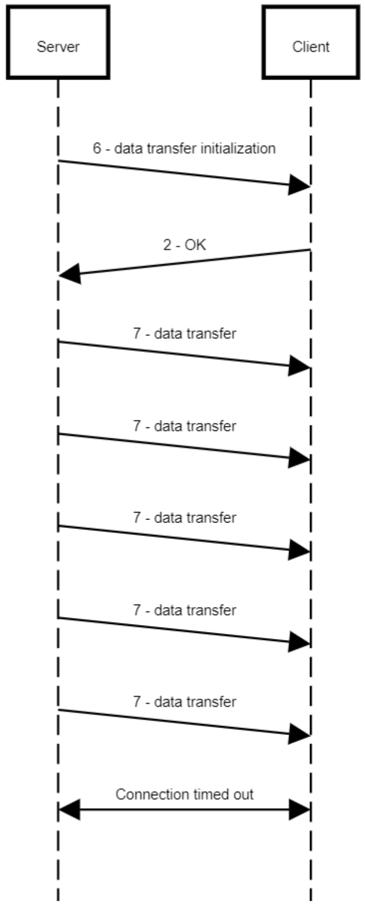
Server



```
        else:
            pass
    except TimeoutError:
        timeout_amount += timeout_amount +1
        if timeout_amount > 5:
            print("Data trasmission timed out! Quitting...")
            time.sleep(2)
            quit()
    except ConnectionResetError:
```

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

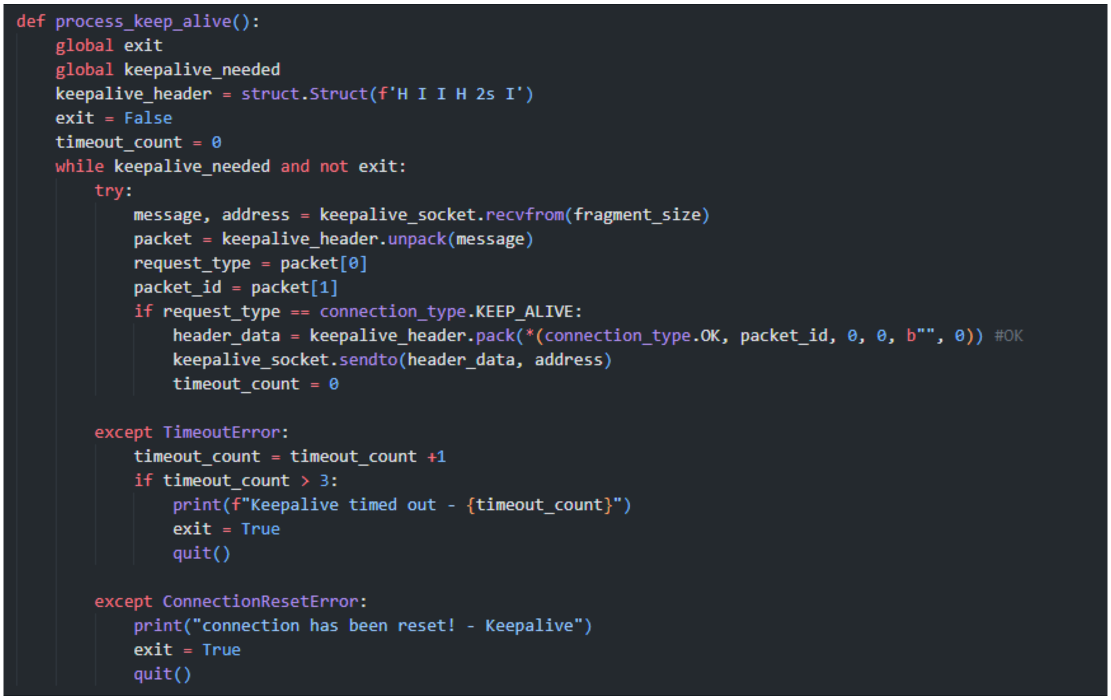
Not responding



Keepalive

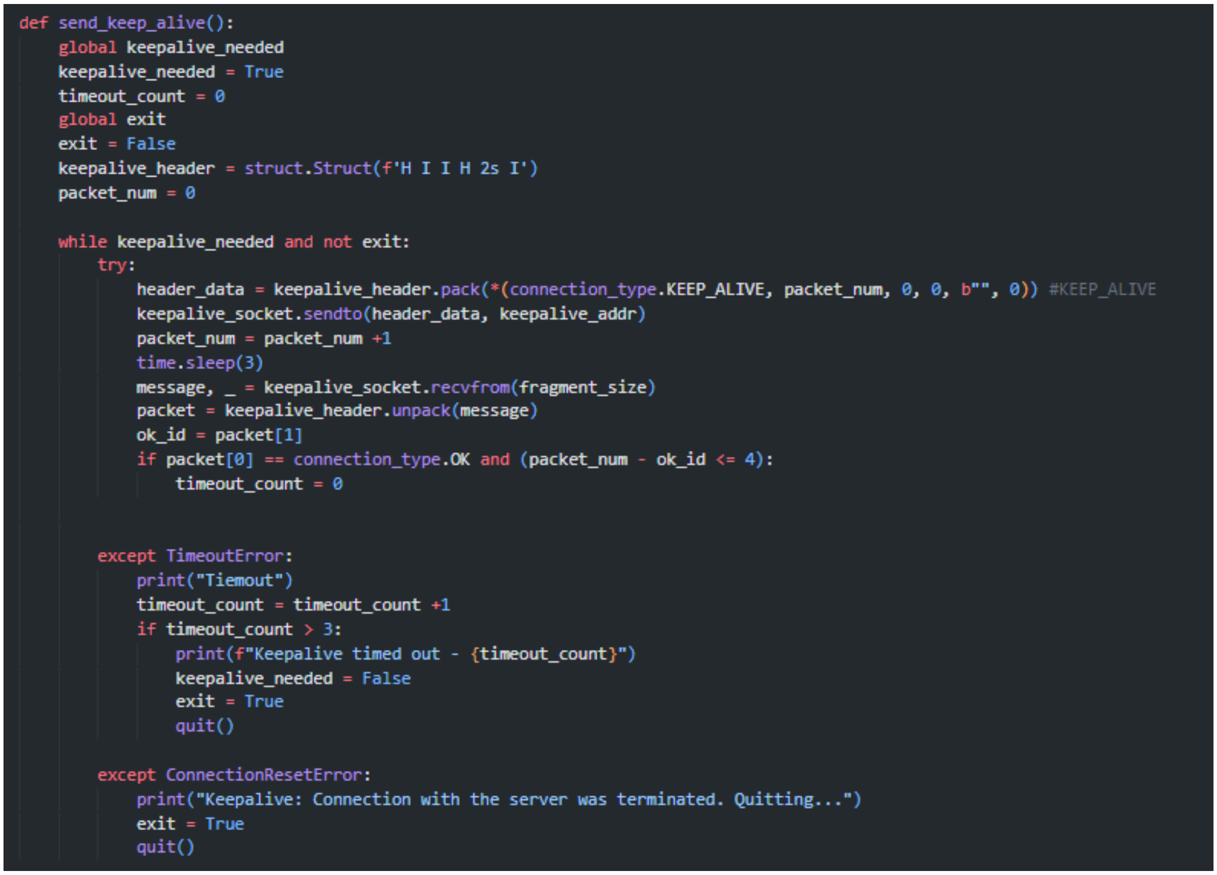
Keepalive is solved using a separate socket on each side. The client socket will send a keepalive request to the server and the server will reply to that request with an OK. If the keepalive request doesn't receive and OK request more than 3 times in a row the keepalive fails and the connection is terminated. 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



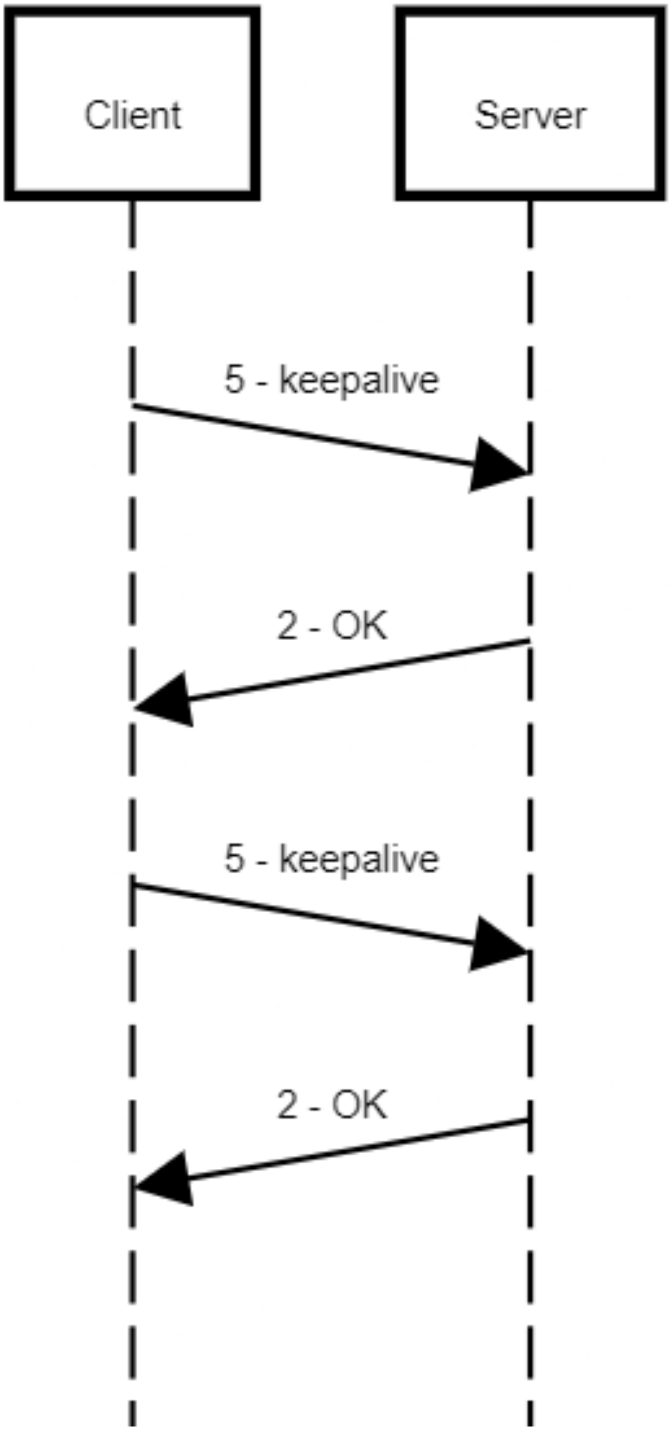
```python
def process_keep_alive():
    global exit
    global keepalive_needed
    keepalive_header = struct.Struct(f'H I I H 2s I')
    exit = False
    timeout_count = 0
    while keepalive_needed and not exit:
        try:
            message, address = keepalive_socket.recvfrom(fragment_size)
            packet = keepalive_header.unpack(message)
            request_type = packet[0]
            packet_id = packet[1]
            if request_type == connection_type.KEEP_ALIVE:
                header_data = keepalive_header.pack(*(connection_type.OK, packet_id, 0, 0, b"", 0)) #OK
                keepalive_socket.sendto(header_data, address)
                timeout_count = 0

        except TimeoutError:
            timeout_count = timeout_count +1
            if timeout_count > 3:
                print(f"Keepalive timed out - {timeout_count}")
                exit = True
                quit()

        except ConnectionResetError:
            print("connection has been reset! - Keepalive")
            exit = True
            quit()
```

Client



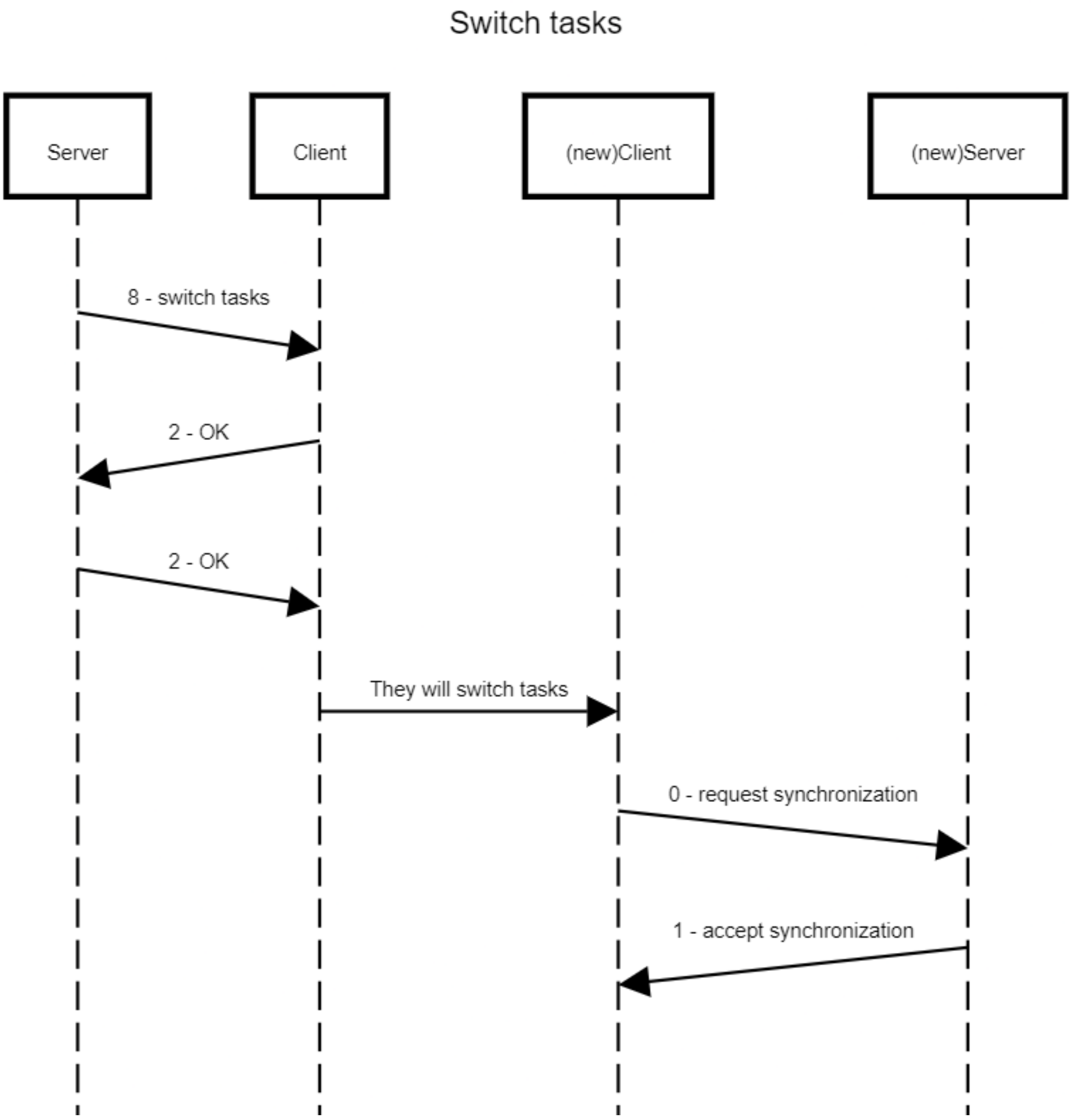
```python
def send_keep_alive():
    global keepalive_needed
    keepalive_needed = True
    timeout_count = 0
    global exit
    exit = False
    keepalive_header = struct.Struct(f'H I I H 2s I')
    packet_num = 0

    while keepalive_needed and not exit:
        try:
            header_data = keepalive_header.pack(*(connection_type.KEEP_ALIVE, packet_num, 0, 0, b"", 0)) #KEEP_ALIVE
            keepalive_socket.sendto(header_data, keepalive_addr)
            packet_num = packet_num +1
            time.sleep(3)
            message, _ = keepalive_socket.recvfrom(fragment_size)
            packet = keepalive_header.unpack(message)
            ok_id = packet[1]
            if packet[0] == connection_type.OK and (packet_num - ok_id <= 4):
                timeout_count = 0


        except TimeoutError:
            print("Tiemout")
            timeout_count = timeout_count +1
            if timeout_count > 3:
                print(f"Keepalive timed out - {timeout_count}")
                keepalive_needed = False
                exit = True
                quit()

        except ConnectionResetError:
            print("Keepalive: Connection with the server was terminated. Quitting...")
            exit = True
            quit()
```

Keepalive



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.

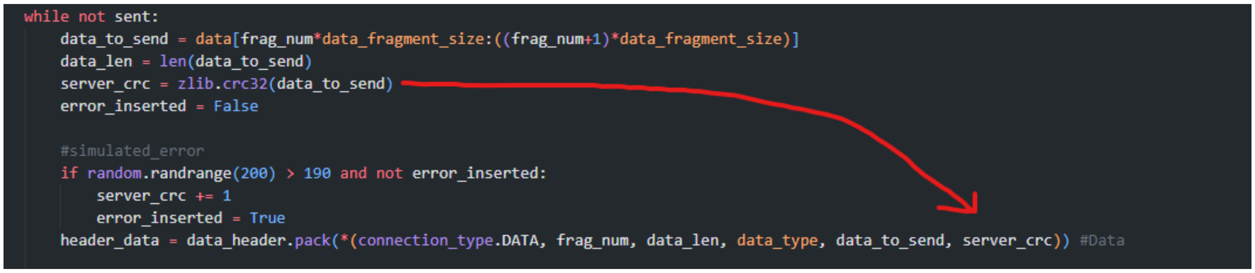


CRC method

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.

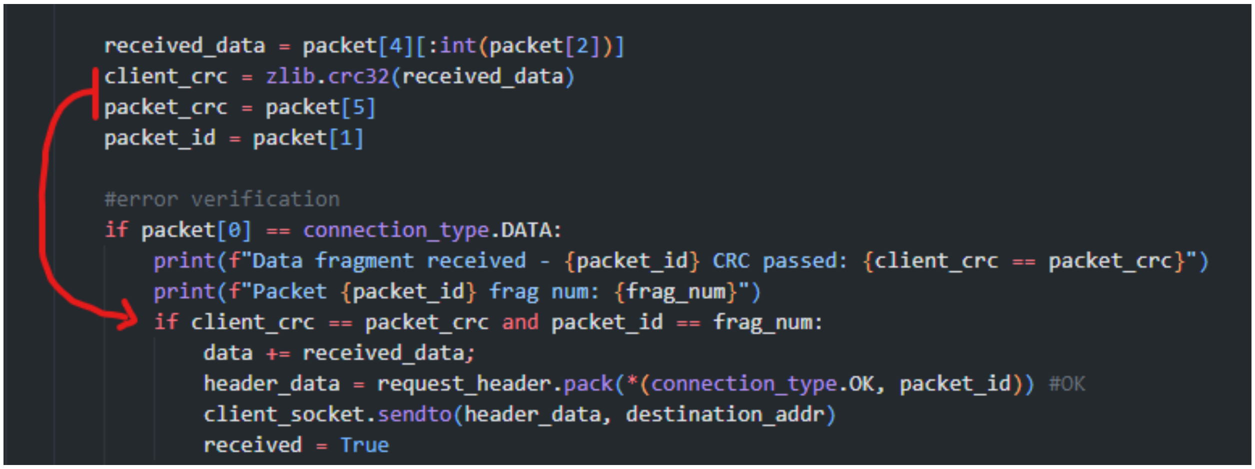
CRC is used when an important data/fragment is received a CRC checksum is generated on the server side which is included in the request header. Then the client will also generate a CRC checksum for the received data and these checksums will be compared.

Server



```python
while not sent:
    data_to_send = data[frag_num*data_fragment_size:((frag_num+1)*data_fragment_size)]
    data_len = len(data_to_send)
    server_crc = zlib.crc32(data_to_send)
    error_inserted = False

    #simulated_error
    if random.randrange(200) > 190 and not error_inserted:
        server_crc += 1
        error_inserted = True
    header_data = data_header.pack(*(connection_type.DATA, frag_num, data_len, data_type, data_to_send, server_crc)) #Data
```

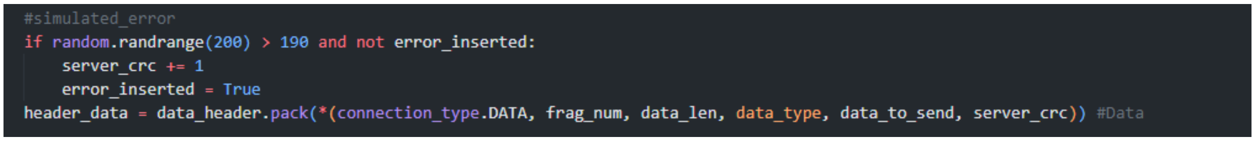
Client



```python
received_data = packet[4][:int(packet[2])]
client_crc = zlib.crc32(received_data)
packet_crc = packet[5]
packet_id = packet[1]

#error verification
if packet[0] == connection_type.DATA:
    print(f"Data fragment received - {packet_id} CRC passed: {client_crc == packet_crc}")
    print(f"Packet {packet_id} frag num: {frag_num}")
    if client_crc == packet_crc and packet_id == frag_num:
        data += received_data;
        header_data = request_header.pack(*(connection_type.OK, packet_id)) #OK
        client_socket.sendto(header_data, destination_addr)
        received = True
```

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.



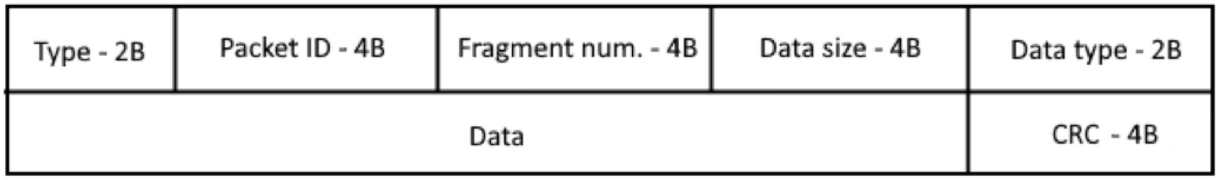
```python
#simulated_error
if random.randrange(200) > 190 and not error_inserted:
    server_crc += 1
    error_inserted = True
header_data = data_header.pack(*(connection_type.DATA, frag_num, data_len, data_type, data_to_send, server_crc)) #Data
```

Changes from the original design

Header design

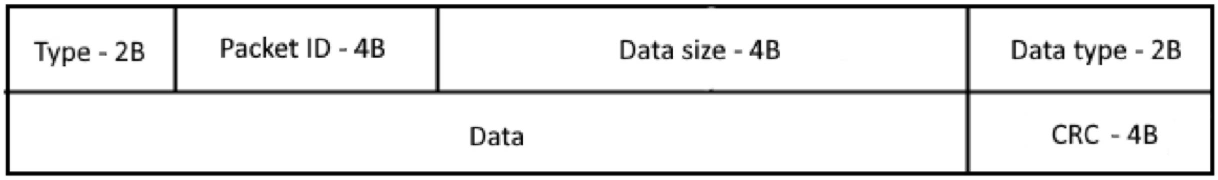
The header has been simplified as there is no need for separate fragment num and packet id. Instead, a common packet id is used.

OLD



Type - 2B	Packet ID - 4B	Fragment num. - 4B	Data size - 4B	Data type - 2B
Data				CRC - 4B

NEW

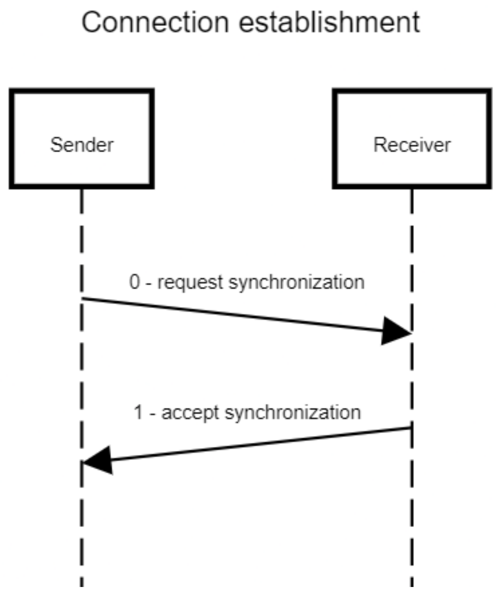


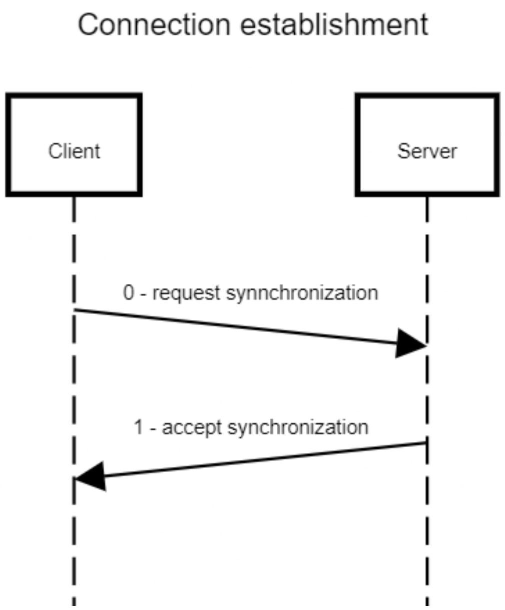
Type - 2B	Packet ID - 4B	Data size - 4B	Data type - 2B
Data			CRC - 4B

Connection establishment

Now the client will initialize the connection

OLD **NEW**

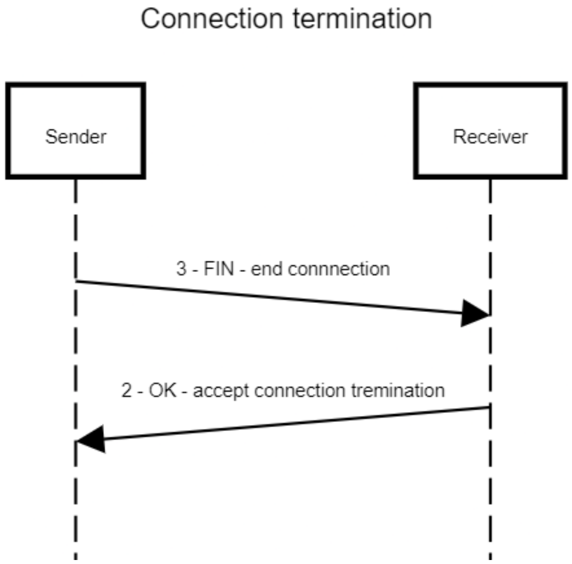


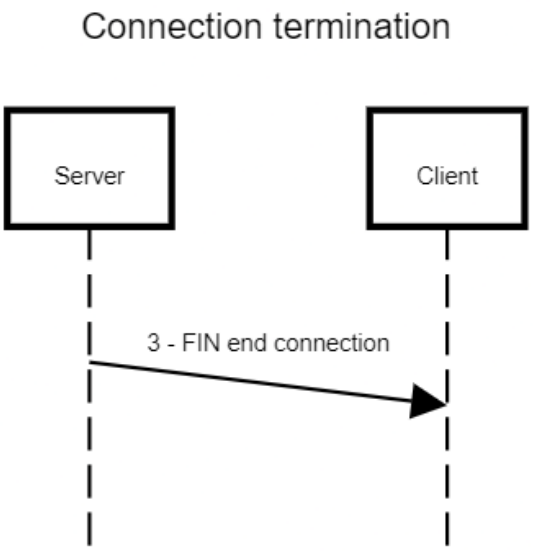


Connection termination

No need to accept connection termination.

OLD **NEW**

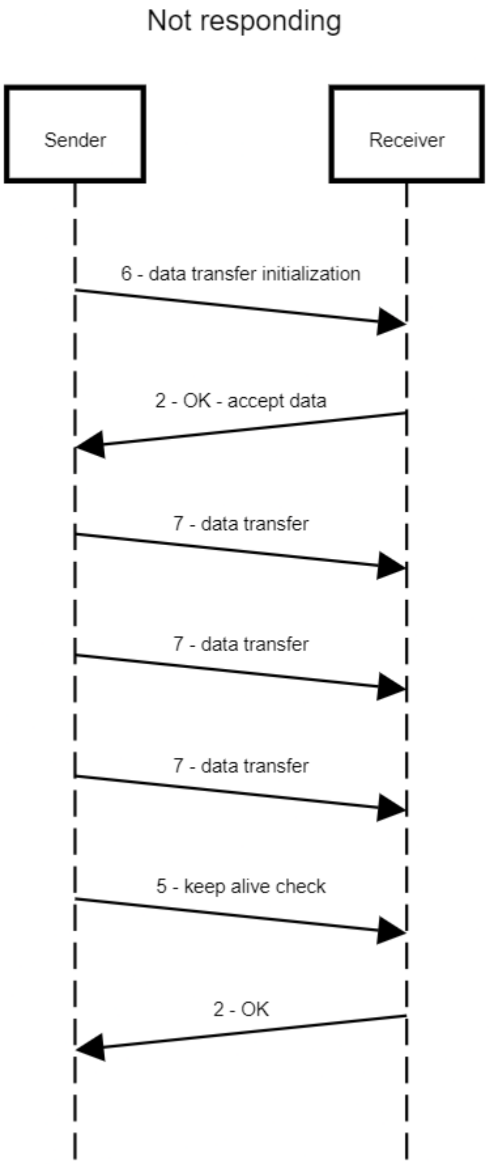

Connection termination
Sender
Receiver
3 - FIN - end connnection
2 - OK - accept connection tremination



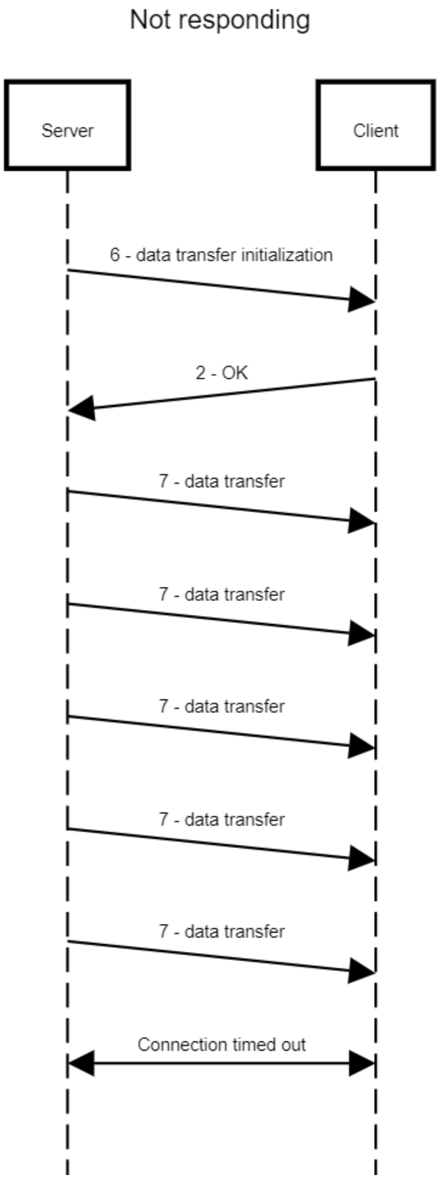
Data transfer timeout

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

OLD



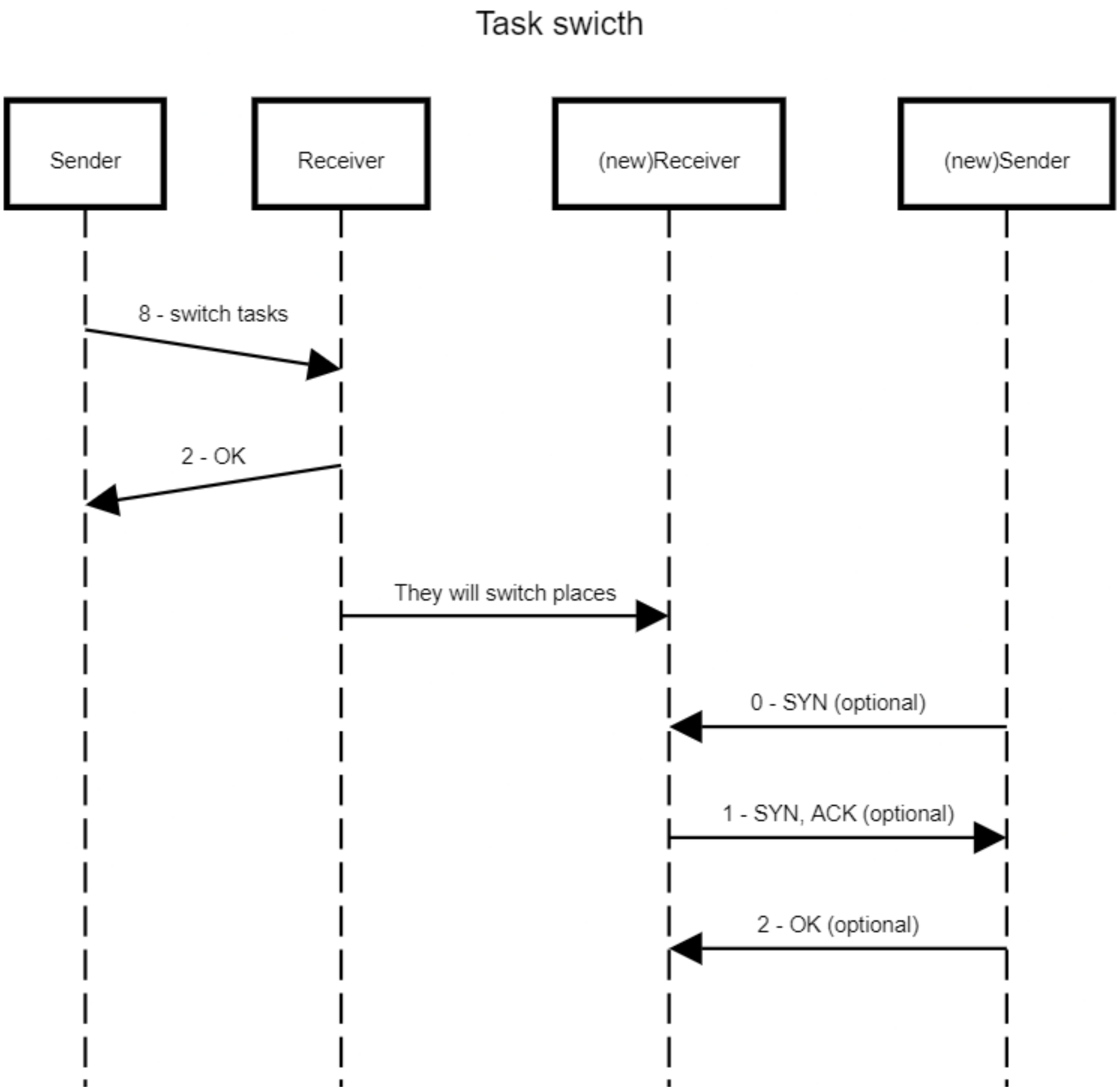
NEW



Switch task

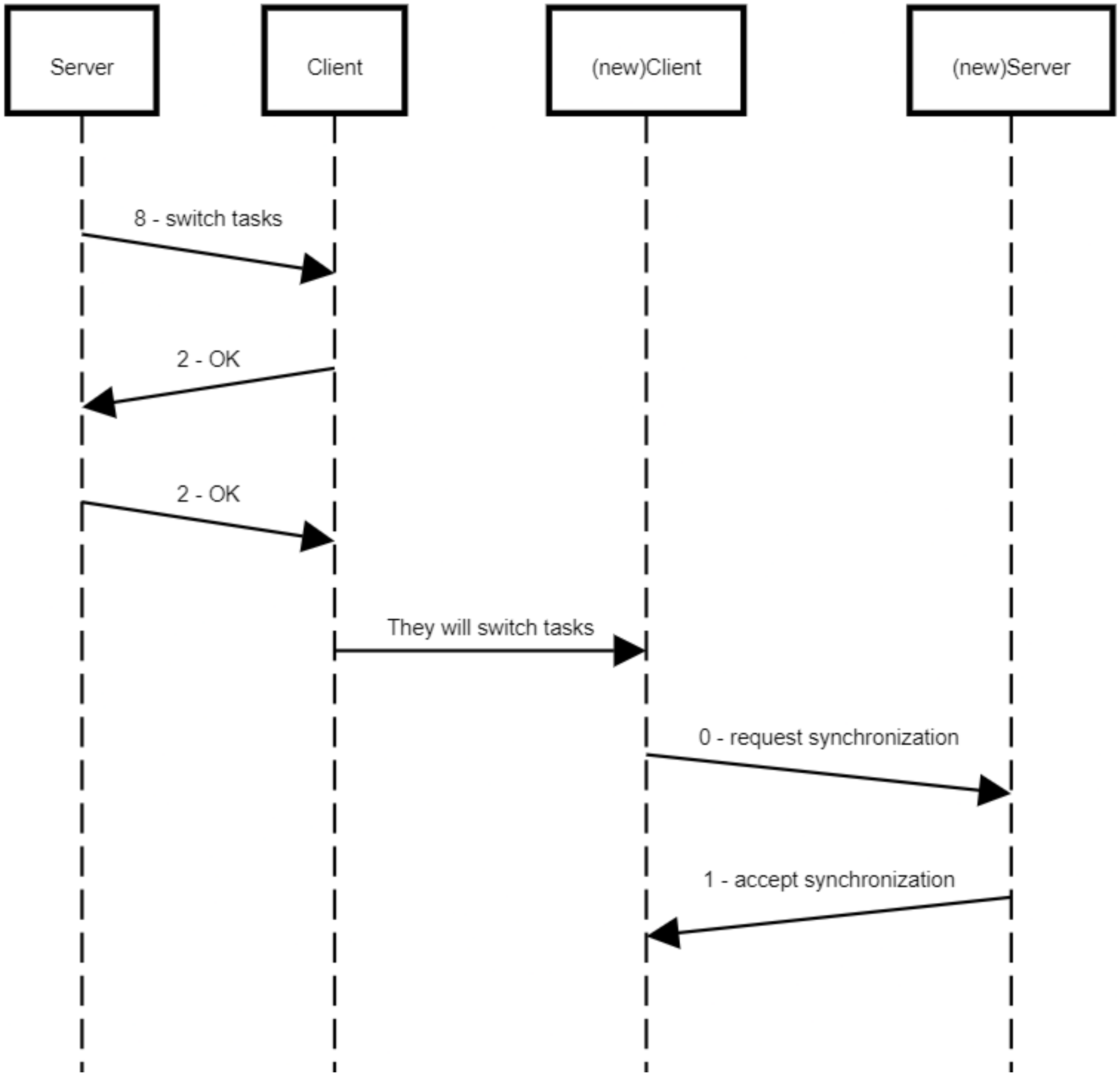
Re-synchronization after switch task won't be optional but required. This will be done automatically. Another change that the server also has to confirm that the client got the task switch request. The synchronization process is also simpler.

OLD

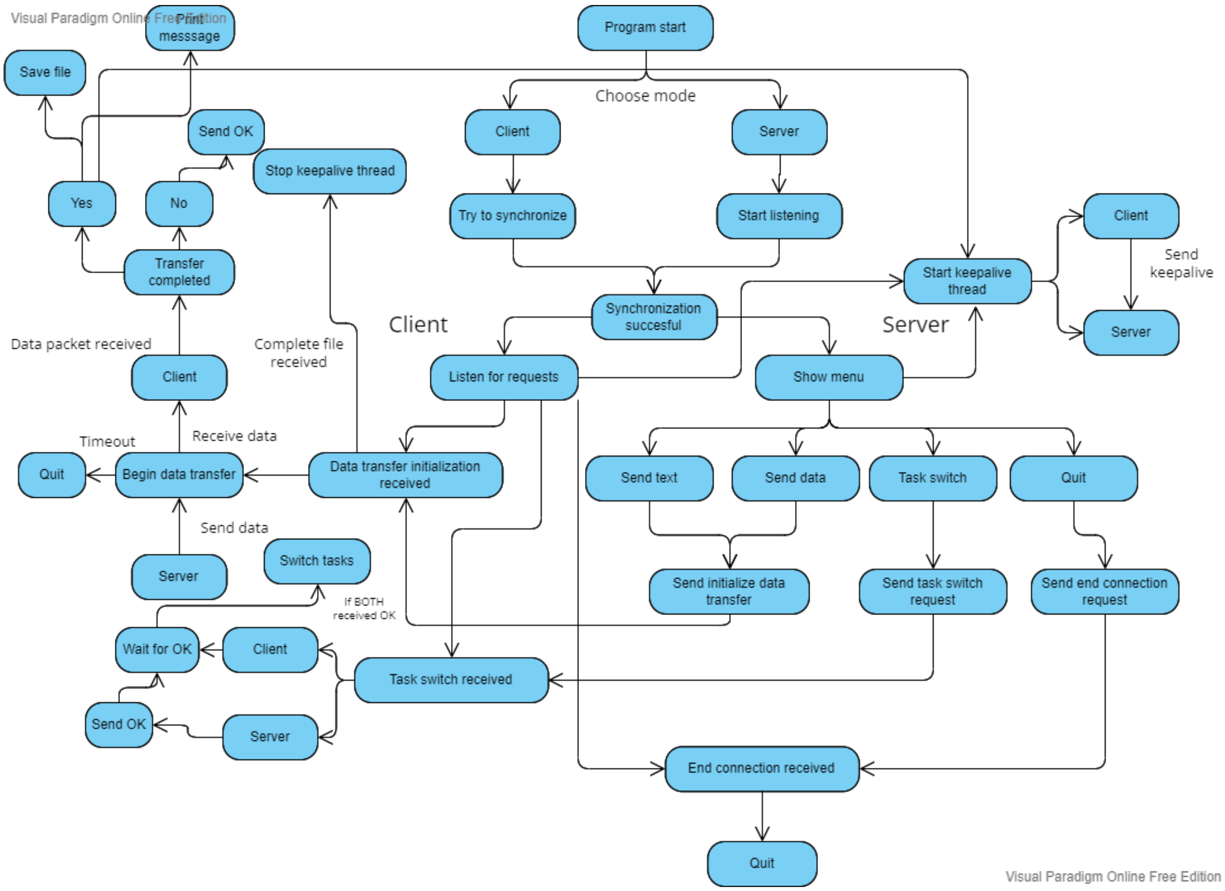


NEW

Switch tasks

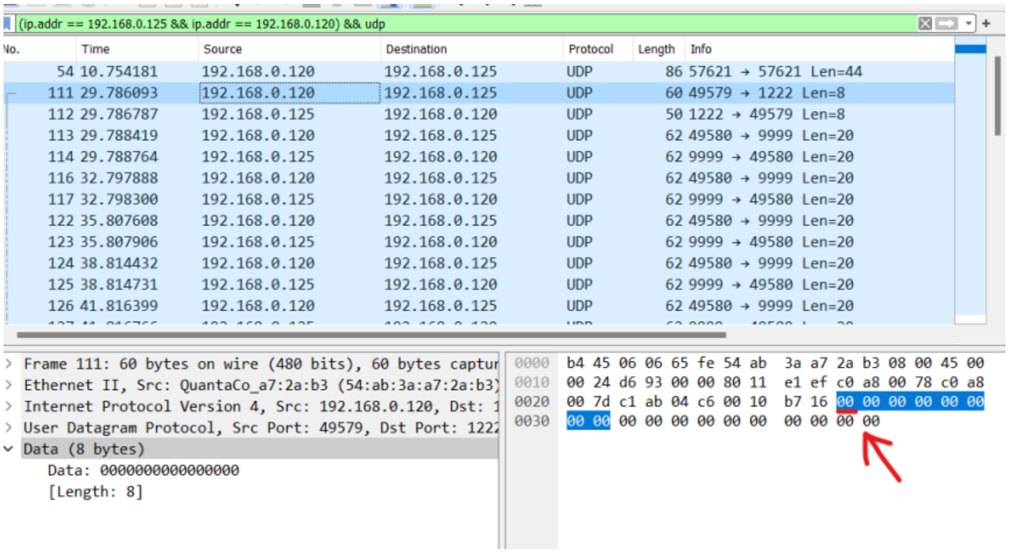


Activity chart



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

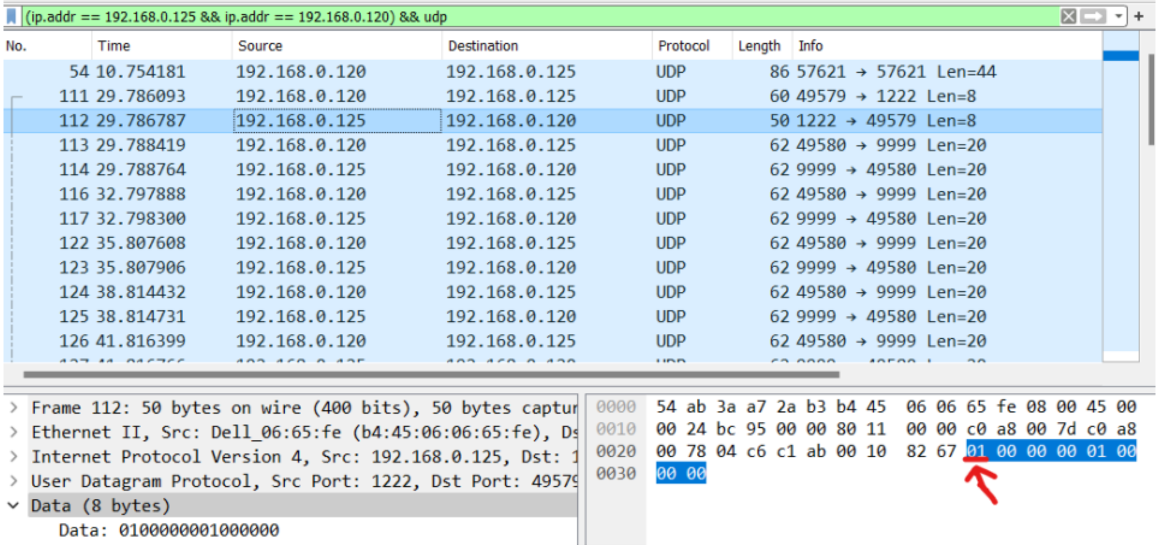


```
(ip.addr == 192.168.0.125 && ip.addr == 192.168.0.120) && udp

No.   Time         Source          Destination     Protocol  Length  Info
   54 10.754181    192.168.0.120   192.168.0.125   UDP           86  57621 → 57621 Len=44
  111 29.786093    192.168.0.120   192.168.0.125   UDP           60  49579 → 1222 Len=8
  112 29.786787    192.168.0.125   192.168.0.120   UDP           50  1222 → 49579 Len=8
  113 29.788419    192.168.0.120   192.168.0.125   UDP           62  49580 → 9999 Len=20
  114 29.788764    192.168.0.125   192.168.0.120   UDP           62  9999 → 49580 Len=20
  116 32.797888    192.168.0.120   192.168.0.125   UDP           62  49580 → 9999 Len=20
  117 32.798300    192.168.0.125   192.168.0.120   UDP           62  9999 → 49580 Len=20
  122 35.807608    192.168.0.120   192.168.0.125   UDP           62  49580 → 9999 Len=20
  123 35.807906    192.168.0.125   192.168.0.120   UDP           62  9999 → 49580 Len=20
  124 38.814432    192.168.0.120   192.168.0.125   UDP           62  49580 → 9999 Len=20
  125 38.814731    192.168.0.125   192.168.0.120   UDP           62  9999 → 49580 Len=20
  126 41.816399    192.168.0.120   192.168.0.125   UDP           62  49580 → 9999 Len=20
```

```
> Frame 111: 60 bytes on wire (480 bits), 60 bytes captur
> Ethernet II, Src: QuantaCo_a7:2a:b3 (54:ab:3a:a7:2a:b3)
> Internet Protocol Version 4, Src: 192.168.0.120, Dst: 1
> User Datagram Protocol, Src Port: 49579, Dst Port: 1222
v Data (8 bytes)
    Data: 0000000000000000
    [Length: 8]
```

```
0000  b4 45 06 06 65 fe 54 ab  3a a7 2a b3 08 00 45 00
0010  00 24 d6 93 00 00 80 11  e1 ef c0 a8 00 78 c0 a8
0020  00 7d c1 ab 04 c6 00 10  b7 16 00 00 00 00 00 00
0030  00 00 00 00 00 00 00 00  00 00 00 00
```

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

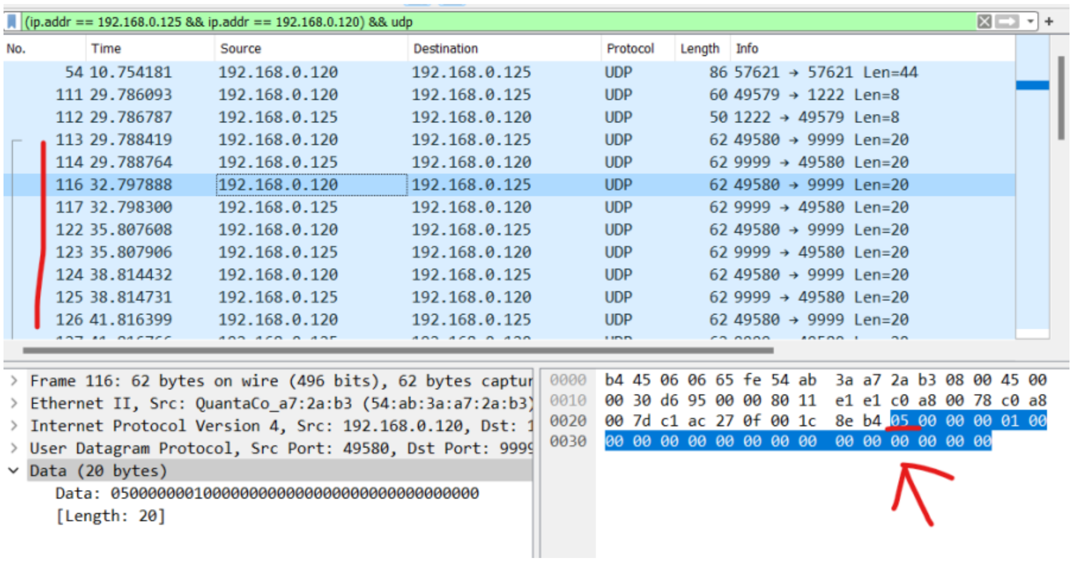


```
(ip.addr == 192.168.0.125 && ip.addr == 192.168.0.120) && udp

No.   Time         Source          Destination     Protocol  Length  Info
   54 10.754181    192.168.0.120   192.168.0.125   UDP           86  57621 → 57621 Len=44
  111 29.786093    192.168.0.120   192.168.0.125   UDP           60  49579 → 1222 Len=8
  112 29.786787    192.168.0.125   192.168.0.120   UDP           50  1222 → 49579 Len=8
  113 29.788419    192.168.0.120   192.168.0.125   UDP           62  49580 → 9999 Len=20
  114 29.788764    192.168.0.125   192.168.0.120   UDP           62  9999 → 49580 Len=20
  116 32.797888    192.168.0.120   192.168.0.125   UDP           62  49580 → 9999 Len=20
  117 32.798300    192.168.0.125   192.168.0.120   UDP           62  9999 → 49580 Len=20
  122 35.807608    192.168.0.120   192.168.0.125   UDP           62  49580 → 9999 Len=20
  123 35.807906    192.168.0.125   192.168.0.120   UDP           62  9999 → 49580 Len=20
  124 38.814432    192.168.0.120   192.168.0.125   UDP           62  49580 → 9999 Len=20
  125 38.814731    192.168.0.125   192.168.0.120   UDP           62  9999 → 49580 Len=20
  126 41.816399    192.168.0.120   192.168.0.125   UDP           62  49580 → 9999 Len=20
```

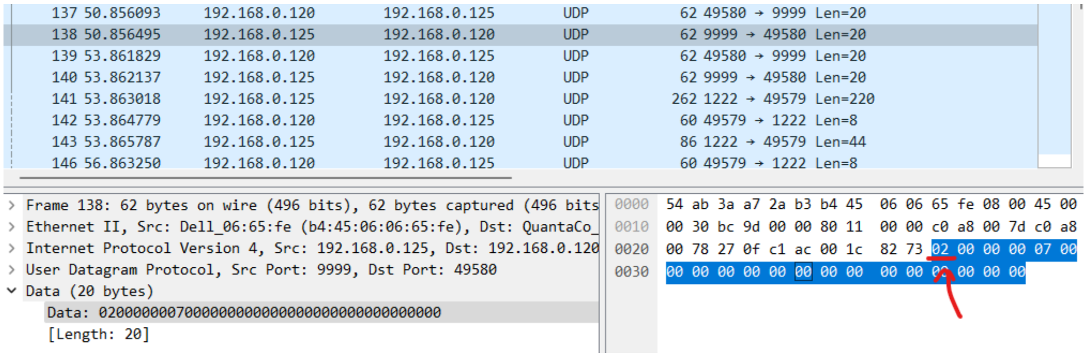
```
> Frame 112: 50 bytes on wire (400 bits), 50 bytes captur
> Ethernet II, Src: Dell_06:65:fe (b4:45:06:06:65:fe), Ds
> Internet Protocol Version 4, Src: 192.168.0.125, Dst: 1
> User Datagram Protocol, Src Port: 1222, Dst Port: 4957
v Data (8 bytes)
    Data: 0100000001000000
```

```
0000  54 ab 3a a7 2a b3 b4 45  06 06 65 fe 08 00 45 00
0010  00 24 bc 95 00 00 80 11  00 00 c0 a8 00 7d c0 a8
0020  00 78 04 c6 c1 ab 00 10  82 67 01 00 00 00 01 00
0030  00 00
```

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

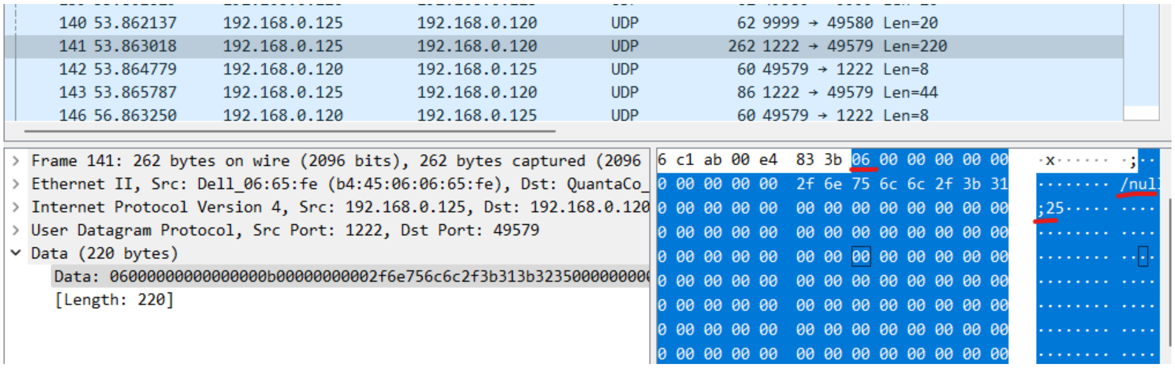


```
(ip.addr == 192.168.0.125 && ip.addr == 192.168.0.120) && udp

No.   Time        Source          Destination     Protocol  Length Info
  54 10.754181   192.168.0.120   192.168.0.125   UDP       86 57621 → 57621 Len=44
 111 29.786093   192.168.0.120   192.168.0.125   UDP       60 49579 → 1222 Len=8
 112 29.786787   192.168.0.125   192.168.0.120   UDP       50 1222 → 49579 Len=8
 113 29.788419   192.168.0.120   192.168.0.125   UDP       62 49580 → 9999 Len=20
 114 29.788764   192.168.0.125   192.168.0.120   UDP       62 9999 → 49580 Len=20
 116 32.797888   192.168.0.120   192.168.0.125   UDP       62 49580 → 9999 Len=20
 117 32.798300   192.168.0.125   192.168.0.120   UDP       62 9999 → 49580 Len=20
 122 35.807608   192.168.0.120   192.168.0.125   UDP       62 49580 → 9999 Len=20
 123 35.807906   192.168.0.125   192.168.0.120   UDP       62 9999 → 49580 Len=20
 124 38.814432   192.168.0.120   192.168.0.125   UDP       62 49580 → 9999 Len=20
 125 38.814731   192.168.0.125   192.168.0.120   UDP       62 9999 → 49580 Len=20
 126 41.816399   192.168.0.120   192.168.0.125   UDP       62 49580 → 9999 Len=20

> Frame 116: 62 bytes on wire (496 bits), 62 bytes captur   0000  b4 45 06 06 65 fe 54 ab  3a a7 2a b3 08 00 45 00
> Ethernet II, Src: QuantaCo_a7:2a:b3 (54:ab:3a:a7:2a:b3)   0010  00 30 d6 95 00 00 80 11  e1 e1 c0 a8 00 78 c0 a8
> Internet Protocol Version 4, Src: 192.168.0.120, Dst: 1   0020  00 7d c1 ac 27 0f 00 1c  8e b4 05 00 00 00 01 00
> User Datagram Protocol, Src Port: 49580, Dst Port: 9999   0030  00 00 00 00 00 00 00 00  00 00 00 00 00 00
v Data (20 bytes)
    Data: 0500000001000000000000000000000000000000
    [Length: 20]
```



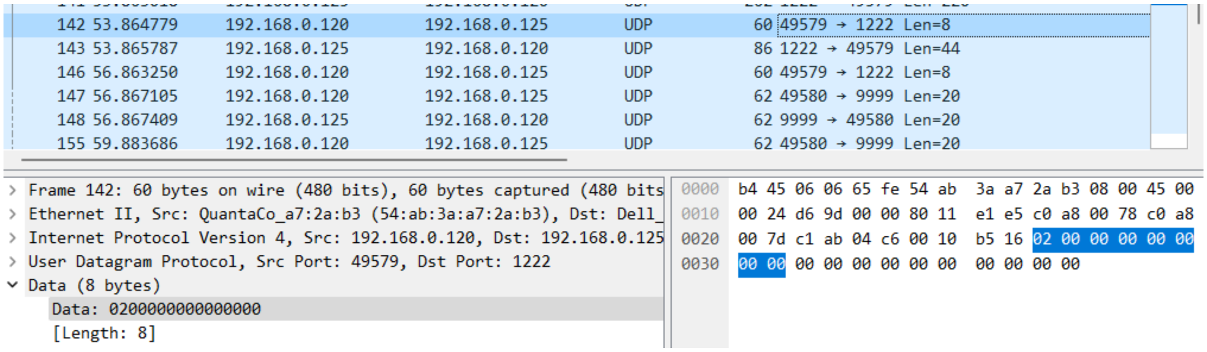
```
 137 50.856093   192.168.0.120   192.168.0.125   UDP       62 49580 → 9999 Len=20
 138 50.856495   192.168.0.125   192.168.0.120   UDP       62 9999 → 49580 Len=20
 139 53.861829   192.168.0.120   192.168.0.125   UDP       62 49580 → 9999 Len=20
 140 53.862137   192.168.0.125   192.168.0.120   UDP       62 9999 → 49580 Len=20
 141 53.863018   192.168.0.125   192.168.0.120   UDP      262 1222 → 49579 Len=220
 142 53.864779   192.168.0.120   192.168.0.125   UDP       60 49579 → 1222 Len=8
 143 53.865787   192.168.0.125   192.168.0.120   UDP       86 1222 → 49579 Len=44
 146 56.863250   192.168.0.120   192.168.0.125   UDP       60 49579 → 1222 Len=8

> Frame 138: 62 bytes on wire (496 bits), 62 bytes captured (496 bits   0000  54 ab 3a a7 2a b3 b4 45  06 06 65 fe 08 00 45 00
> Ethernet II, Src: Dell_06:65:fe (b4:45:06:06:65:fe), Dst: QuantaCo_   0010  00 30 bc 9d 00 00 80 11  00 00 c0 a8 00 7d c0 a8
> Internet Protocol Version 4, Src: 192.168.0.125, Dst: 192.168.0.120   0020  00 78 27 0f c1 ac 00 1c  82 73 02 00 00 00 07 00
> User Datagram Protocol, Src Port: 9999, Dst Port: 49580                0030  00 00 00 00 00 00 00 00  00 00 00 00 00 00
v Data (20 bytes)
    Data: 0200000007000000000000000000000000000000
    [Length: 20]
```

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



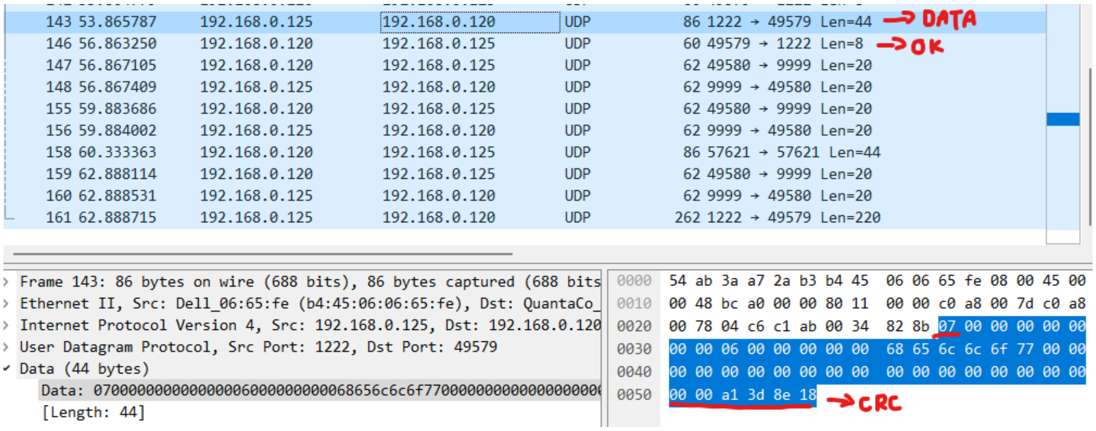
```
 140 53.862137   192.168.0.125   192.168.0.120   UDP       62 9999 → 49580 Len=20
 141 53.863018   192.168.0.125   192.168.0.120   UDP      262 1222 → 49579 Len=220
 142 53.864779   192.168.0.120   192.168.0.125   UDP       60 49579 → 1222 Len=8
 143 53.865787   192.168.0.125   192.168.0.120   UDP       86 1222 → 49579 Len=44
 146 56.863250   192.168.0.120   192.168.0.125   UDP       60 49579 → 1222 Len=8

> Frame 141: 262 bytes on wire (2096 bits), 262 bytes captured (2096   6 c1 ab 00 e4  83 3b 06 00 00 00 00 00   ·x······ ·;··
> Ethernet II, Src: Dell_06:65:fe (b4:45:06:06:65:fe), Dst: QuantaCo_  0 00 00 00 00  2f 6e 75 6c 6c 2f 3b 31   ········ /nul
> Internet Protocol Version 4, Src: 192.168.0.125, Dst: 192.168.0.120  0 00 00 00 00  00 00 00 00 00 00 00 00   ;25····· ····
> User Datagram Protocol, Src Port: 1222, Dst Port: 49579              0 00 00 00 00  00 00 00 00 00 00 00 00   ········ ····
v Data (220 bytes)                                                     0 00 00 00 00  00 00 00 00 00 00 00 00   ········ ····
    Data: 060000000000000000b00000000002f6e756c6c2f3b313b3235000000000   0 00 00 00 00  00 00 00 00 00 00 00 00   ········ ····
    [Length: 220]                                                      0 00 00 00 00  00 00 00 00 00 00 00 00   ········ ····
                                                                       0 00 00 00 00  00 00 00 00 00 00 00 00   ········ ····
                                                                       0 00 00 00 00  00 00 00 00 00 00 00 00   ········ ····
```

5. OK from the client.



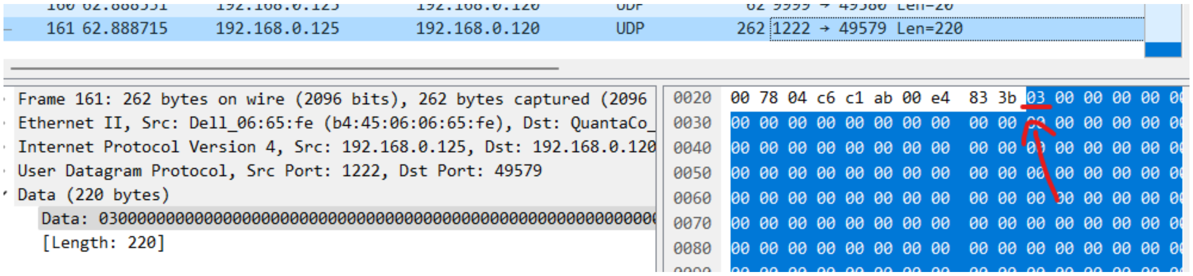
```
142 53.864779   192.168.0.120   192.168.0.125   UDP    60 49579 → 1222 Len=8
143 53.865787   192.168.0.125   192.168.0.120   UDP    86 1222 → 49579 Len=44
146 56.863250   192.168.0.120   192.168.0.125   UDP    60 49579 → 1222 Len=8
147 56.867105   192.168.0.120   192.168.0.125   UDP    62 49580 → 9999 Len=20
148 56.867409   192.168.0.125   192.168.0.120   UDP    62 9999 → 49580 Len=20
155 59.883686   192.168.0.120   192.168.0.125   UDP    62 49580 → 9999 Len=20

> Frame 142: 60 bytes on wire (480 bits), 60 bytes captured (480 bits
> Ethernet II, Src: QuantaCo_a7:2a:b3 (54:ab:3a:a7:2a:b3), Dst: Dell_
> Internet Protocol Version 4, Src: 192.168.0.120, Dst: 192.168.0.125
> User Datagram Protocol, Src Port: 49579, Dst Port: 1222
v Data (8 bytes)
    Data: 0200000000000000
    [Length: 8]

0000  b4 45 06 06 65 fe 54 ab  3a a7 2a b3 08 00 45 00
0010  00 24 d6 9d 00 00 80 11  e1 e5 c0 a8 00 78 c0 a8
0020  00 7d c1 ab 04 c6 00 10  b5 16 02 00 00 00 00 00
0030  00 00 00 00 00 00 00 00  00 00 00 00
```

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



```
143 53.865787   192.168.0.125   192.168.0.120   UDP    86 1222 → 49579 Len=44
146 56.863250   192.168.0.120   192.168.0.125   UDP    60 49579 → 1222 Len=8
147 56.867105   192.168.0.120   192.168.0.125   UDP    62 49580 → 9999 Len=20
148 56.867409   192.168.0.125   192.168.0.120   UDP    62 9999 → 49580 Len=20
155 59.883686   192.168.0.120   192.168.0.125   UDP    62 49580 → 9999 Len=20
156 59.884002   192.168.0.125   192.168.0.120   UDP    62 9999 → 49580 Len=20
158 60.333363   192.168.0.120   192.168.0.125   UDP    86 57621 → 57621 Len=44
159 62.888114   192.168.0.120   192.168.0.125   UDP    62 49580 → 9999 Len=20
160 62.888531   192.168.0.125   192.168.0.120   UDP    62 9999 → 49580 Len=20
161 62.888715   192.168.0.125   192.168.0.120   UDP   262 1222 → 49579 Len=220

> Frame 143: 86 bytes on wire (688 bits), 86 bytes captured (688 bits
> Ethernet II, Src: Dell_06:65:fe (b4:45:06:06:65:fe), Dst: QuantaCo_
> Internet Protocol Version 4, Src: 192.168.0.125, Dst: 192.168.0.120
> User Datagram Protocol, Src Port: 1222, Dst Port: 49579
v Data (44 bytes)
    Data: 070000000000000006000000000068656c6c6f770000000000000000000000
    [Length: 44]

0000  54 ab 3a a7 2a b3 b4 45  06 06 65 fe 08 00 45 00
0010  00 48 bc a0 00 00 80 11  00 00 c0 a8 00 7d c0 a8
0020  00 78 04 c6 c1 ab 00 34  82 8b 07 00 00 00 00 00
0030  00 00 06 00 00 00 00 00  68 65 6c 6c 6f 77 00 00
0040  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
0050  00 00 a1 3d 8e 18
```

7. End connection – exit program



```
161 62.888715   192.168.0.125   192.168.0.120   UDP   262 1222 → 49579 Len=220

Frame 161: 262 bytes on wire (2096 bits), 262 bytes captured (2096
Ethernet II, Src: Dell_06:65:fe (b4:45:06:06:65:fe), Dst: QuantaCo_
Internet Protocol Version 4, Src: 192.168.0.125, Dst: 192.168.0.120
User Datagram Protocol, Src Port: 1222, Dst Port: 49579
Data (220 bytes)
    Data: 030000000000000000000000000000000000000000000000000000000000000
    [Length: 220]

0020  00 78 04 c6 c1 ab 00 e4  83 3b 03 00 00 00 00 00
0030  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
0040  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
0050  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
0060  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
0070  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
0080  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
```