Computer and Communication Networks: Semester Assignment - Communication Using the UDP Protocol

Marián Kurčina

ZS 2024

xkurcinam@stuba.sk
marian.kurcina2003@gmail.com

Table of Contents

Introduction

My task is to implement a P2P application using a custom protocol and UDP, capable of sending messages and files. The main part of the assignment was to design a custom protocol that meets the following requirements:

a) Establishing a connection between two parties and negotiating connection parameters
b) Sending data in fragments of a user-specified size
c) Verifying the integrity of the received message
d) Resending lost or corrupted messages
e) Ensuring mutual control between nodes to check if the participant on the other side is still active
f) Intentionally creating an error in one of the transmitted messages to simulate corrupted data

Based on these conditions, I designed my own protocol and described the processes executed during the application's operation.

I programmed the application in Python using the following libraries: os, socket, threading, time, struct, random, tkinter, and queue.

Designed Protocol

<table>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td><td>13</td><td>14</td><td>15</td><td>16</td><td>17</td><td>18</td><td>19</td><td>20</td><td>21</td><td>22</td><td>23</td><td>24</td><td>25</td><td>26</td><td>27</td><td>28</td><td>29</td><td>30</td><td>31</td></tr>
<tr><td colspan="8">1B</td><td colspan="8">2B</td><td colspan="8">3B</td><td colspan="8">4B</td></tr>
<tr><td>H</td><td>E</td><td>K</td><td>D</td><td>A</td><td>R</td><td>M</td><td>+</td><td colspan="24">Fragment Number</td></tr>
<tr><td colspan="8">Fragment number</td><td colspan="16">Window size</td><td colspan="8">Checksum</td></tr>
<tr><td colspan="8">Checksum</td><td colspan="24">(Data)</td></tr>
</table>

My protocol has a header size of 9B and consists of Flags, Fragment Number (Number of fragments), Window Size (Max window size), Checksum, and Data.

The first part of the header is **Flags**, which identifies the type of message. In its binary representation, each digit indicates whether the message belongs to a specific category (0 – No, 1 – Yes):

Handshake – Identifies a handshake message.

Exit – Identifies a connection termination message.

Keepalive – Identifies a Keep-Alive message.

Datatransfer – Identifies messages used for data transfer.

ACK – Identifies acknowledgment messages.

REQ – Identifies messages requesting retransmission of corrupted data.

Msg – Identifies a text message (without saving).

Additional information – Used to identify the third handshake message and fragmented messages.

The next part is **Fragment Number**, which identifies the order of a fragment during fragmentation. When negotiating transfer parameters before sending a file, it specifies the total number of fragments.

Window Size determines the window size for **Selective Repeat** and specifies the maximum window size during parameter negotiation.

Checksum is used to verify data integrity. It is computed using **CRC 16-CCITT**.

The final part is **Data**, which carries the actual transmitted data. During file transfer parameter negotiation, this section contains the file name.

Message Types

10000000 – First handshake message
10001000 – Second handshake message
10001001 – Third handshake message

01000000 – Exit message
01001000 – Exit message acknowledgment

00100000 – Keepalive message
00101000 – Keepalive acknowledgment

00010010 – Unfragmented text message
00011010 – Unfragmented text acknowledgment
00010110 – Request for retransmission of unfragmented text

10010010 – Negotiation message for fragmented text transfer
10011010 – Acknowledgment of fragmented text transfer negotiation
10010110 – Request for retransmission of fragmented text transfer negotiation

10010000 – Negotiation message for file transfer
10011000 – Acknowledgment of file transfer negotiation
10010100 – Request for retransmission of file transfer negotiation

00010000 – Message for sending a file fragment
00011000 – Acknowledgment of received file fragment
00010100 – Request for retransmission of a file fragment

Method Description

Connection Initiation

At the beginning, User 1 attempts to establish a connection by sending a **handshake1** message. User 2 receives it and sends back an **acknowledgment – handshake2**. Once this message reaches User 1, they send an acknowledgment to User 2. This process tests the connection in both directions, and from this point onward, the connection will be regularly tested using the **Keep-Alive** method.

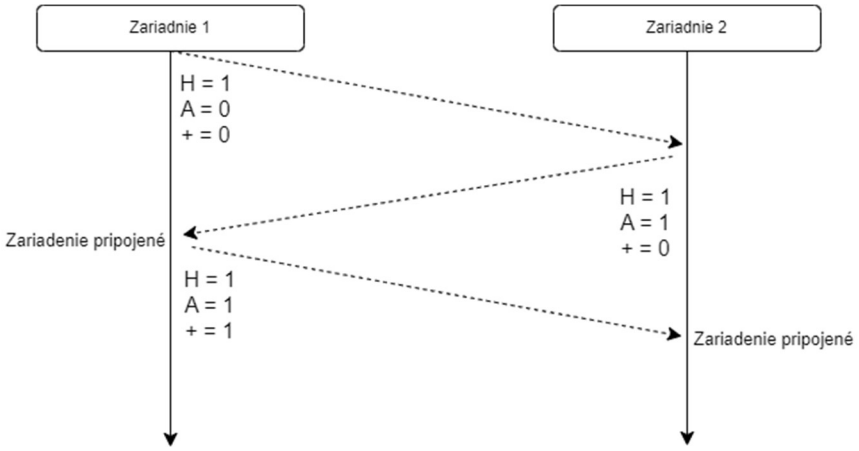

Diagram 1: Connection Initiation

Connection Termination

To terminate the connection, User 1 sends an **exit** message to User 2. User 2 receives it and sends an acknowledgment back. Once this process is completed, the **Keep-Alive** testing stops.

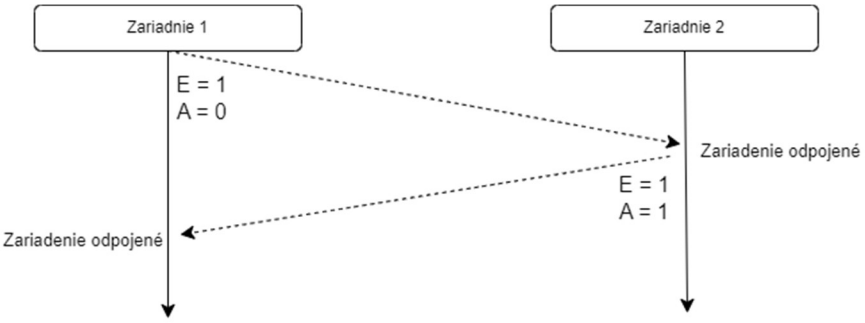

Diagram 2: Connection Termination

Connection Maintenance

To maintain the connection, the **Keep-Alive** method will be used, with regular **keepalive** messages sent from both communicators. During periods of inactivity (no messages being sent), the connection will be checked every 5 seconds. If 3 consecutive messages go unanswered, the connection will be terminated, processes will stop, and the user will be required to attempt reconnection. Once reconnection is successful, the user can resume processes from before the connection was terminated.

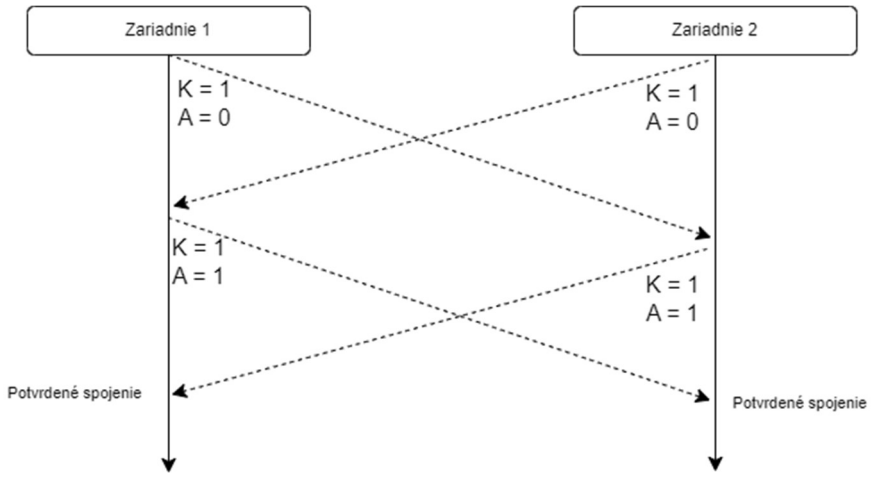

Diagram 3: Connection Maintenance

Sending Unfragmented Text

When sending a message, User 1 sends a message to User 2 with an empty **Fragment Number** and **Window Size** field. User 2 then either sends an **acknowledgment** or requests a retransmission of the message.

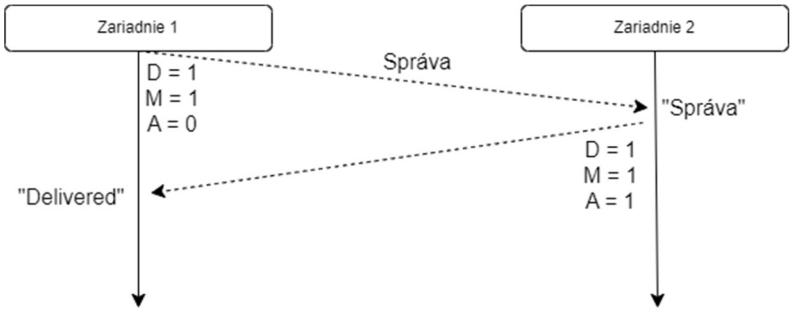

Diagram 4: Sending Unfragmented Text

Negotiating Parameters for Sending

When negotiating parameters before sending a file or fragmented text, the sender sends a message that informs the receiver of the parameters important for sending and storing the file or text. The **Fragment Number** field will contain the number of the last fragment (for unfragmented file transfer, this field will have a value of 0). The **Window Size** field will contain the maximum value for the

window. In the **Data** section, when sending a file, the file name will be specified, while for fragmented text, this field will remain empty. After receiving the parameters, the receiver sends an acknowledgment.

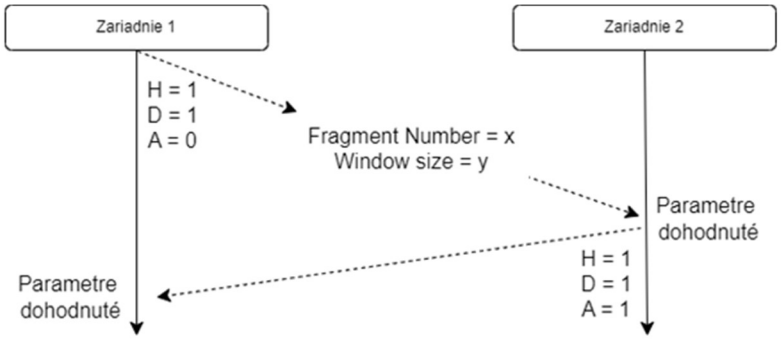

Diagram 5: Negotiating Parameters for Sending Text

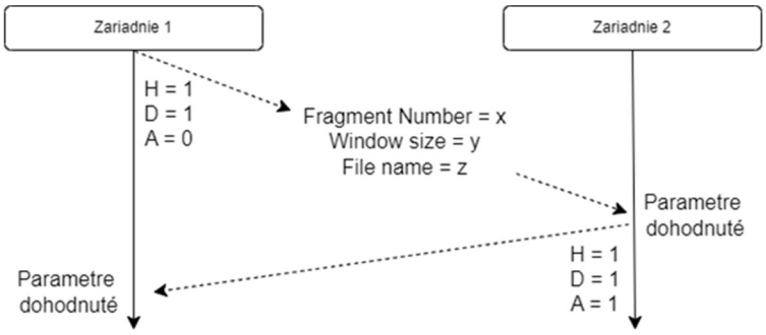

Diagram 6: Negotiating Parameters for File Transfer

Sending a File and Fragmented Text

When sending an unfragmented file, only one message with data will be sent. If the receiver successfully receives it, they will send an acknowledgment.
When sending a fragmented file or text, the sender will send as many fragments as the **Window Size** allows. In other words, they will send all fragments within the window, where the window is defined by the first fragment (the first fragment for which the sender has not received an acknowledgment) and the window size. The receiver will attempt to store the received fragments in order. After storing the fragments in memory, the receiver will send an acknowledgment for the last stored fragment. If the receiver cannot store any fragment, they will send a **request** for the fragment they are expecting to store next. Upon receiving the request, the sender will resend the fragment. After receiving the acknowledgment, the sender will move the window forward to the next fragment after the confirmed one.

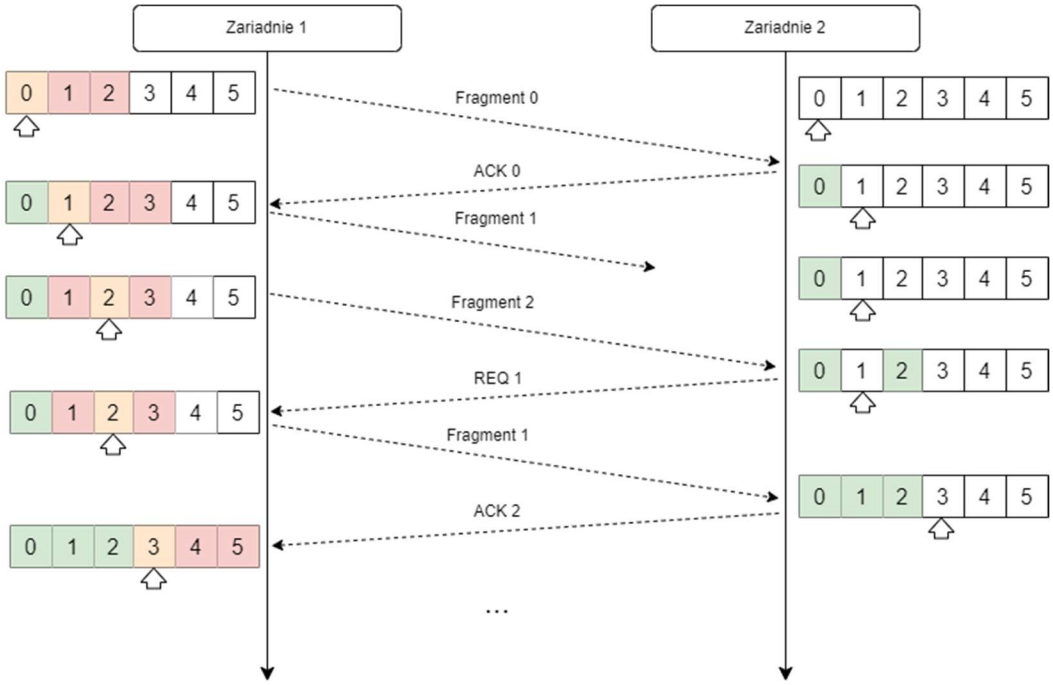

Diagram 7: Sending Fragmented Data

If a sudden connection failure occurs during data transfer, the communicators will check the connection after 5 seconds of inactivity by sending 3 **keepalive** messages, each 5 seconds apart. If the communicator receives an acknowledgment, the transfer will continue. If no acknowledgment is received for any of the 3 messages, the connection will be terminated.

Data Corruption and Loss Checking

To check for data corruption, the **Checksum** field will be used. This field will contain a value calculated from the data.

For checksum calculation, I have chosen **CRC-16-CCITT**. The checksum is computed as follows:

1. For each byte in the data, a left shift by 8 bits is performed, and a bitwise XOR operation is carried out with the current CRC value (initially set to FFFF in hexadecimal).
2. Then, we perform 8 iterations based on the value of the first bit:
    - If the first bit is 1, shift the current CRC to the left and perform the XOR operation.
    - If the first bit is 0, simply shift the CRC to the left without the XOR operation.

This process is repeated for each byte of the data.

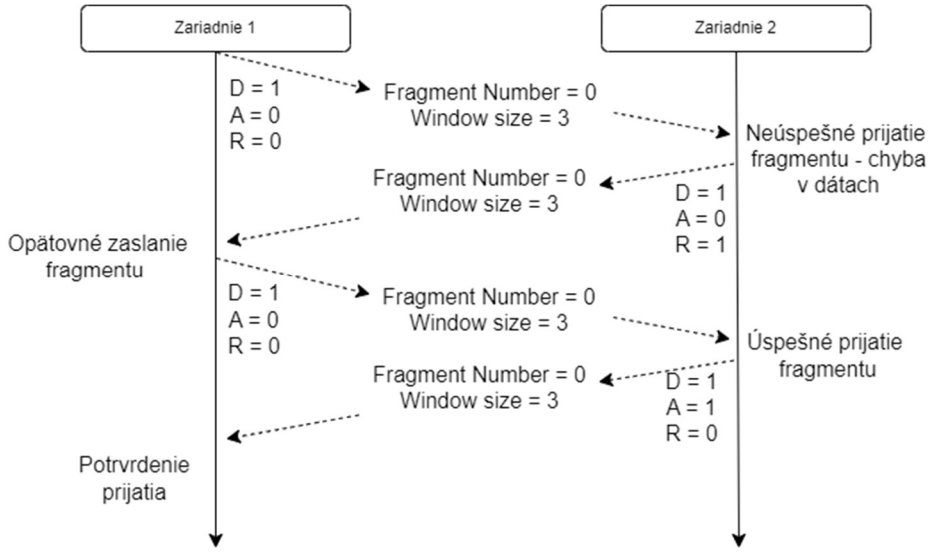

Diagram 8: Receiving a Corrupted Message

Data Corruption Simulation

When sending data, the user can set the **error rate** for data transmission, ranging from 0 to 50. After calculating the checksum of the message, one byte at the end will be modified to a different value, simulating data corruption. This will occur based on the specified error rate.

Application Description

Upon launching the application, the user is prompted to input the connection parameters, including their own IP address and port, as well as the IP address and port of the other user. After entering these parameters, a GUI window is displayed, where the user can control the communication. Before any actions can be performed, the communicator must be set up. The user needs to configure the file storage address, the size of a single fragment (the maximum size is set so that the entire message is smaller than 1500B), and the error rate for message corruption. These settings can be changed during program execution. After saving the settings, the user can attempt to connect. Upon successful connection, the user has the option to disconnect, with the possibility to reconnect after disconnection.

Once connected, the connection is monitored using **keepalive** messages. The user can send messages or files. To send a message, the user simply types it in the message window, and the message will be successfully sent. If the message is smaller than the fragment size, it will be sent as a whole. If the message is larger, it will be sent in fragments. The entire message will appear in the output window of both users. To send a file, the user enters its address, and information about the file's sending/receiving and its address will be displayed in the output window. All other important information about messages, the connection, and time is displayed in the terminal. During the program's execution, the user can see the status of their communicator in the application (whether it is connected, disconnected, or in the process of sending).

When the application is closed, all threads are terminated using a global variable, and the socket is closed.

Protocol Detection in Wireshark

To detect my protocol, we were required to create a Lua script. My script is set for two ports, 1111 and 2222. If other ports are used during transmission, Wireshark will not be able to detect it. My script identifies the message type based on flags and displays the truth values of each flag. It also prints the values for fragment number, window size, checksum, and data. Additionally, I created a coloring rule in Wireshark to highlight the messages I sent.

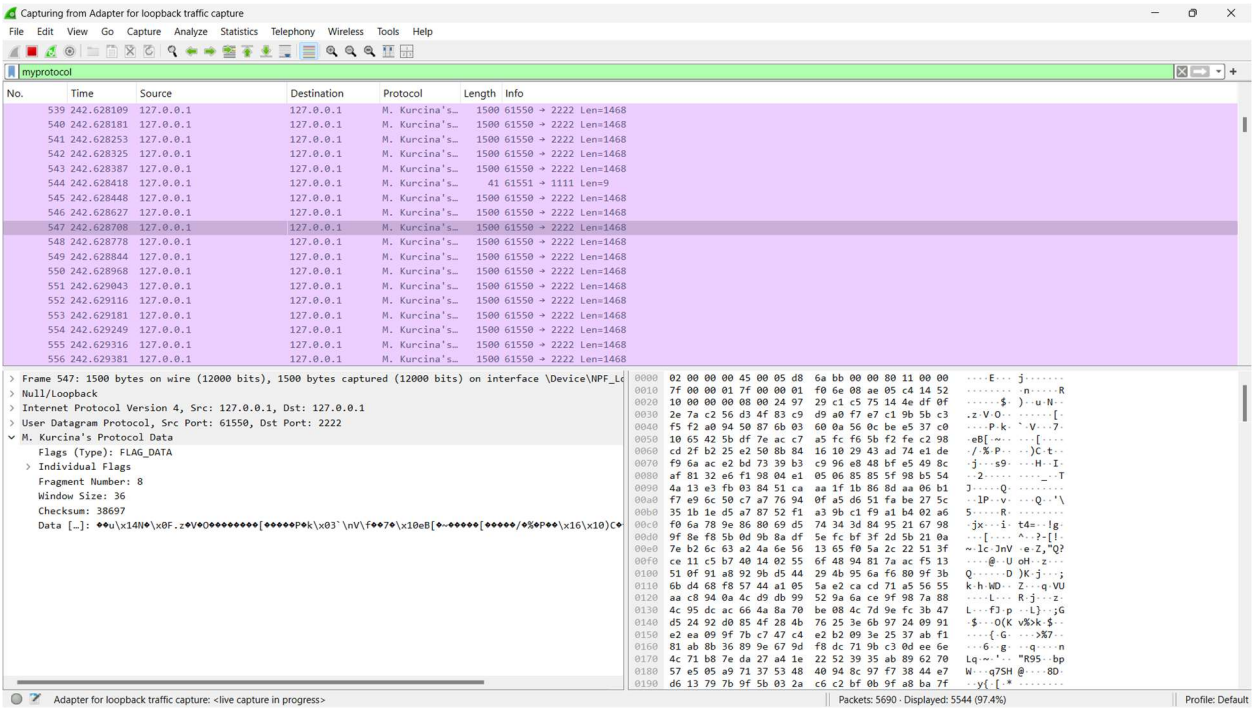

Image 1: Example of recognizing parts of my protocol using a Lua script and coloring rules.

Conclusion

My task was to design a protocol that would meet the requirements in the assignment and create an application to establish a connection and send messages using my protocol. I have successfully completed the task. My application can establish a connection, send text and files, both fragmented and non-fragmented, check the connection, and terminate the connection, all using my designed protocol. My protocol is designed to fulfill the functionalities outlined in the assignment. I have described these functionalities in my documentation, and I was able to implement the application accordingly.