Protocol over UDP
in
Computer and Communication Networks

Júda Vodrážka

Faculty of Informatics and Information Technolo gies,
Slovak University of Technology in Bratislava
xvodrazka@stuba.sk
student ID: 120909

November 25, 2024

Contents

1 Introduction

We have been tasked with creating our own protocol over UDP. The protocol is meant to be used for peer-to-peer communication. General criteria are as follows:

- The program must be implemented in one of the following programming languages: C, C++, C#, Python and Rust
- The user must be able to set their listening port
- The user must be able to set the destination IP address and port
- The user must be able to choose a maximum fragment size on their end and be able to change it dynamically while the program is running
- The end device must be able to display the following information about a sent/received file/message:
  - name and the absolute file path on said end device
  - size and number of fragments, inclusing the total size
- Simulate a failure of transmission by sending at least one invalid fragment
- The receiving end device must be able to notify the sender the correct and incorrect transmission of fragments. For incorrect transmission of a fragment, the receiver requests the corrupted data.
- Be able to send a 2MB file within 60 seconds and save it on the receiving end device as the same file
- Be able to choose the file save location

For the first control point of the assignment we needed to write the documentation describing our conceived protocol and its functionality, Additionally, an established communication over UDP, including a handshake was needed.

2 Control Point - phase 1

2.1 Protocol header structure

The protocol employs similar techniques as already existing protocols. For example the handshake works on the same basis as the TCP three-way handshake, and fragmentation is a modified version of IPv4 fragmentation.

The header contains eight individual fields:

- Sequence number (32 bits)
- Acknowledgement number (32 bits)
- Checksum (16bits)

- Identification (16bits)
- Type (4 bits)
- Reserved (4 bits)
- Flags (8 bits)
- Fragment number (16 bits)

The total size of the header is 16 Bytes, which means the total payload for Ethernet networks is in between 1416-1456 Bytes.

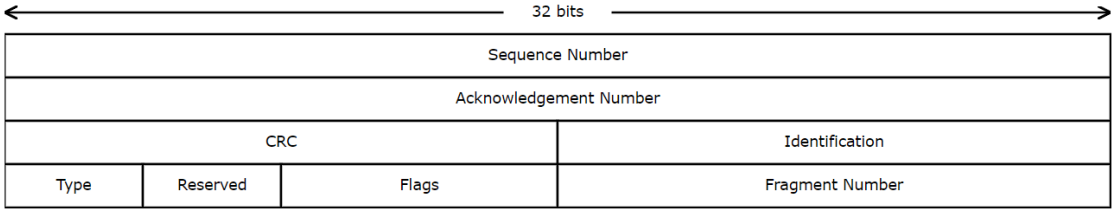

Figure 1: Protocol header

The *Sequence Number* is a numerical value representing the oridnal number of the datagram in the current connection session. The *Acknowledgement Number* represents the next byte that is expected to arrive. In other words, it is confirming the reception of data up to that byte. This technique is inspired by the TCP protocol, utilizing both Sequence and Acknowledgement numbers as well as their respective sizes in bytes.

CRC field is used for verifying data integrity. That is done using CRC16. Normally the value of CRC16 is appended to the end of the data, however, for ease of use my implementation adds it to the header.

Identification is a unique number representing each sent or received datagram. The number is mainly used for fragmentation along with the *Fragment Number* field. When there is an attempted transmission of data of size greater than 1400 bytes (default fragment size value) fragmentation occurs. Furthermore, since the user is supposed to be able to set their own fragment size within the program, fragmentation may also occur if the data is larger than the user-set fragment size value. Fragmented datagrams have the same *Identification Number* making it possible to merge them back into single data unit after reception.

Type field is a simple four-bit value, representing the type of data sent. While the task states it is necessary to be able to send messages and files, meaning the *Type* could have been represented by a flag value in the *Flags* field, it is imperative to be able to tell what type of data is getting transmitted, for post processing and encoding reasons. So far there are three types of values accepted in the *Type* field (however, there may be more added later on):

- 0 - plain text data
- 1 - .txt file
- 2 - image file

Reserved field is four bits set to zero. It serves only one purpose and that is to have the header be aligned to the 32bit standard.

Flags are bit-wise values, each representing a different attribute. Since these attributes can only be either on or off, it is pointless to dedicate an entire field to each one of them. The flags are mostly similar to the flags in the TCP header, however, there are a few alterations. The bits (from left to right) represent the following attributes:

- NACK - *Negative Acknowledge*: set to 1 if the receiver did not receive the sent datagram
- MFG - *More fragments*: set to 1 if more fragments are to be expected
- FRG - *Fragmented*: set to 1 if the data has been fragmented
- FIN - *Finnish*: used for requesting the end of connection
- SYN - *Synchronize*: used for establishing connection
- RST - *Reset*: drops the connection abruptly
- PSH - *Push*: set to 1 if data are being transmitted
- ACK - *Acknowledge*: used for acknowledging received data and for establishing connection

Lastly, *Fragment Number* field denotes the number fragment. It is obviously used solely when the data is fragmented, therefore only when the flag *FRG* has been set to 1. Using this number, the recipient can rebuilt the fragmented data in the correct order, even if they have been not been received as such.

2.2 Establishing connection

Since the communication is peer-to-peer, either one of the end devices may initiate connection. To do that, both devices should be listening on their respective ports. If a peer tries initiating a connection with a peer whose port is not open (listening), the communication times out and the connection is refused.

Establishing communication works on the same principle as in communication over TCP. Suppose there are two peers, peer A and peer B, trying to establish a connection. A three way handshake will occur. The initiator, peer A, sends an empty datagram (that is a datagram without data) to the user-set destination IP address and port with the *SYN* flag set to 1. If peer B with said IP address is listening on the port, he will reply with another empty datagram with flags *ACK* and *SYN* set to 1. Finally, peer A sends the last empty datagram with the *ACK* flag set to 1, and the connection is established.

After that both peers can communicate with each other by sending data, until the connection is terminated, either by one of the peers or by other factors (e.g. timeout, network issues, etc.)

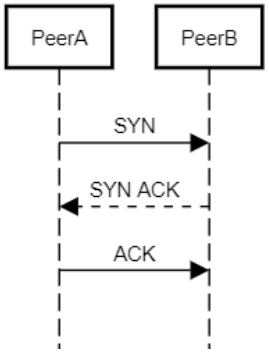

Figure 2: Three way handshake between PeerA and PeerB

2.3 CRC16

To ensure data integrity, CRC16 is used. The way that CRC16 works is that both end devices get a predefined generator polynomial, which acts as a key for ensuring the received data has not been corrupted. In the case of CRC16, the generator polynomial is 16-bit.

Before the data are sent, CRC is set to 16 zeros and appended to the data. XOR binary division is applied to the data by the generator polynomial. The remainder of said division is the generated CRC. Usually this is appended to the end of the data, however, I have decided to add it to the header for ease of use.

When receiving data, the receiving peer confirms data integrity by performing XOR binary division on the received data by the same generator polynomial. If the remainder is zero, the data has not been corrupted otherwise the datagram is considered invalid, thrown away and requested for retransmission.

2.4 ARQ - Stop & Wait

To ensure the sent data has been delivered and received, the receiver needs to send an empty datagram with the flag *ACK* set to 1 after every datagram received. Apart from sending an *ACK*, the *Sequence Number* and *Acknowledgement Number* are updated to match the data sent/received. If the sender does not receive an *ACK* with the correct *Acknowledgement Number* within 2 seconds, the sent datagram is considered lost and is subsequently retransmmitted.

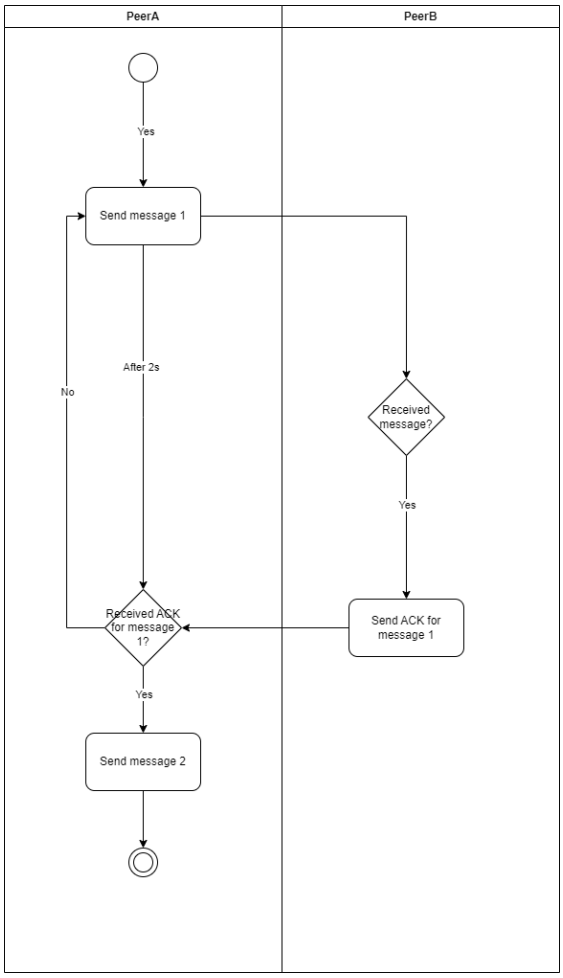

Figure 3: ARQ - Stop & Wait

2.5 Keep-Alive

After a connection has been established, it needs to be maintained. For that *Keep-Alive* is used. When a communication goes idle for too long, it may get disconnected by a firewall, or a NAT device. Therefore, *Keep-Alive* functionality sends an empty datagram witho the *ACK* flag set to 1 after a certain period of time (the current hypothesized length is 5 seconds, though it may be subject to change).

Furthermore, *Keep-Alive* helps detect when the connection has timed out. If the other end device does not respond to three *Keep-Alive* probes, the connection is considered dead and is subsequently terminated.

2.6 Implementation

My implementation has been done in Python using the `socket` library. The code is attached as a separate file called `communication-over-udp.py`, Once ran, a GUI window pops up where the user can set necessary values, such as the listening port, destination IP address and destination port. After a successful connection, the two peers can send messages, which will be displayed in the center of the window.

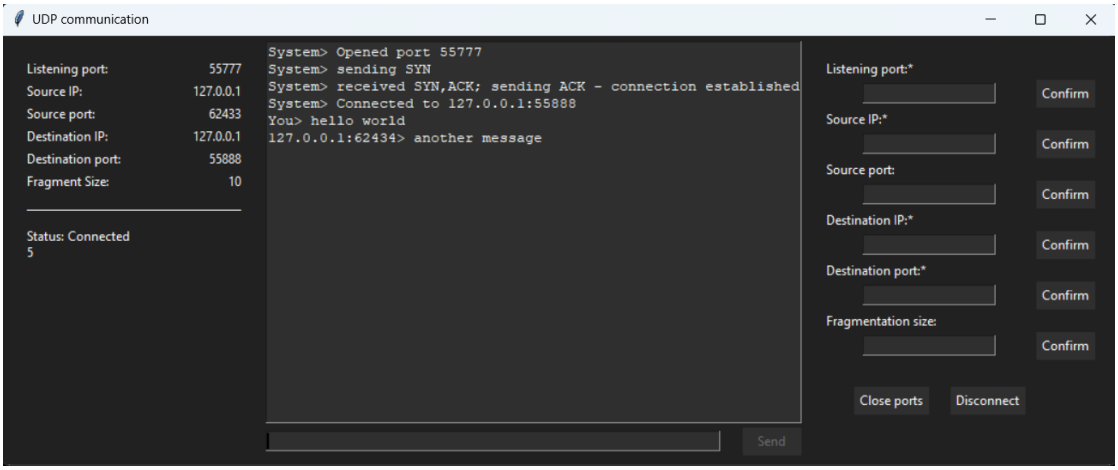

Figure 4: GUI window

3 Final submission - phase 2

3.1 Changes

The final implementation of the protocol differs slightly from the original design. First and foremost the protocol header no longer includes a type field and contains a few more flags.

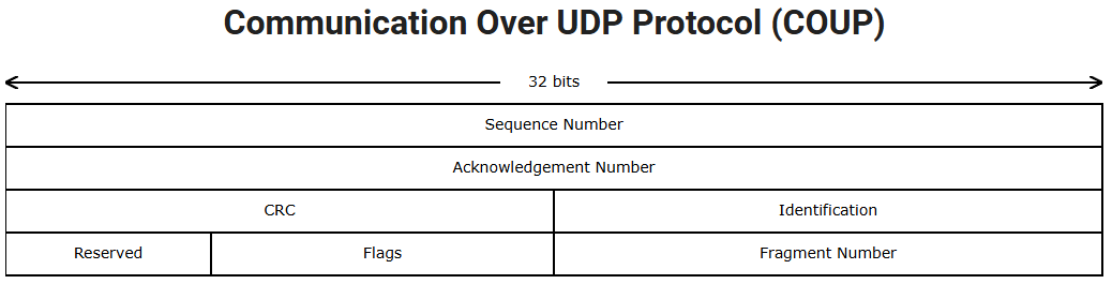

Figure 5: Final design of COUP

As is shown in figure 5, the `Flags` field is now of length 10 bits and the `Reserved` is 6 bits long. Two new flags have been added - `KEA` and flag `FIL`. Every five seconds a keep alive request is send from either of the peers. Upon receiving such request a keep alive reply is sent. These are denoted by the flag `KEA` in conjunction with flags `SYN` and `ACK` respectively. Flag `FIL` is always used with flag `PSH` and denotes that the transmitted data are a file instead of plain text. This lets the corresponding method know not to decode it directly, rather extract the name and write the data into a file instead.

3.1.1 ARQ - selective repeat

Another change has been made to the automatic repeat request method. In the end I have opted for selective repeat ARQ instead of stop-and-wait, since the increase in speed (and points earned) is significant.

Selective repeat works by sending a batch of packets concurrently. The amount of packets sent at once is determined by a so called sliding window. The receiving end will send an acknowledge for each packet it receives and a NAK for each packet that fails the CRC16 check. When the sender receives ACK for the first packet in the sliding window, it slides the window to up until the first unacknowledged packet.

This ensures that even when one packet fails to get acknowledged (either the acknowledgment or the packet itself gets lost) the other ones will be acknowledged and do not need to be retransmitted once more.

If an acknowledgment for a sent packet is not received within certain time period (set by `ack_timeout`, set to 0.5 seconds) it gets retransmitted and the timer resets.

The receiver knows if all packets are received when the number of received (and acknowledged) packets is equal to the fragment number of the last packet + 1 (the addition of 1 being necessary, since the fragment count starts at 0).

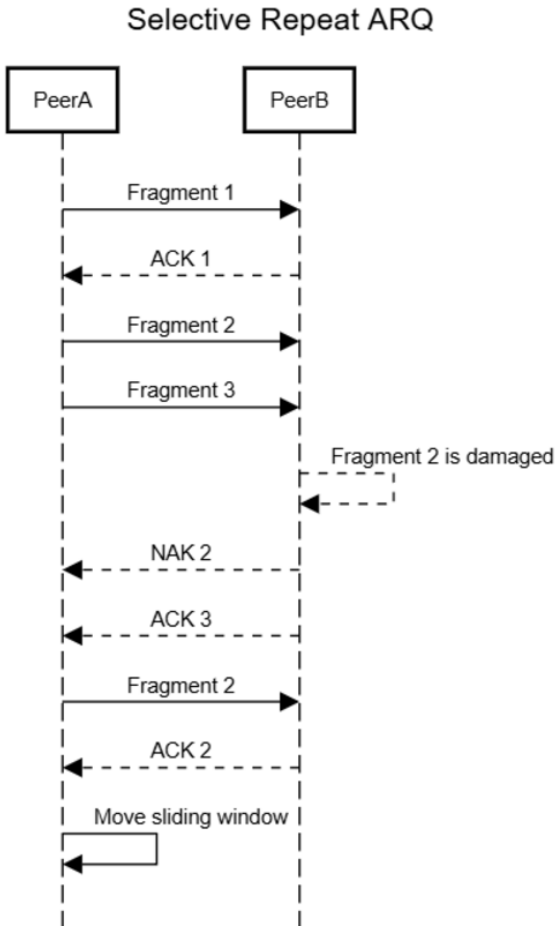

Figure 6: ARQ - selective repeat

3.1.2 File transmission

Additionally, a user is now able to send a file instead of just sending a message. A file can be either sent by manually typing `\sendfile [absolute path]` into the message input field. Another way a file can be transmitted is to click on "Select file" button located in the control section of the graphical user interface. A window will open which lets the user select any file they want to send. Once a file is selected, the program automatically writes out the `\sendfile [absolute path]` and all the user need to do is press "Send".

One more change has been made in terms of downloading. In this version of the application, the user can also choose where they want their received files to be downloaded. The default folder is the user's Downloads folder, however, this can be easily changed by clicking on the "Download folder" button and selecting a new download location.

3.1.3 Fault simulation

Furthermore, a fault simulation has been implemented. To send a malformed packet, the user types `\fault` followed by the message they are trying to transmit. It should also be noted, that the fault simulation works with file transmission. The user can send a malformed packet by typing `\fault` before `\sendfile [absolute path]`.

If the sent data will not get fragmented, then a random length of the data will be changed for the faulty transmission. This ensures, that the CRC16 value will not match the one calculated for the original message.

On the other hand, if the data will get fragmented, a random number of fragments is chosen and each fragment has a random length of the payload "corrupted". This is supposed to simulate a real-life-like experience of packet corruption.

3.2 Complex showcase

Figure 7 shows a complex showcase of a communication between two endpoints. The data sent are color coded in 4 colors. Light-green packets are system packets without any payload, dark-green packets are those that contain payload, whether it be a plain message or a file, red packets signify `NAK` flag, or in other words not acknowledge and lastly light-blue packets are keep alive.

The communication starts with three light-green packets representing the initial handshake, also shown in 2. They are followed by a three instances of keep alive heartbeat messages. There are six packets in total meaning, because the connection is established and each keep alive request has its own keep alive response.

Next 4 packets were two messages. Since both were sent without simulating an occurrance of fault, they are followed by a light-green system packet (ACK). After that there are another two instances of heartbeat.

Following that, is a packet sent with a fault and immediately after is a red NAK. The NAK is answered by a corrected packet, which is later acknowledged.

Lastly there are 2 packets of keep alive request WITH keep alive response. After these two packets, I have closed the connection on one peer. Therefore, the other peer sent three more keep alive requests, but since there was no keep alive response acquired within the time limit no more messages have been sent through. This indicates the end of the connection.

Figure 8 shows the graphical user interface of the two peers communicating in figure 7.

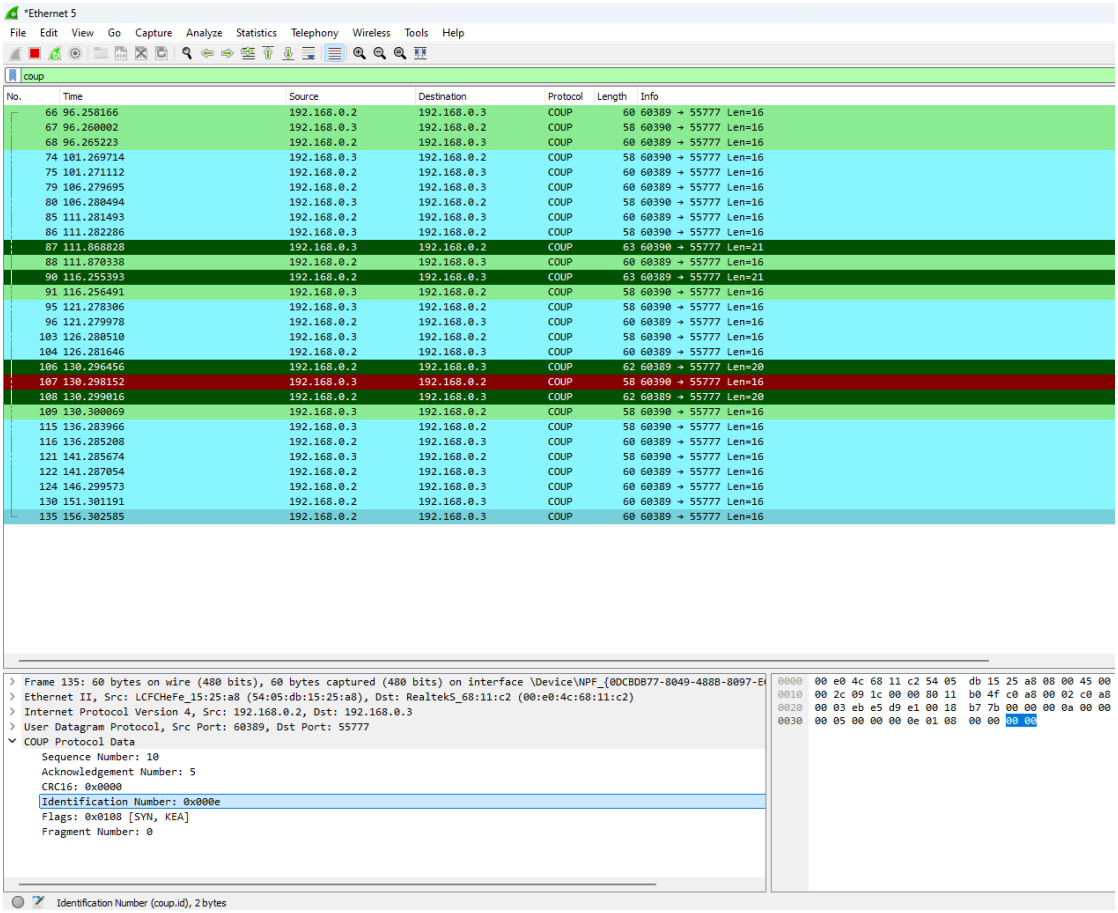

Figure 7: Wireshark capture

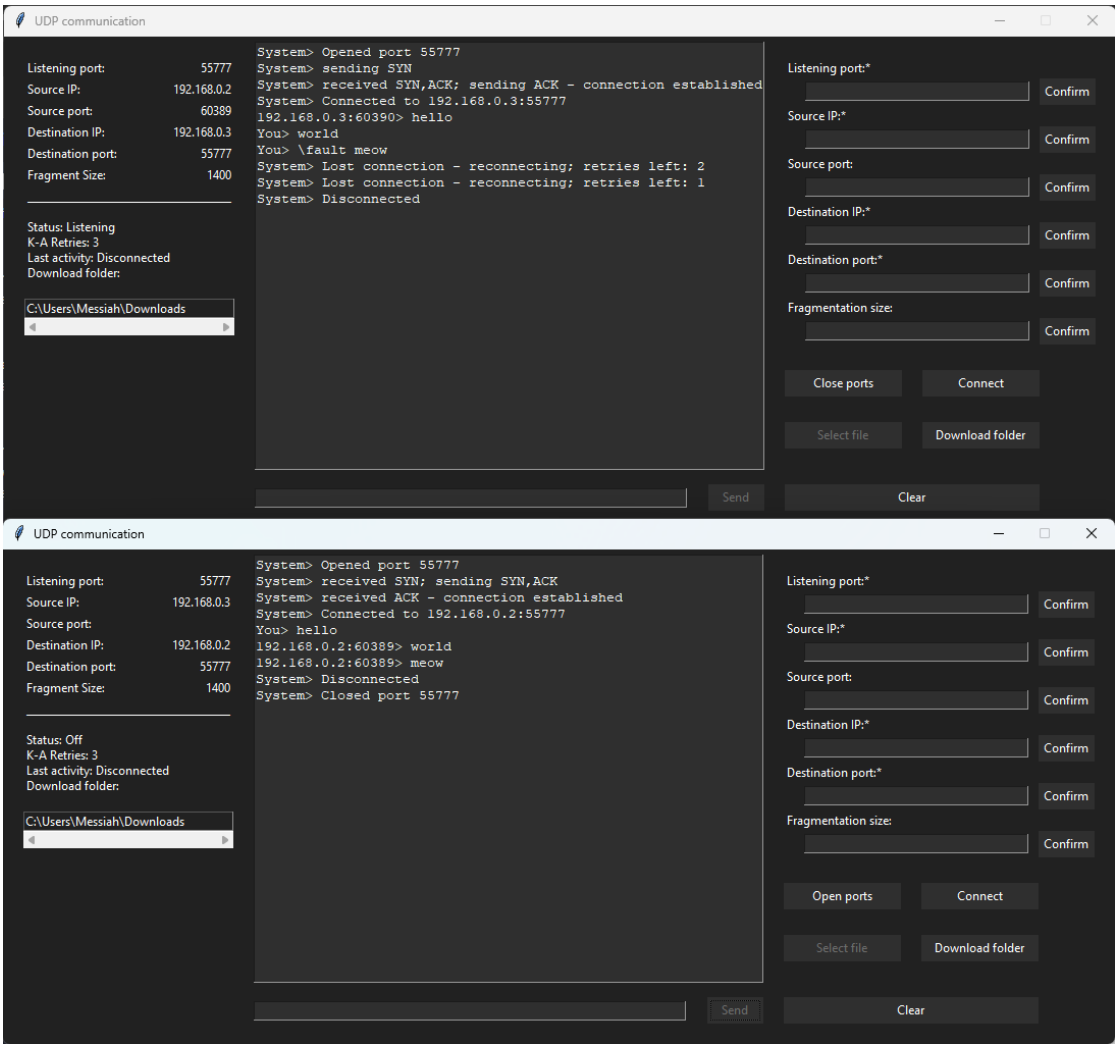

Figure 8: Communication between two peers

4 Conclusion

I have successfully achieved a stable communication over UDP through my protocol.