
# Client for a Chat Server Using IPK24-CHAT Protocol

**IPK 1 Project**

*Documentation*

Aleksander Postelga
xposte00

**Date:** April 1, 2024

## Contents

**1 Introduction** **2**

**2 Implementation Details** **3**
2.1 TCP Implementation . . . . . 3
2.2 UDP Implementation . . . . . 4

**3 Testing** **5**
3.1 TCP testing . . . . . 5
3.2 UDP . . . . . 6

**4 Conclusion** **8**

**5 Sources** **8**

# 1 Introduction

The main goal of the project is to develop and implement chat application, which is able to communicate with a remote server using the IPK24-CHAT protocol. The protocol has got two variants - each built on top of a different transport protocol.

The first variant is a TCP transport protocol that allows us a reliable communication channel, ensuring that messages are delivered in the order they were sent and without loss.

The second variant utilizes the UDP transport protocol, which offers faster message delivery by foregoing the overhead of ensuring order and reliability. UDP's connectionless communication model allows for quick data transmission without the need for establishing and maintaining a connection, making it efficient for broadcasting messages to multiple recipients.

Program execution:
You can compile program by command *make* and execute it by *./ipk24chat-client*

Execution of the program supports this number of arguments:

- -t [tcp—udp] You can choose what transport protocol to use
- -s [address] Server IP address or hostname
- -p [port] Port number the default is 4567
- -d [ms] Confirmation timeout for udp default is 250ms
- -r [count] Maximum number of udp retransmissions default is 3
- -h prints help to program execution

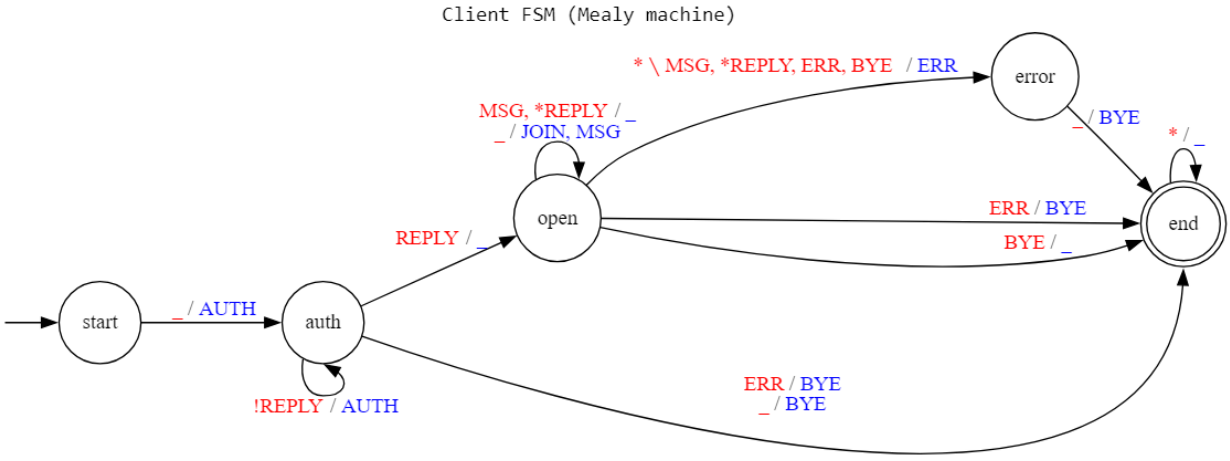
In table 1 there are commands that user can use in client

| **Command** | **Parameters** | **Behavior** |
|---|---|---|
| /auth | Username Secret DisplayName | Sends authentication message to the server |
| /join | ChannelID | Requests server to join the channel with channelID |
| /rename | DisplayName | Change user display name |
| /help | | Writes the help message with all commands |

Table 1: Client commands

# 2 Implementation Details

The program's operation is governed by an FSM (Finite State Machine) (picture 1), which assists us in efficiently managing the communication process between the client and the server. By defining a clear set of states and transitions based on the incoming and outgoing messages, the FSM ensures that the program behaves predictably in response to various events. This structured approach facilitates handling complex interactions, such as authentication, message exchange, and error management, by guiding the client through a series of well-defined steps. More details about the FSM machine are written on the project page.




For better functioning i have two global structures *user* and *mc* (message content). User structure has all information about user, like his username, display name and password also it has time values, retries and message id for udp implementation. Message content structure is used for easy information handling which is sent by server.

## 2.1 TCP Implementation

TCP operates as follows: when it is determined that a user wishes to establish a connection using the TCP transport protocol, the *tcp_client* function is initiated. This function creates a socket, establishes a connection with the server, and initiates the operation of the FSM (Finite State Machine). The FSM, in turn, transitions to the *auth* state and starts the *fsm_auth_state_TCP* function. This function awaits the user to input the */auth* command with the correct parameters and to receive confirmation from the server that the user has been successfully authenticated and connected to the default channel (general). Following successful authentication, the FSM transitions to the OPEN state. If an error occurs, the connection to the server is terminated, and the client is disconnected.

In the OPEN state, the core logic of operation is enacted. The FSM executes the logic contained within the *fsm_open_state_tcp* function. One of the most challenging aspects to implement was handling incoming messages while the user is typing a message. This issue was resolved using the *select* function, which allows for the parallel checking of messages from the server and stdin. Upon receiving a message from the server, the *handle_incoming_message* and *parse_tcp_input* functions parse this message and execute logic according to the message content. In reading mode from stdin, user commands such as */join*, */rename*, */help* are checked, and logic corresponding to these commands is executed. If none of the aforementioned commands are recognized, it indicates that the user has sent a message. In the specified TCP format, the user's message is then sent to the server.

This structured approach ensures a streamlined process for establishing TCP connections, authenticating users, and handling both incoming server messages and user input in a manner that aligns with the format of official documentation. The utilization of FSM for managing

different states of the connection, combined with the selective reading of inputs, allows for an efficient and effective communication protocol within the application.

## 2.2 UDP Implementation

UDP implementation works the same way as TCP. But it has a few changes: Construct_message function which construct a message to be sent, I send the message with send_UDP_message function, which sends and waits for confirmation from the server.

Also my implementation of udp has a solution for protocol issues like packet loss and packet delay/duplication.

The solution for packet loss is having cofirmation timeouts and confirmation retries. Then the message from client is sent, function waits before the server sends the confirmation of message if it waits too long the message will be sent again and function will wait again for the confirmation. Client by default has 3 attempts (1 initial + 3 retries) to get confirmation from the server the interval between attempts by default is 250 ms. After all retries client sends the error and bye message to server.

The solution of packet delay/duplication is having the message id which is sent to server with all message, rather than confirm. The message id is a 2 byte number for a unique identifiation of a particular message. It is sent in network byte order, to transform it i used function *htons*. After the message is sent i increment the message id in structure user, and the new message will be sent with incremented value of message id.

This method improves how we use UDP by adding features to fix common issues like missing messages and delays. Typically, UDP is quick but doesn't guarantee that messages are received in the right order, or at all. With these adjustments, UDP becomes a more reliable choice for situations where speed is crucial but you also want to make sure your messages are correctly received. This is especially useful for real-time applications where the slowness of more reliable methods like TCP isn't acceptable.

The most challenging in udp implementation is to implement dynamic port. My implementation is: after, an auth message from client is sent to default port 4567 (or port what user provide), the next message, rather than confirm, will be written and saved in global structure user which have server address variable, and then all messages will be sent and received with this new information.

# 3 Testing

For local testing, i create a mock server to simulate the behavior of a real server. This mock server establishes connections with multiple clients and can send and receive messages to them. It does not have the logic of channels and other features; it was created only for basic testing of key aspects of the program, such as connection establishment, format verification of sent messages, and reception and processing of messages. The main testing was conducted using a Public Reference Server to check the program against real-world scenarios. In this section i will provide wireshark and testing results using public reference server.

## 3.1 TCP testing

On wireshark screenshot below is represented testing of tcp client variant using reference public server for real-world tests

On this screenshot is tested authentication with bad data. After sending the authentication message we receive a reply that user secret is not valid. After that we can send authentication message again.

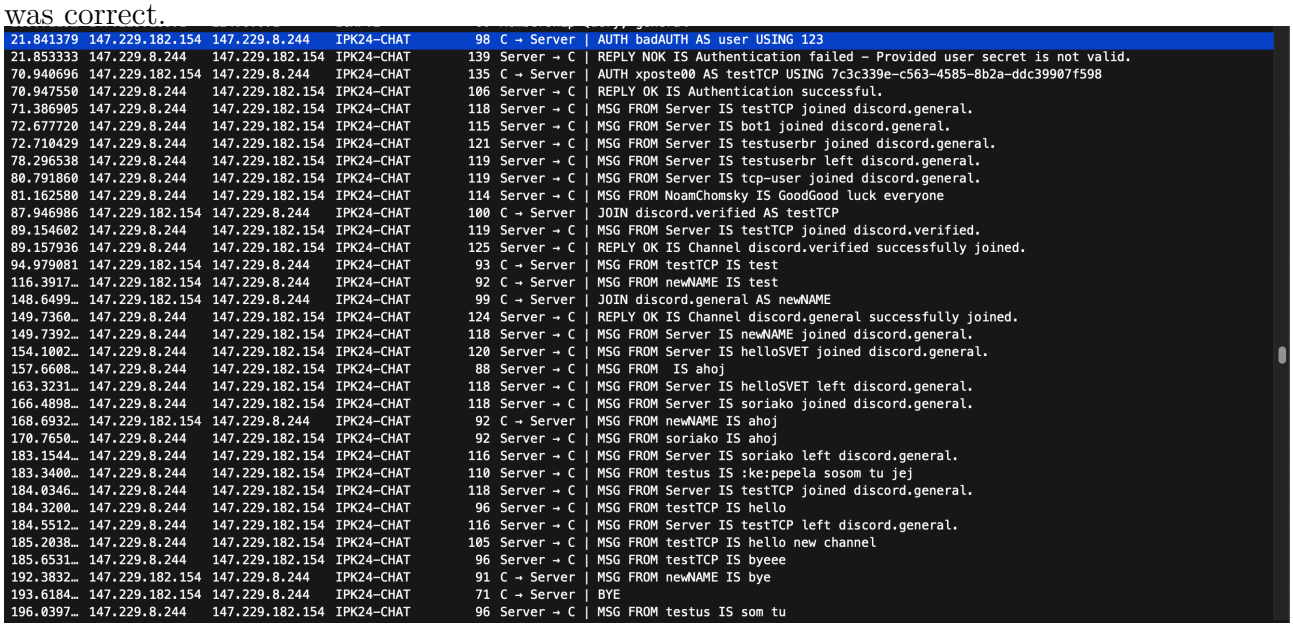
After that i provide a good data and send authentication message again the server accept it and I joined general channel.

Next was tested receiving the messages from server how it seen on screenshot I correctly receive all messages.

Testing of joining another channel also was correct, I correctly receive a message from server that I joined another channel after I've send a join message.

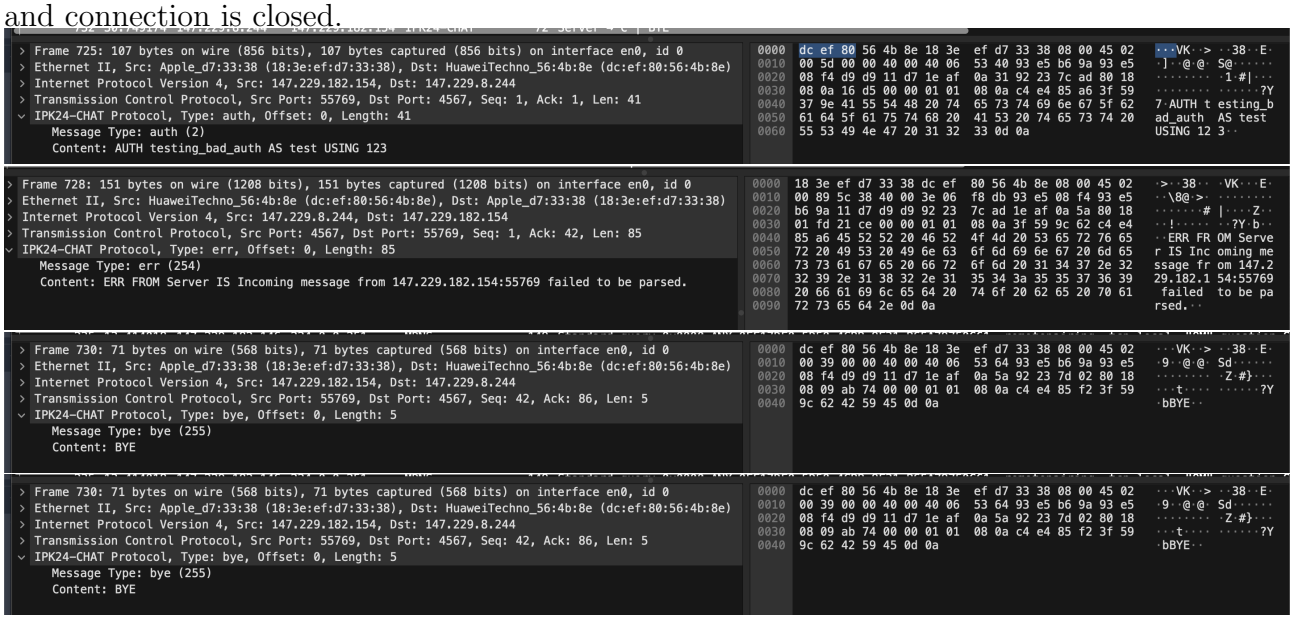
Test of sending messages, message correctly was send Test of renaming, I locally rename myself and send new message with new name, it also correctly was send

The last send is propper closing connection, after C-c the client sent a BYE message, what was correct.



| | | | | | | |
|---|---|---|---|---|---|---|
| 21.841379 | 147.229.182.154 | 147.229.8.244 | IPK24-CHAT | 98 | C → Server | AUTH badAUTH AS user USING 123 |
| 21.853333 | 147.229.8.244 | 147.229.182.154 | IPK24-CHAT | 139 | Server → C | REPLY NOK IS Authentication failed - Provided user secret is not valid. |
| 70.940696 | 147.229.182.154 | 147.229.8.244 | IPK24-CHAT | 135 | C → Server | AUTH xposte00 AS testTCP USING 7c3c339e-c563-4585-8b2a-ddc39907f598 |
| 70.947550 | 147.229.8.244 | 147.229.182.154 | IPK24-CHAT | 106 | Server → C | REPLY OK IS Authentication successful. |
| 71.386905 | 147.229.8.244 | 147.229.182.154 | IPK24-CHAT | 118 | Server → C | MSG FROM Server IS testTCP joined discord.general. |
| 72.677720 | 147.229.8.244 | 147.229.182.154 | IPK24-CHAT | 115 | Server → C | MSG FROM Server IS bot1 joined discord.general. |
| 72.710429 | 147.229.8.244 | 147.229.182.154 | IPK24-CHAT | 121 | Server → C | MSG FROM Server IS testuserbr joined discord.general. |
| 78.296538 | 147.229.8.244 | 147.229.182.154 | IPK24-CHAT | 119 | Server → C | MSG FROM Server IS testuserbr left discord.general. |
| 80.791860 | 147.229.8.244 | 147.229.182.154 | IPK24-CHAT | 119 | Server → C | MSG FROM Server IS tcp-user joined discord.general. |
| 81.162580 | 147.229.8.244 | 147.229.182.154 | IPK24-CHAT | 114 | Server → C | MSG FROM NoamChomsky IS GoodGood luck everyone |
| 87.946986 | 147.229.182.154 | 147.229.8.244 | IPK24-CHAT | 100 | C → Server | JOIN discord.verified AS testTCP |
| 89.154602 | 147.229.8.244 | 147.229.182.154 | IPK24-CHAT | 119 | Server → C | MSG FROM Server IS testTCP joined discord.verified. |
| 89.157936 | 147.229.8.244 | 147.229.182.154 | IPK24-CHAT | 125 | Server → C | REPLY OK IS Channel discord.verified successfully joined. |
| 94.979081 | 147.229.182.154 | 147.229.8.244 | IPK24-CHAT | 93 | C → Server | MSG FROM testTCP IS test |
| 116.3917… | 147.229.182.154 | 147.229.8.244 | IPK24-CHAT | 92 | C → Server | MSG FROM newNAME IS test |
| 148.6499… | 147.229.182.154 | 147.229.8.244 | IPK24-CHAT | 99 | C → Server | JOIN discord.general AS newNAME |
| 149.7360… | 147.229.8.244 | 147.229.182.154 | IPK24-CHAT | 124 | Server → C | REPLY OK IS Channel discord.general successfully joined. |
| 149.7392… | 147.229.8.244 | 147.229.182.154 | IPK24-CHAT | 118 | Server → C | MSG FROM Server IS newNAME joined discord.general. |
| 154.1002… | 147.229.8.244 | 147.229.182.154 | IPK24-CHAT | 120 | Server → C | MSG FROM Server IS helloSVET joined discord.general. |
| 157.6608… | 147.229.8.244 | 147.229.182.154 | IPK24-CHAT | 88 | Server → C | MSG FROM  IS ahoj |
| 163.3231… | 147.229.8.244 | 147.229.182.154 | IPK24-CHAT | 118 | Server → C | MSG FROM Server IS helloSVET left discord.general. |
| 166.4898… | 147.229.8.244 | 147.229.182.154 | IPK24-CHAT | 118 | Server → C | MSG FROM Server IS soriako joined discord.general. |
| 168.6932… | 147.229.182.154 | 147.229.8.244 | IPK24-CHAT | 92 | C → Server | MSG FROM newNAME IS ahoj |
| 170.7650… | 147.229.8.244 | 147.229.182.154 | IPK24-CHAT | 92 | Server → C | MSG FROM soriako IS ahoj |
| 183.1544… | 147.229.8.244 | 147.229.182.154 | IPK24-CHAT | 116 | Server → C | MSG FROM Server IS soriako left discord.general. |
| 183.3400… | 147.229.8.244 | 147.229.182.154 | IPK24-CHAT | 110 | Server → C | MSG FROM testus IS :ke:pepela sosom tu jej |
| 184.0346… | 147.229.8.244 | 147.229.182.154 | IPK24-CHAT | 118 | Server → C | MSG FROM Server IS testTCP joined discord.general. |
| 184.3200… | 147.229.8.244 | 147.229.182.154 | IPK24-CHAT | 96 | Server → C | MSG FROM testTCP IS hello |
| 184.5512… | 147.229.8.244 | 147.229.182.154 | IPK24-CHAT | 116 | Server → C | MSG FROM Server IS testTCP left discord.general. |
| 185.2038… | 147.229.8.244 | 147.229.182.154 | IPK24-CHAT | 105 | Server → C | MSG FROM testTCP IS hello new channel |
| 185.6531… | 147.229.8.244 | 147.229.182.154 | IPK24-CHAT | 96 | Server → C | MSG FROM testTCP IS byeee |
| 192.3832… | 147.229.182.154 | 147.229.8.244 | IPK24-CHAT | 91 | C → Server | MSG FROM newNAME IS bye |
| 193.6184… | 147.229.182.154 | 147.229.8.244 | IPK24-CHAT | 71 | C → Server | BYE |
| 196.0397… | 147.229.8.244 | 147.229.182.154 | IPK24-CHAT | 96 | Server → C | MSG FROM testus IS som tu |

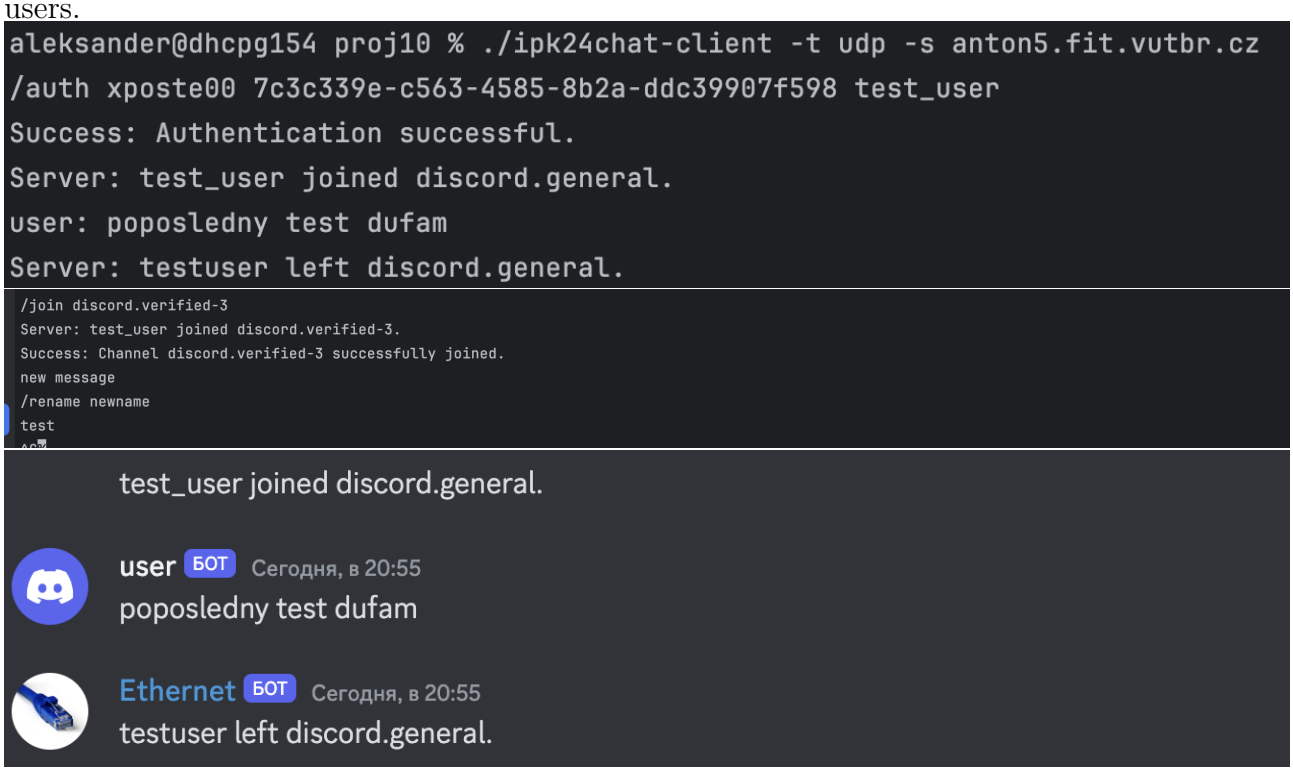
Authentication error: on screenshots below is seen how the client works if after sending a authentication client receives error message. After client receives error the bye message is sent and connection is closed.

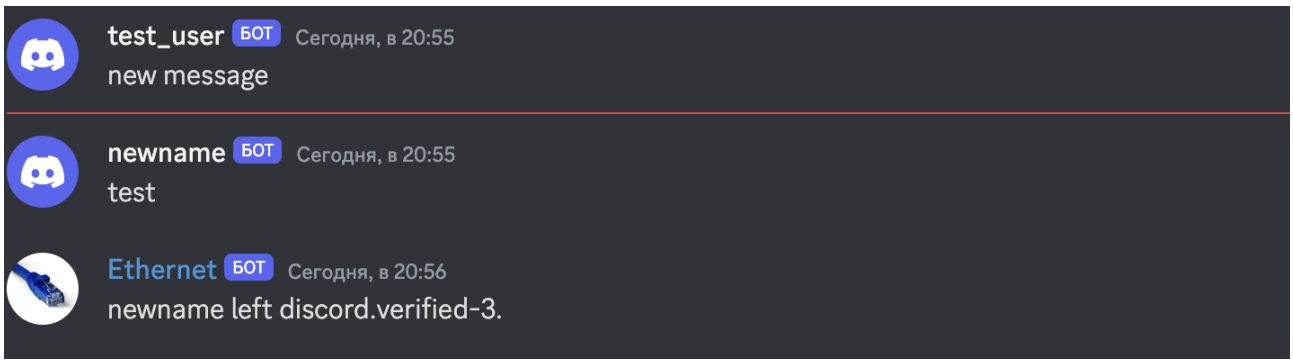



## 3.2 UDP

The test was on reference public server and screenshots will be provided from discord for greater comprehension and readability of the reader because messages are binary.

On screenshots bellow you can see tests of main implementation details, like authentication, joining server, sending messages, changing name and receiving messages from server and other users.




test_user БОТ Сегодня, в 20:55
new message
newname БОТ Сегодня, в 20:55
test
Ethernet БОТ Сегодня, в 20:56
newname left discord.verified-3.

# 4 Conclusion

The IPK24-CHAT client project, aiming to develop a chat application leveraging TCP and UDP protocols, offered a deep dive into network communications' intricacies. Successfully integrating TCP for reliable, ordered message delivery and UDP for fast, broadcast capabilities, the project demonstrated the effectiveness of using both protocols in harmony. A Finite State Machine (FSM) facilitated structured communication flow, ensuring smooth user interaction and reliable message exchange. Challenges such as adapting UDP to enhance reliability through confirmation timeouts and message IDs were overcome, showcasing innovative problem-solving. Extensive testing, including local simulations and public server interactions, validated the application's performance and robustness. This project not only enhanced my understanding of network protocols but also highlighted the importance of testing and innovation in software development.

# 5 Sources


- Catch CTRL+C in C: `https://stackoverflow.com/questions/4217037/catch-ctrl-c-in-c`
- Programming network applications: `https://moodle.vut.cz/pluginfile.php/823898/mod_folder/content/0/IPK2023-24L-04-PROGRAMOVANI.pdf`
- setsockopt function: `https://pubs.opengroup.org/onlinepubs/000095399/functions/setsockopt.html`
- RFC 1350 - The TFTP Protocol: `https://datatracker.ietf.org/doc/html/rfc1350#autoid-4`