Assignment: Communication application using the UDP protocol

Milan Šeliga

ID: 127279

STU FIIT

Table of Contents

[Header Structure 3](#_Toc180425669)

[Type field values: 3](#_Toc180425670)

[Estabilishing connection using 3-way handshake 4](#_Toc180425671)

[Stop-and-Wait ARQ and Fragmentation 5](#_Toc180425672)

[Message receiving steps: 6](#_Toc180425673)

[CRC for data integrity 8](#_Toc180425674)

[Packet structure and assembly 9](#_Toc180425675)

[Sources 10](#_Toc180425676)

### 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:

1. **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.**

1. **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.**

1. **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.**

1. **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).**

1. **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.**

1. **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.**

1. **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.

1. **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.

1. **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.

#### A diagram of a network Description automatically generatedExample 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

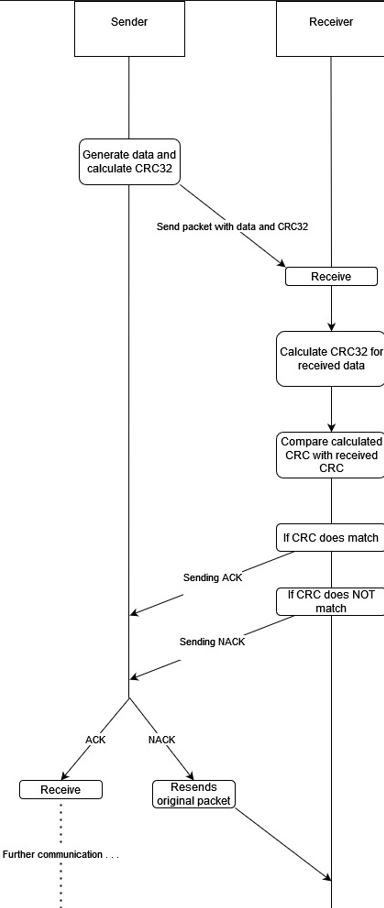
We implement Keep-Alive mechanism to ensure that the connection between two peers remains active, even during idle periods.

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.



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.

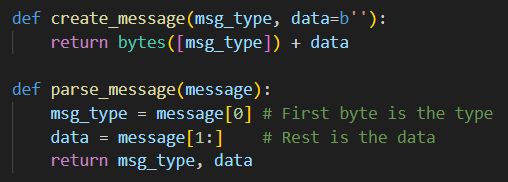
### Threading

A computer screen with text

Description automatically generatedThreading 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.

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



3 Example for KB

Finálne odovzdanie

## Header structure

In the control checkpoint i designed my header structure to hold 10 Bytes, but during implementation, we could change up or straight up remove some fields, because we can use other fields to fulfill the same process.

Our example was the Flags field, where i imagined it would hold the values of the Type variables. But we can merge Type and Flags into 1 field of 1B -> Type. We also ended up changing the order and some of the Type variables altogether. We added types for NAK when trying retransmitting corrupted messages, fragmentation of the messages, fragmentation of the files . . .

We also omitted the Total Fragments field and instead we just send fragments and discern the last using a separate type (MSG\_LAST\_FRAGMENT/MSG\_FILE\_LAST\_FRAGMENT) seq. Crc

We changed fragment number into a sequence number, only change was we added 2 bytes so now the Sequence number field is 4 Bytes long.

And we kept CRC field untouched, mostly because it was already useful and important field, for verifying the integrity of received files.

+------------------+---------+

| Field Name | Size |

+------------------+---------+

| Type | 1 byte |

| Sequence Number | 4 bytes |

| CRC | 4 bytes |

| Payload Data | Variable|

+------------------+---------+

### Header fields:

1. **MSG\_DEFAULT:** Standard message type for general communication.

Used for sending small messages that fit within a single packet.

1. **MSG\_SYN:** Initiates a connection request.
2. **MSG\_SYN\_ACK:** Acknowledges a connection initiation.
3. **MSG\_ACK:** A general acknowledgment message to confirm the receiving of various messages or fragments.
4. **MSG\_FIN:** Signals for grateful termination of a connection.
5. **MSG\_KEEP\_ALIVE:** Maintains an active connection.
6. **MSG\_FRAGMENT:** Represent a fragment of a larger message that requires fragmentation. Usually sent along with other fragments to make up a single file / message after reordering and joining of the PDUs.
7. **MSG\_LAST\_FRAGMENT:** Marks the final fragment in a sequence.
8. **MSG\_FILE\_INFO:** Sends metadata about a file being transferred. It’s sent before sending the contents.
9. **MSG\_FILE\_FRAGMENT:** Represents a fragment of a file. Used for sending larger messages that need to be split into multiple fragments.
10. **MSG\_FILE\_LAST\_FRAGMENT:** Denotes the last fragment of a file.
11. **MSG\_FRAGMENT\_NAK:** Requests retransmission by sending a nak, due to a detected error. Negative acknowledgment indicates that a fragment was not received correctly.

## Message Creation and Parsing

**create\_message(msg\_type, sequence\_number, data=b''):**

Constructs a message with the specified type, sequence number, and data. It computes a CRC-32 checksum for data integrity and assembles the message in the defined format. It does it in these steps:

1. *We calculate the CRC-32 through zlib.crc32 function*
2. *CRC checksum is converted into 4 bytes along with sequence number.*
3. *By concatenating [ 1B msg\_type + 4B sequence number + 4B CRC checksum and than the rest of the data/message, we construct the message.*

These steps provide us a consistent interpretation of messages by the receiver.

**parse\_message(message):**

Deconstructs a received message to extract the message type, sequence number, data, and verifies the CRC-32 checksum to ensure data integrity.

## Connection Management

The connection set up process ensures that both the client and the server are ready for data exchange, even if UDP is a connectionless protocol. We can achieve this through a **three-way handshake** process and implementing keep-alive system for maintaining this connection made with 3-Way HS

### Three-Way HandshakeA diagram of a network Description automatically generated

We took inspiration from the TCP-like behavior for reliability.

The steps remain the same as during control checkpoint:

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.

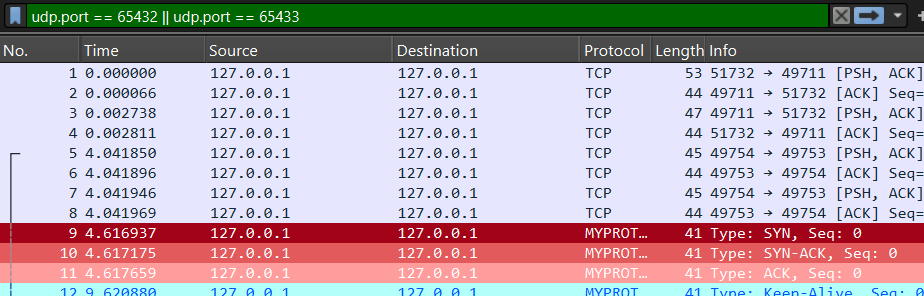
1. **SYNACK** (SYN Acknowledgment):

* 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.

1. **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, visualized via wireshark and our custom lua script:



If any step in the handshake fails due to packet loss, the sender will retry sending [MSG\_SYN](vscode-file://vscode-app/c:/Program%20Files/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) up to a maximum number of retries ([max\_retries=3](vscode-file://vscode-app/c:/Program%20Files/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-sandbox/workbench/workbench.html) ).

### Keep-Alive Mechanism

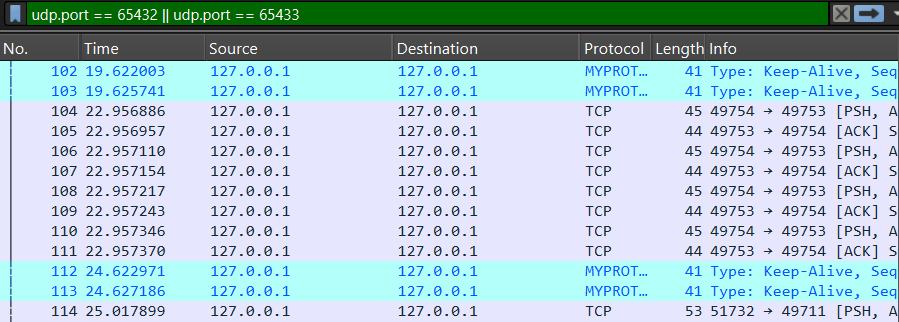
The Keep-Alive mechanism is integral to maintaining an active and responsive connection between the peers.

* *It addresses the stateless nature of UDP by ensuring that the connection remains active.*

##### Keep-Alive PDU

* *Utilizes the Type field, using the value of 5 (MSG\_KEEP\_ALIVE)*
* *It has no additonal data, making it lightweight for continous sending*
* *This packet is sent at regularly to confirm connection is still up*
* When either peer does not receive a keep-alive message even after 3 timeout intervals *(15s).* After that, *the connection is closed and terminated.*

Example keep-alive process flow, visualized via wireshark and our custom lua script:



## CRC32

We employed CRC32 to **detect any modifications** to the *transmitted data*.

## We use the *zlib* library for calculating CRC using the function *zlib.crc32().* This results in a checksum value, which the peers can use to verify the integrity of the data by comparing the **transmitted checksum** with a **newly calculated** **one** on the *received data*. If the values match, the data is **intact**; otherwise, it indicates corruption and triggers *retransmission*.

#### Calculating CRC32

CRC32 is calculated in zlib.crc32 or in general by processing the input data byte by byte through a predefined polynomial-based algorithm in zlib.crc32 the polynomial used is **0xEDB88320 (IEEE 802.3)**. Each byte is combined with the current CRC value using a bitwise XOR operation, and the result is shifted and modified according to the polynomial. This process iteratively updates the CRC value for each byte, producing a final checksum that represents the entire data.

The output is a 32-bit (CRC**32**) integer that serves as a representation of the data’s integrity.

A screenshot of a computer program

Description automatically generated

*Example of calculation*

## ARQ

After trying to implement go-back-N countless of times I could not manage the connection and it would ultimately always fail. So I decided to stick to Stop-and-Wait ARQ

Simulating corrupted message

Header size

## Wireshark and overview

## External resources:

<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/>