

Program and communication protocol design

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

Header structure of my protocol

Visualization:



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

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

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.

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.

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.

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} .

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.

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:

```
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:

```
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.

ARQ functioning

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.

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:

```
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.

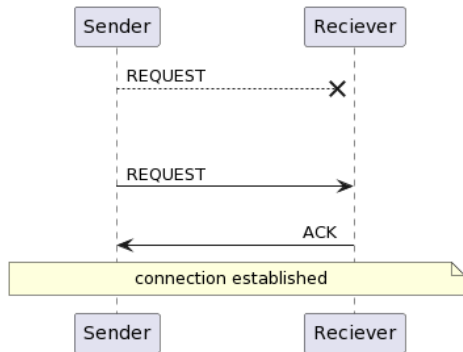
Sender side:

```
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)
```

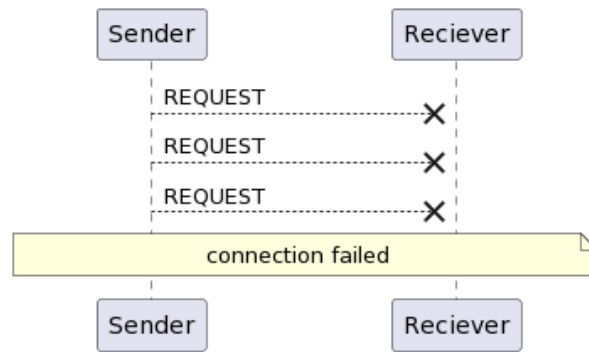
This method once again runs on its own thread. And it is called when a connection is established. Every 5 seconds it sends an ACK packet meaning it acknowledges that it is still connected. Since the server does not respond to packets that are meant to keep connection this may still try to send while 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.

Initialize connection diagram

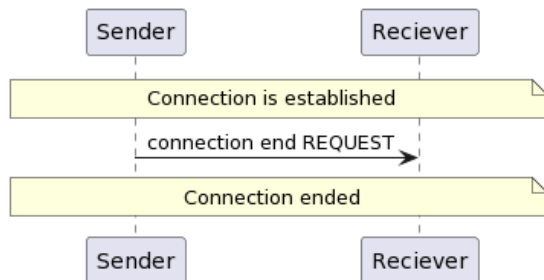


When we initialize a connection, we send a request for connection. If request is acknowledged, we know we have connected successfully.

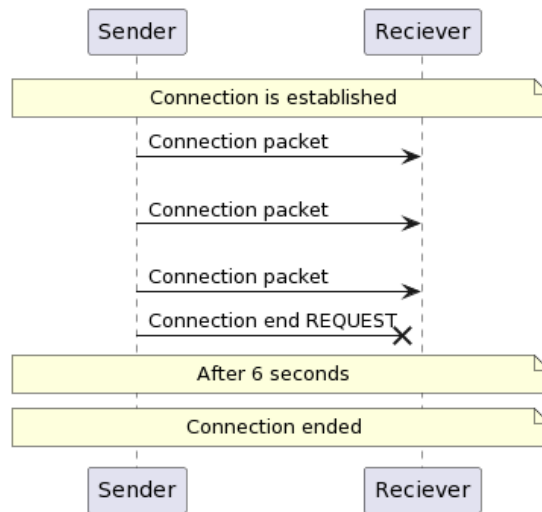


Request for connection has 3 tries. Connection to server fails only if all 3 requests have been timed out.

Closing connection diagram



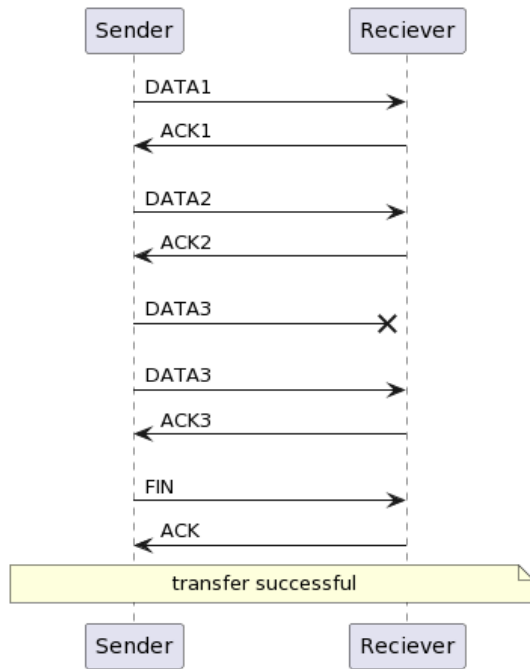
When we want to close connection, we sent a request for closing a connection. Even though it is called request it does not need a reply. If it is sent connection ends immediately.



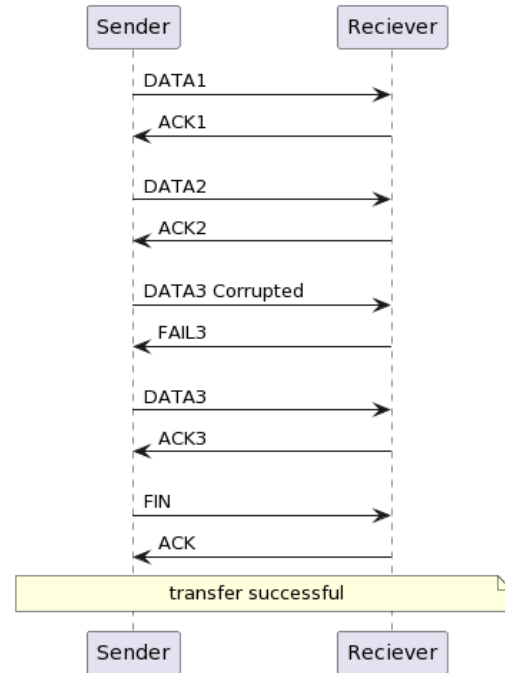
If REQUEST is lost connection ends 6 seconds after last packet was received. Connection is also ended when client disconnects from the internet and server does not receive a packet every 6 seconds, meaning a request to end does not need to sent.

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.