Assignment: Communication application using the UDP protocol

Milan Šeliga

ID: 127279

STU FIIT

Table of Contents

Introduction

We will be implementing our own peer-to-peer UDP communicator app. We design and program our own custom network header on the application layer. We will be using Python programming language for our program, because it contains useful ibraries and provides easier time managing threads and other processes.

Header Structure

TYPE	Fragment Number	Total Fragments	CRC	Flags	Data
1B	2B	2B	4B	1B	

Type field values:

0. **SYN (Initialization)**

- This type represents the initial message to start the connection.
  The sender begins by transmitting a SYN message to request a connection with a peer.

1. **SYNACK (Acknowledgment)**
- This message is sent as a response to the SYN request to acknowledge that the connection request has been received.

2. **ACK (Acknowledgment)**
- This type is used for sending an acknowledgment after receiving a message or fragment.
  It confirms successful reception, helping to ensure reliable data transfer.

3. **NACK (Negative Acknowledgment)**
- This message type signals an error, indicating that the data or fragment was corrupted or missing.
  It requests a retransmission from the sender.

4. **SND (Transfer of Text/File)**
- Used to transmit either a text message or file.
  The actual data is included in the data field, with additional flags used to differentiate between text and file data (e.g., a file fragment).

5. **LSND (Last Fragment)**

- Signifies that the current fragment is the final one in the sending sequence. The receiver knows to expect no further fragments related to this message.

6. **Keep-Alive (KA)**

- This message is periodically sent to verify that the connection is still active. If the receiver fails to respond within a given time frame, the connection will be closed.

7. **END (Connection Termination)**
    - Used to terminate the connection between the peers. It signals that no further communication will take place.

Estabilishing connection using 3-way handshake

We will be using the 3-way handshake, typical of TCP protocol, to estabilish an initial connection to the peer. The steps are:

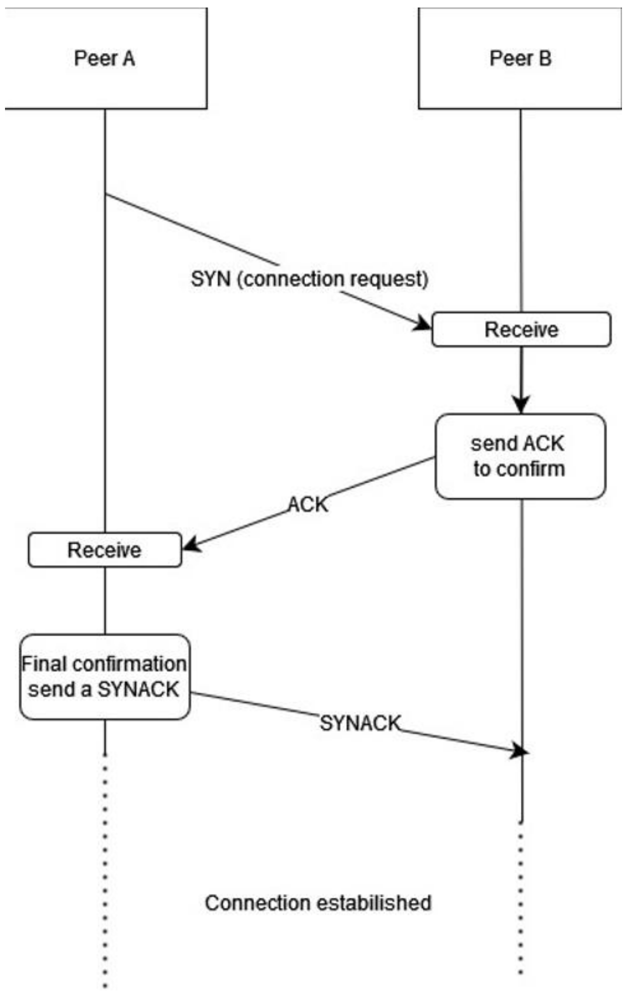
1. **SYN** (Synchronize Request):
    - Peer A initiates the connection by sending a SYN message to Peer B.
        - This message signals Peer A wants to establish a connection with the Peer B.

2. **SYNACK** (SYN Acknowledgment):
    - Upon receiving the SYN message, Peer B responds with a SYNACK message.
        - This message acknowledges that the SYN message was received and that Peer B is ready to establish the connection.

3. **ACK** (Acknowledgment):
    - Peer A, after receiving the SYN-ACK message, sends a final ACK message to Peer B.
        - This message confirms that both sides are ready. Once this ACK message is received by Peer B, the connection is fully established.

Example Flow:

1. Peer A sends a SYN → Server
2. Peer B responds with SYN-ACK → Client
3. Peer A responds with ACK → Server
    - Once this exchange is complete, the connection is established.



1 generall 3-way handshake diagram

Stop-and-Wait ARQ and Fragmentation

For fragmentation of (e.g., large) files, we will use the Stop-and-Wait ARQ protocol to ensure reliable transmission. The file will be divided into smaller fragments that can fit within the network's Maximum Transmission Unit (MTU), with each fragment being sent one at a time.

After each fragment is transmitted, the sender will pause and wait for an acknowledgment (ACK) from the receiver before sending the next fragment.

If the receiver detects an issue with the fragment, it will send a negative acknowledgment (NACK), then the sender has to retransmit the fragment.

This method ensures that each fragment is delivered correctly before proceeding, but may not be fast, since only 1 packet is being processed at any time.

Header size:

MTU (for Wi-Fi): 1500 bytes
Header Size: 10 bytes
Available space for Data: 1500 - 10 = **1490 bytes**

Maintaining connection

We implement Keep-Alive mechanism to ensure that the connection between two peers remains active, even during idle periods.

We periodically send Keep-Alive message

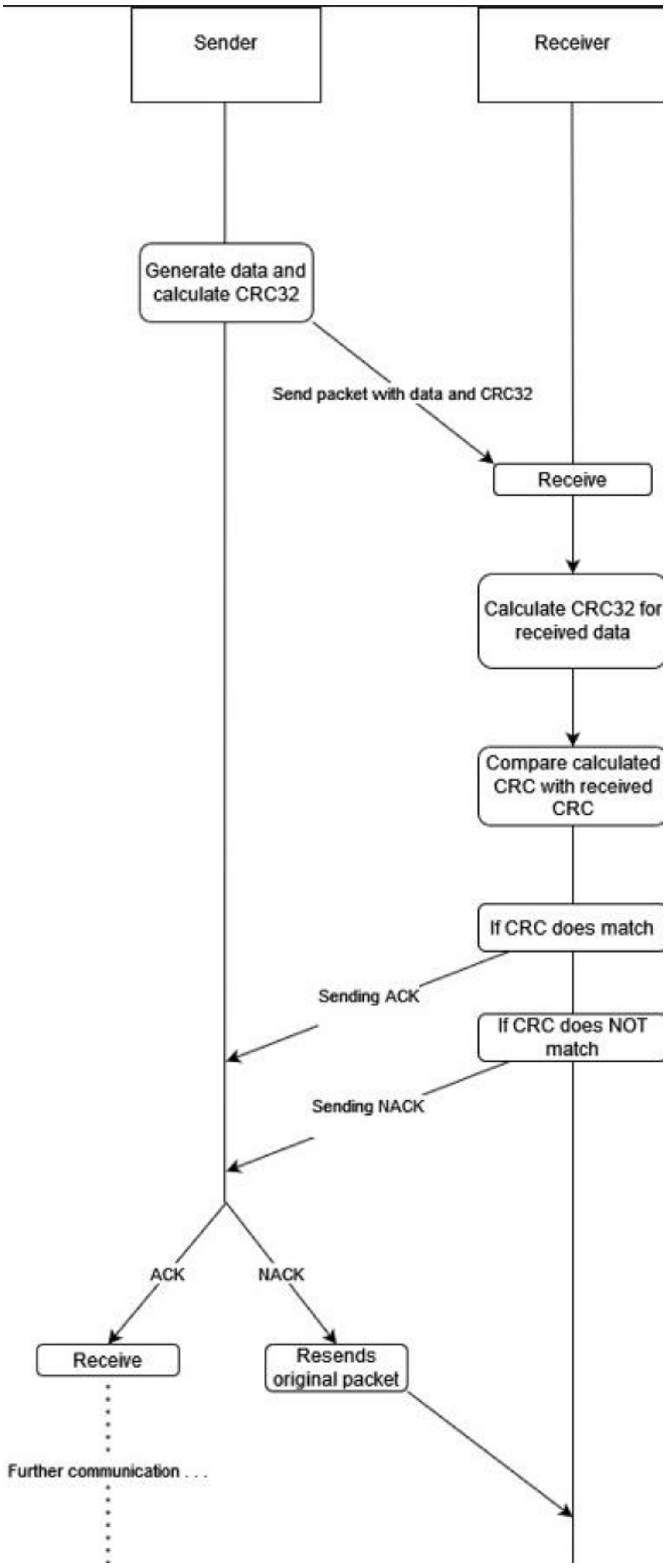
If one peer fails to receive a Keep-Alive message within a 10s window, it assumes the connection has been lost and we close the connection.

Message integrity verification process:

1. Receive Fragments: The receiver collects incoming fragments and checks each one's fragment number and total number of fragments to track which fragments are missing.

2. Check Integrity: The receiver verifies the integrity of each fragment using methods like CRC. If a fragment is missing or corrupted, the receiver sends a NACK to request retransmission of that specific fragment.

3. Reassemble Message: Once all fragments are received and verified, the receiver reassembles the message in the correct order. If the message isn't fragmented, the receiver processes it immediately upon receipt and sends an ACK or NACK depending on integrity verification.


Sender
Receiver
Generate data and calculate CRC32
Send packet with data and CRC32
Receive
Calculate CRC32 for received data
Compare calculated CRC with received CRC
If CRC does match
Sending ACK
If CRC does NOT match
Sending NACK
ACK
NACK
Receive
Resends original packet
Further communication . . .

2 Example of sending, receiving and then verifying the packet

CRC for data integrity

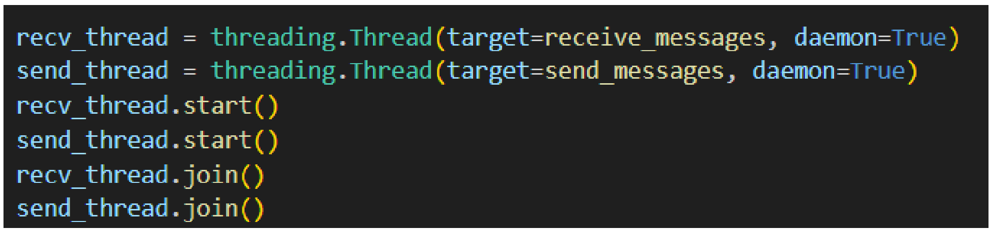
We are using CRC32, which divides the message into 32 bites (thus 4B header), used for calculating whether our packets were damaged somehow or they came in pristine condition. We will be using CRC32 from the standard zlib library.

Here are the steps of CRC verification:

1. When sending a message, a CRC value is calculated based on the message data.
2. This CRC is appended to the message header in the designated CRC field.
3. Upon receiving the message, the recipient recalculates the CRC using the same algorithm and compares it with the CRC value in the header.
4. If the two values match, the message is considered intact.
   If they differ, the message is flagged as corrupted and will be retransmitted using the ARQ stop-and-wait mechanism.

Threading

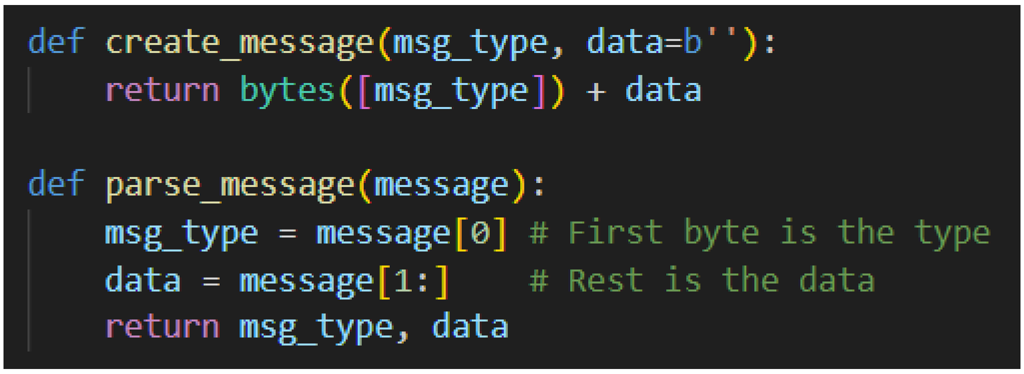
Threading allows the recv_thread() and send_thread to run in parallel on separate threads, thus we can still listen for incoming messages, while at the same time we are still able to send messages to other peer.



```
recv_thread = threading.Thread(target=receive_messages, daemon=True)
send_thread = threading.Thread(target=send_messages, daemon=True)
recv_thread.start()
send_thread.start()
recv_thread.join()
send_thread.join()
```

We start by creating Thread object with the threading.Thread(target, args) function. Then we proceed to (now finnaly) start the coresponding threads, than we wait for each of them to finish execution before continuing

Packet structure and assembly



```python
def create_message(msg_type, data=b''):
    return bytes([msg_type]) + data

def parse_message(message):
    msg_type = message[0] # First byte is the type
    data = message[1:]     # Rest is the data
    return msg_type, data
```

3 Example for KB

Sources

https://www.geeksforgeeks.org/stop-and-wait-arq/

https://www.geeksforgeeks.org/modulo-2-binary-division/

https://datatracker.ietf.org/doc/html/rfc768

https://app.diagrams.net/

https://realpython.com/intro-to-python-threading/