Matúš Rusňák ID: 116286

Communication with UDP protocol

In this document I will explain my created protocol and program operation process. This program sends UDP packets which contain my HEADER and potentially data we want to send. It allows text message transfer and file transfer (mathematically up to around 1 TB). This program acts both as a receiver and a sender

1. Header structure of my protocol

1.1 Visualization:

1.2 Packet explanation

packet type (2 bytes) = ACK, FIN, REQUEST, DATA

msg type (2 bytes) = FILE, TEXT, NONE, FAIL

frag num (6 bytes) = num of current fragment

checksum (4 bytes) = checksum of packet

data (0-1458 bytes) = message data we want to send

Total header size = 14 bytes

1.3 Packet in raw data

packet type -> ACK = 00, FIN = 01, REQUEST = 10, DATA = 11

msg type -> NONE = 00, TXT = 01, FILE = 10, FAIL = 11

frag num -> number x where x can be: 0 <x> 2**48

checksum -> 32bit integer representing checksum of header and data

data -> actual data we want to send

1.4 Header description

This packet structure enables you to transfer text messages and files which are under 1 TB size (this will not work however because you will most likely run out of RAM at that point). It works on a simple principle which is that it contains packet type, message type, fragment number, checksum and data.

1.4.1 Packet type

Packet type is used to determine packet's main functionality. ACK packets are used to acknowledge correct transfer of sent packets as well as to keep connection between client and the server. Client sends an ACK packet every 5 seconds to signal the server that it still potentially wants to send messages. FIN packets are used to tell server that one message or file transfer was finished and is ready to be processed. REQUEST packets are used to request for a connection or when a packet is lost or corrupted, they are used to inform the client to resend packets again. DATA packets as the name suggest are used to transfer raw bytes to the server.

1.4.2 Msg type

Message type is used to categorize what type of message was sent. FILE means we are transferring file data. TXT means we are transferring text data. NONE means no data is transferred - these packets can be for example packets to keep connection or request for connection. FAIL message type is used to tell is some type of error occurred during transfer.

1.4.3 Fragment number

Fragment number is used to determine if message will be transferred or not and what is its fragment number. If fragment number is 0 the server will know that that is the only packet it will receive for that message transfer. This can only happen however if we are sending a small text message. When we are for example sending a packet that will contain a FILE DATA the first packet will always represent the name of the file we want to transfer, and it will be followed by another packet which contains the actual file data. Meaning file transfer will be always fragmented. Fragment number can go up to 2**48.

1.4.4 Checksum

Checksum is represented by a 32-bit long integer that is returned from library zlib using crc32() function. It is compared whenever a packet is received (on both sides). If for some reason the compared number do not match server (receiver) will asks for a resend of packets. If client (sender) receives corrupted packet it will respond by sending the last sent packet again, not by requesting the server to resend the corrupted packet.

2. Checksum control

Every time a packet is send the sending side will calculate its checksum. Every time a packet is received the receiving side will recalculate packet’s checksum again. And that is done in this function:

```python
def compare_checksum(msg: bytes) -> bool:
    """
    helper function that compares checksum of sent packet with checksum of received packet
    :param msg: received packet
    :return: check result
    """
    sent_checksum = msg[CHECKSUM_START:CHECKSUM_END]
    server_checksum = msg[:CHECKSUM_START] + msg[CHECKSUM_END:]
    if zlib.crc32(server_checksum).to_bytes(4, 'big') != sent_checksum:
        return False
    return True
```

Function uses method crc32() from library zlib. It separates checksum from received packet and creates new checksum. If new checksum does not equal the sent checksum it returns false. Indicating that a corrupted packet was received

On the other side checksum of packets is created in this function:

```python
def create_check_sum(msg: bytes) -> bytes:
    """
    helper function that returns given packet in bytes
    """
    checksum = zlib.crc32(msg).to_bytes(4, 'big')
    return checksum
```

Function takes an argument msg which can be only header or header with data. It returns create checksum which is later inserted at the end or if msg contained data as well it is inserted between header and data we want to send.

3. ARQ method

For this assignment I decided to implement the Stop & Wait ARQ. Meaning after sending one packet we wait for the answer from receiver. If receiver replies by sending a packet that indicates corruption of packet or when it does not reply at all, and timeout exception is called the sender will send the same packet again until it is not correctly acknowledged by the receiver. Code representation:

```
while True:
    try:
        self.total_number_of_packets_send += 1
        self.send(packet)
        msg, addr = self.client.recvfrom(PROTOCOL_SIZE)
        if not util.compare_checksum(msg):
            print("[ERROR] Server sent corrupted packet, resending packets")
        if msg[PACKET_TYPE_START:PACKET_TYPE_END] == ACK:
            if msg[FRAG_NUM_START:FRAG_NUM_END] == packet[FRAG_NUM_START:FRAG_NUM_END]:
                remaining_tries = 3
                break
        else:
            self.number_of_incorrect_packets_send += 1
            print("[ERROR] Server received corrupted packet, resending packets")
            continue
    except TimeoutError:
        if remaining_tries <= 0:
            self.keep_connection_thread = False
            print(f'[ERROR] {self.target_host} is unreachable')
            self.program_interface.connection_error()
            return
        remaining_tries -= 1
        print("[ERROR] Server did not ACK packet resending packet")
        print(f"[ERROR] Remaining tries {remaining_tries}")
        continue
```

This is a method that is called when we want to send a message that is not fragmented. As we can see from the code, we first send the packet we want to send and then we wait for an answer. If we don't get an answer and the listening is timed out, we resend the packet. If listening is timed out 3 times however, we stop trying to send the packet and inform the user that we can not reach the server (receiver). If a server sends a response packet we check if that packet is an acknowledgment and if the fragment number matches are sent packet. If yes the packet was successfully send and if not we know that the packet was corrupted on the way to the server (receiver) and we resend it again.

4. Connection keeping methods

Whenever we want to send something using my program, we first initialize connection. That is done using a initialize method that sends the server (receiver) a request for connection. If the receiver does not respond 3 times the connection is not established. However, if the receiver receives a request for connection, it creates its own client handler and sends a response to the client informing that a connection has been formed. Once it is formed the server with the created object client handler creates a new thread that waits 6 seconds for a packet to be transferred. If no packet Is sent by that time, it deletes its client handler along with all the data and marks that IP: port address as disconnected. Code representation on the receiver side:

```python
def hold_connection(self) -> None:
    """
    method that measures connection time if user has not timed out
    """
    while self.connection_time != 0 and self.server.running:
        if self.server.stopped():
            sleep(1)
            continue
        sleep(1)
        self.connection_time -= 1
    self.server.remove_connection(self.addr)
```

Every 1 seconds the thread lowers the remaining time (6 seconds) of the connected client. Unless the server is stopped. Since my program is a sender and a receiver at the same time if we try to send from the receiver the server is stopped. After it is back again it has again 6 seconds to hold the connection.

Sender side:

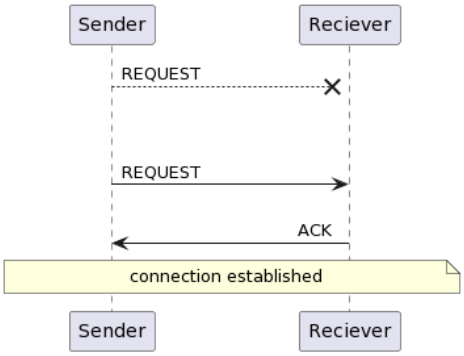
```python
def keep_connection(self) -> None:
    """
    method that sends packet every 5 seconds to keep connection to the server,
    it runs on a separate thread
    """
    while self.keep_connection_thread:
        self.create_and_send_packet(ACK + NONE + NO_FRAGMENT + NO_CHECKSUM)
        sleep(5)
```

This method once again runs on its own thread. And it is called when a connection is established. Every 5 seconds it sends an ACK packet meaning it acknowledges that it is still connected. Since the server does not respond to packets that are meant to keep connection this may still try to send while the server is down, and it will indicate we are still connected. That is okay however because every time we call the command “send” the client (sender) will check if server (receiver) is online. Meaning we will not run into a situation where we want to send something, and the server is offline.

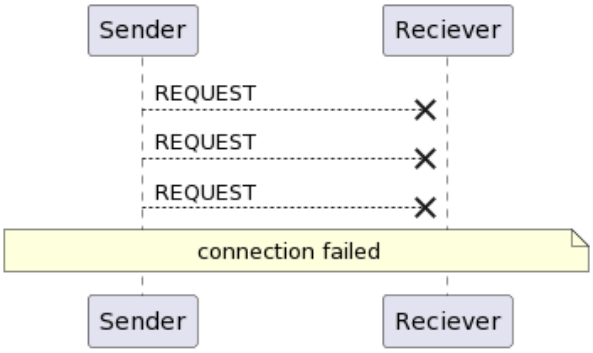
Sender can also use command “close” which will send a packet informing receiver that that client is no longer connected and it will remove its data.

5. Diagrams

5.1 Initialize connection diagram

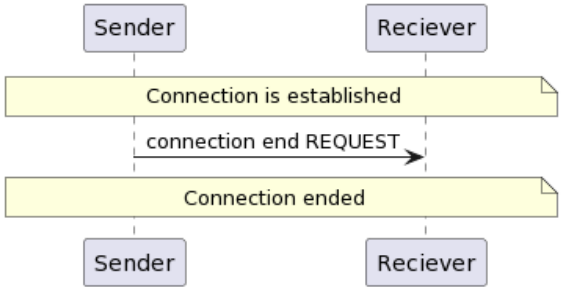


When we initialize a connection, we send a request for connection. If request is acknowledged, we know we have connected successfully.

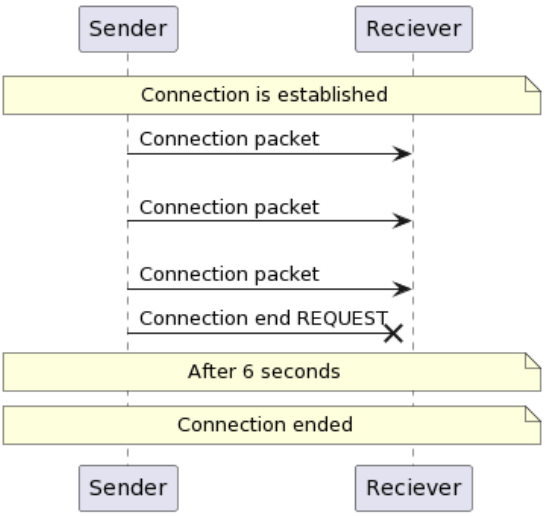


Request for connection has 3 tries. Connection to server fails only if all 3 requests have been timed out.

5.2 Closing connection diagram



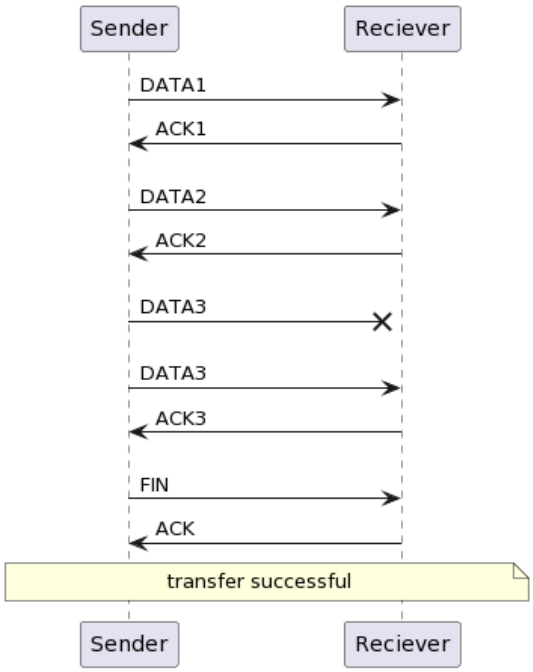
When we want to close connection, we sent a request for closing a connection. Even though it is called request it does not need a reply. If it is sent connection ends immediately.



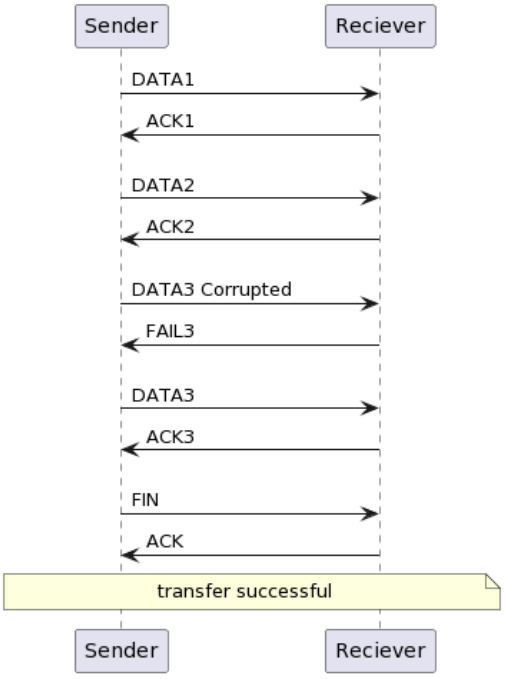
If REQUEST is lost connection ends 6 seconds after last packet was received. Connection is also ended when client disconnects from the internet and server does not receive a packet every 6 seconds, meaning a request to end does not need to sent.

5.3 Data transfer diagram

It is important to that every time we use command send an initialize method is called. Which checks if connection is still active like in part initialize connection diagram



This situation shows when data is lost and therefore not received by the server. If sender is timed out on the acknowledgment from receiver it sends the packet again.

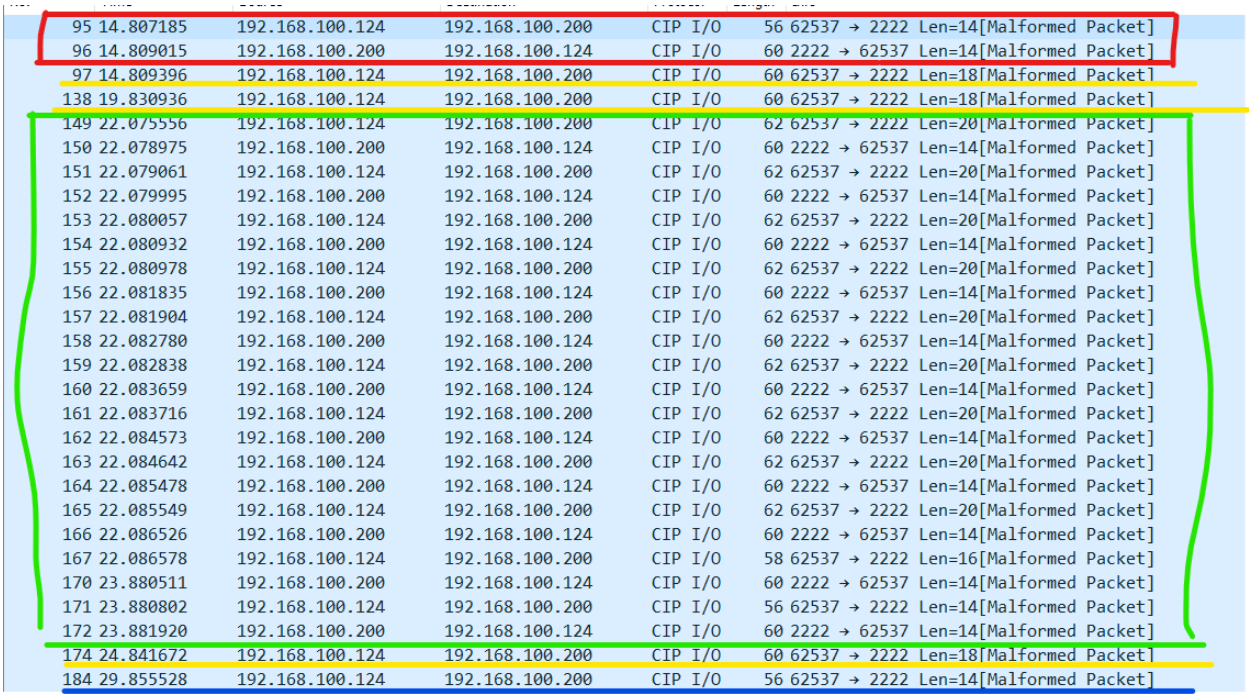


In this situation we send a corrupted packet to the receiver. Receiver informs sender that it was corrupted and sender sends it again.

There is also a situation where receiver sends a corrupted ACK or FIN. In that situation we follow the same pattern in which the sender sends the most recent sent packet again. If single DATA packet is not ACKED 3 times in a row message transfer is terminated.

6. Test scenario

Sending a txt message from 192.168.100.124 (PC1) to 192.168.100.200 (PC2). It is important to note that receiver is made so it can handle multiple clients (senders) at once. It is only limited by computing power.



95	14.807185	192.168.100.124	192.168.100.200	CIP I/O	56	62537 → 2222 Len=14[Malformed Packet]
96	14.809015	192.168.100.200	192.168.100.124	CIP I/O	60	2222 → 62537 Len=14[Malformed Packet]
97	14.809396	192.168.100.124	192.168.100.200	CIP I/O	60	62537 → 2222 Len=18[Malformed Packet]
138	19.830936	192.168.100.124	192.168.100.200	CIP I/O	60	62537 → 2222 Len=18[Malformed Packet]
149	22.075556	192.168.100.124	192.168.100.200	CIP I/O	62	62537 → 2222 Len=20[Malformed Packet]
150	22.078975	192.168.100.200	192.168.100.124	CIP I/O	60	2222 → 62537 Len=14[Malformed Packet]
151	22.079061	192.168.100.124	192.168.100.200	CIP I/O	62	62537 → 2222 Len=20[Malformed Packet]
152	22.079995	192.168.100.200	192.168.100.124	CIP I/O	60	2222 → 62537 Len=14[Malformed Packet]
153	22.080057	192.168.100.124	192.168.100.200	CIP I/O	62	62537 → 2222 Len=20[Malformed Packet]
154	22.080932	192.168.100.200	192.168.100.124	CIP I/O	60	2222 → 62537 Len=14[Malformed Packet]
155	22.080978	192.168.100.124	192.168.100.200	CIP I/O	62	62537 → 2222 Len=20[Malformed Packet]
156	22.081835	192.168.100.200	192.168.100.124	CIP I/O	60	2222 → 62537 Len=14[Malformed Packet]
157	22.081904	192.168.100.124	192.168.100.200	CIP I/O	62	62537 → 2222 Len=20[Malformed Packet]
158	22.082780	192.168.100.200	192.168.100.124	CIP I/O	60	2222 → 62537 Len=14[Malformed Packet]
159	22.082838	192.168.100.124	192.168.100.200	CIP I/O	62	62537 → 2222 Len=20[Malformed Packet]
160	22.083659	192.168.100.200	192.168.100.124	CIP I/O	60	2222 → 62537 Len=14[Malformed Packet]
161	22.083716	192.168.100.124	192.168.100.200	CIP I/O	62	62537 → 2222 Len=20[Malformed Packet]
162	22.084573	192.168.100.200	192.168.100.124	CIP I/O	60	2222 → 62537 Len=14[Malformed Packet]
163	22.084642	192.168.100.124	192.168.100.200	CIP I/O	62	62537 → 2222 Len=20[Malformed Packet]
164	22.085478	192.168.100.200	192.168.100.124	CIP I/O	60	2222 → 62537 Len=14[Malformed Packet]
165	22.085549	192.168.100.124	192.168.100.200	CIP I/O	62	62537 → 2222 Len=20[Malformed Packet]
166	22.086526	192.168.100.200	192.168.100.124	CIP I/O	60	2222 → 62537 Len=14[Malformed Packet]
167	22.086578	192.168.100.124	192.168.100.200	CIP I/O	58	62537 → 2222 Len=16[Malformed Packet]
170	23.880511	192.168.100.200	192.168.100.124	CIP I/O	60	2222 → 62537 Len=14[Malformed Packet]
171	23.880802	192.168.100.124	192.168.100.200	CIP I/O	56	62537 → 2222 Len=14[Malformed Packet]
172	23.881920	192.168.100.200	192.168.100.124	CIP I/O	60	2222 → 62537 Len=14[Malformed Packet]
174	24.841672	192.168.100.124	192.168.100.200	CIP I/O	60	62537 → 2222 Len=18[Malformed Packet]
184	29.855528	192.168.100.124	192.168.100.200	CIP I/O	56	62537 → 2222 Len=14[Malformed Packet]

Red packets – Establishing connection packets

We sent packet to the server (receiver 192.168.100.200) and wait for a response. If we receive an ACK like we did above we can start sending

Yellow packets - Packets that keep connection

Every 5 seconds we send a packet to inform receive that connection is still active. We don't wait for a reply. Sending is stopped during message transfer since it's not needed. Before every message transfer we check from client (sender 192.168.100.124) if receiver is online

Green packets – Data transfer

These packets contain the actual data we want to send. Implementing ARG stop & wait we always wait for an ACK from receiver. At the end we send finalizing packet implying that message transfer is successful and message /file can be displayed/saved. Finalizing packet also waits for an ACK since its arguably the most important packet. Without it the receiver wont display our message or save the file.

6.1 Error simulation

For this program I have also implemented an error simulation. Simulation sends one randomly selected packet and corrupts it. They way you run this simulation test is if you enter an empty message “” you want to send or an empty file name “”. If it’s a message program will generate random message that will be sent with a corrupted packet whereas if its an empty file name the program will send file in test/simulation_file.txt and corrupt one of its data packets. Example:

```
[INPUT] Message you want to send:
[SIMULATION] Starting simulation for TXT message with corrupted packet
[SIMULATION] Generated message: 3NL7QGYU0986OKNG
[ERROR] Server received corrupted packet, resending packet
```

This is an output you will see in the console on the sender side. And this is an output you will see on the receiver side:

```
[CLIENT] ('192.168.100.124', 65140) fragment 1 received
[CLIENT] ('192.168.100.124', 65140) fragment 2 received
[CLIENT] ('192.168.100.124', 65140) fragment 3 received
[CLIENT] ('192.168.100.124', 65140) fragment 4 received
[CLIENT] ('192.168.100.124', 65140) fragment 5 received
[CLIENT] ('192.168.100.124', 65140) fragment 6 received
[CLIENT] ('192.168.100.124', 65140) fragment 7 received
[CLIENT] ('192.168.100.124', 65140) sent corrupted packet, sending error
[CLIENT] ('192.168.100.124', 65140) fragment 8 received
[CLIENT] Message from ('192.168.100.124', 65140) client: 3NL7QGYU0986OKNG
```

7. Usage

If you want to use this program change directory to project default and run the following command:

python main.py

The main commands of this program are: ***send*** – starts connection handling, ***close*** – close an active connection (if there is one), ***exit*** – close the whole program and ***dir*** – this command changes the file received save directory (default is ./Downloads/).

Program will ask you for multiple inputs such as: listening IP and PORT addresses, IP and PORT where you want to send, whether you want to send a file or a text message, fragment size, file name or the actual text message you want to send. Inputs are handled so crash creating inputs should not arise.

Console prints should be informative about what to. At the end of message transmission, a statistical information is outputted. Like the total number of packets send/received, incorrect packets received, message length and others.

8. Changes after initial design - None