Slovak University of Technology in Bratislava

Faculty of Informatics and Information Technologies

Communication application using the UDP protocol ( Implementation)

Computer and Communication Networks — Assignment #2

Oleksandr Oksanich

Contents

[Implementation 1](#_Toc152633465)

[Changes 1](#_Toc152633466)

[Protocol Header 1](#_Toc152633467)

[Fragment Number 2](#_Toc152633468)

[Message Type 2](#_Toc152633469)

[Header Checksum 3](#_Toc152633470)

[Maintaining the Connection 3](#_Toc152633471)

[Checksum Methods 3](#_Toc152633472)

[A step-by-step example 3](#_Toc152633473)

[ARQ 4](#_Toc152633474)

[Communication Processing Diagram 5](#_Toc152633475)

[Main Python Modules 7](#_Toc152633476)

[Server Implementation 7](#_Toc152633477)

[Client Implementation 8](#_Toc152633478)

[Switch Implementation 9](#_Toc152633479)

[User Interface 10](#_Toc152633480)

[Examples 10](#_Toc152633481)

[Wireshark Testing 16](#_Toc152633482)

# Implementation

## Changes

1. The “Length” field has been removed from the header
2. The “Message Type” field’s length has been reduced to 4 bits (instead of 8 bits)
3. A few more messages types have been introduced:
   1. FRAGMENT\_COUNT
   2. FILE\_PATH
   3. ACK\_AND\_SWITCH
   4. TEXT
4. All the messages types have been reordered by their binary value
5. Stop and Wait is now used as the ARQ method instead of Go-Back-N
6. The server is no longer supposed to send “TIMEOUT” messages to the client
7. The server is no longer supposed to send “FIN” messages to the client when it wants to terminate the connection
8. The server is no longer supposed to send “SWITCH\_NODES” messages to the client because it now asks for a switch while sending the last data fragment

## Protocol Header

|  |  |
| --- | --- |
| **Field** | **Length (bits)** |
| Fragment Number | 24 |
| Message Type | 4 |
| Header Checksum | 32 |
| *Reserved* | 4 |

### Fragment Number

The number of the current fragment in a sequence of fragments. Its length is 3 bytes because being able to transfer a 2 MB file requires at least 2 million fragments which can be accommodated at least by a 3-byte number.

### Message Type

|  |  |  |
| --- | --- | --- |
| **Type** | **Binary Value** | **Meaning** |
| DATA | 0000 | Data transfer |
| ACK | 0001 | Message acknowledgement |
| NACK | 0010 | Negative acknowledgement (mostly used as a request for retransmission of specific packets) |
| INIT | 0011 | Connection establishment signal |
| FIN | 0100 | Connection termination signal |
| KEEP\_ALIVE | 0101 | Persistent connection signal |
| TIMEOUT | 0110 | Persistent connection timeout signal |
| CHANGE\_MAX\_FRAGMENT\_SIZE | 0111 | Request to change the fragment size limit (1–1464 due to Ethernet II limitations) |
| FRAGMENT\_COUNT | 1000 | Total upcoming data fragment count |
| FILE\_PATH | 1001 | Source file path message |
| SWITCH\_NODES | 1010 | Client request to switch the roles |
| ACK\_AND\_SWITCH | 1011 | Server request to switch the roles (once the last data fragment is acknowledged) |
| TEXT | 1100 | Text message |

### Header Checksum

The field is described in the [“Checksum Methods”](#_Checksum_Methods) section.

## Maintaining the Connection

In order to maintain the connection, I use a separate socket running on its own thread on the sending side which occasionally (every 10 seconds) sends a message of the “KEEP\_ALIVE” type to the server. The server must acknowledge the message by replying to it; otherwise, the client terminates the connection after some time while sending the “TIMEOUT” message.

The keep-alive thread is paused while any data fragment or any text message is being transferred.

## Checksum Methods

I am going to use the function “crc32” from the Python library “zlib”. The length of its return value is 32 bits (i.e., 4 bytes), therefore the corresponding header field has the same length. The checksum is calculated based on all the other header fields (fragment number and message type) and the data itself.

I decided to go with CRC-32 instead of CRC-16, CRC-8, and so on, since computational overhead is not a significant concern whereas error detection and collision prevention are stronger with CRC-32.

The “crc32” function uses the following polynomial:

(BIN: 0b00000100110000010001110110110111, HEX: 0x4c11db7)

For convenience in describing an algorithm that provides the same output as zlib.crc32, I have used the reversed representation of the polynomial (i.e., 0b11101101101110001000001100100000, 0xedb88320) so that the input data would not have to be reversed. The algorithm is as follows:

1. Set the initial CRC value to 0xffffffff
2. Iterate each byte of the input data and XOR it with the current CRC value
3. While iterating the bytes, iterate each bit of the current byte, apply a right shift by 1 bit to the CRC value and XOR it with the polynomial if the last bit of CRC before the shift is 1
4. Once the byte loop is over, XOR the final CRC value with 0xffffffff

### A step-by-step example

|  |  |
| --- | --- |
| Input | 0b11010011 |
| CRC | 0b11111111111111111111111111111111 |
| XOR with the 1st byte | 0b11111111111111111111111100101100 |
| 1st bit shift | 0b1111111111111111111111110010110 |
| 2nd bit shift | 0b111111111111111111111111001011 |
| 3rd bit shift | 0b11111111111111111111111100101 |
| XOR with the polynomial | 0b11110010010001110111110011000101 |
| 4th bit shift | 0b1111001001000111011111001100010 |
| XOR with the polynomial | 0b10010100100110110011110101000010 |
| 5th bit shift | 0b1001010010011011001111010100001 |
| 6th bit shift | 0b100101001001101100111101010000 |
| XOR with the polynomial | 0b11001000100111100100110001110000 |
| 7th bit shift | 0b1100100010011110010011000111000 |
| 8th bit shift | 0b110010001001111001001100011100 |
| XOR with 0xffffffff | 3453512931 |

## ARQ

I intend to use the Stop and Wait ARQ method as the error-control method for data transmission for this protocol. The sender transmits a single data frame to the receiver and waits for an acknowledgment before sending the next frame. Upon receiving a frame, the receiver sends back an ACK to confirm successful reception. If the frame is corrupted or lost, the receiver sends a negative acknowledgment or does not respond respectively. As illustrated by the diagram below, data corruption and data losses cause sequential retransmission (next fragments are only sent after the last lost fragment is re-sent and acknowledged).

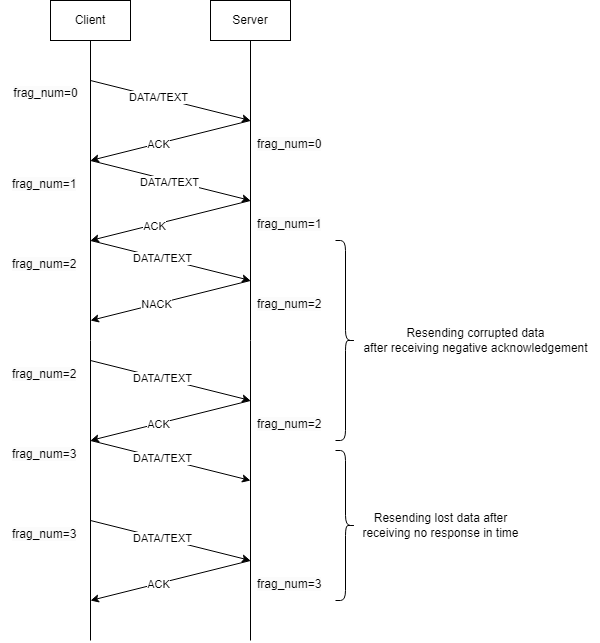


Figure 1

## Communication Processing Diagram

The sequence diagram below demonstrated all the possible communication options between the two nodes taking into account all the available message types and respective user scenarios.

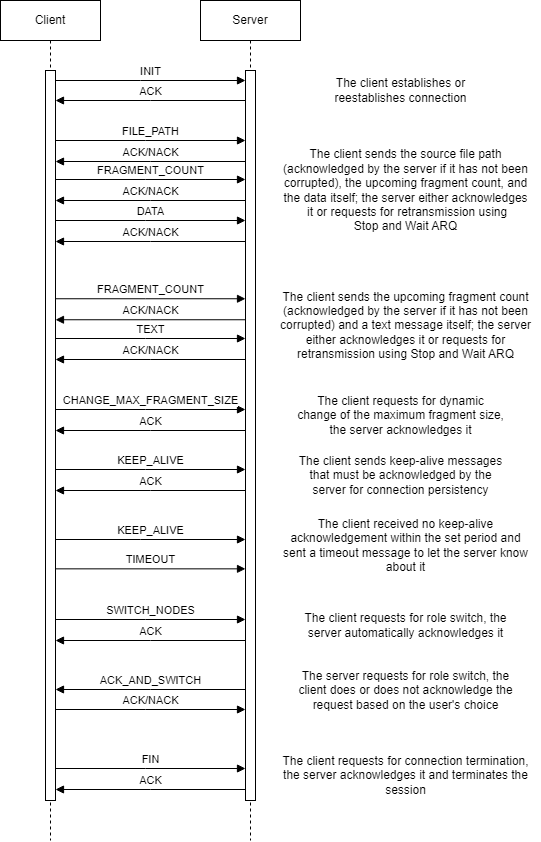


Figure 2

## Main Python Modules

* socket — a standard Python module that provides an interface for establishing communication between different sockets
* zlib — a built-in Python module that is often used for file compression and decompression as well as checksum calculation (what it is used for in this project)
* threading — a built-in module that provides high-level functionality for creating and working with threads (used for sending keep-alive messages)
* ipaddress — a built-in module used for input IP address validation in this project
* time — a standard module used for connection timeouts in the project
* bitarray — a third-party module used for operating data bit by bit (required for message serialization and deserialization)

## Server Implementation

When launched, the program in the server mode asks the user to set the IP address and the port. If both are correct, the server socket will go up and start listening to incoming messages (namely the connection initialization request and keep-alive messages). The rest of the options are illustrated by the diagram below (the full-sized version can be found in the project directory).

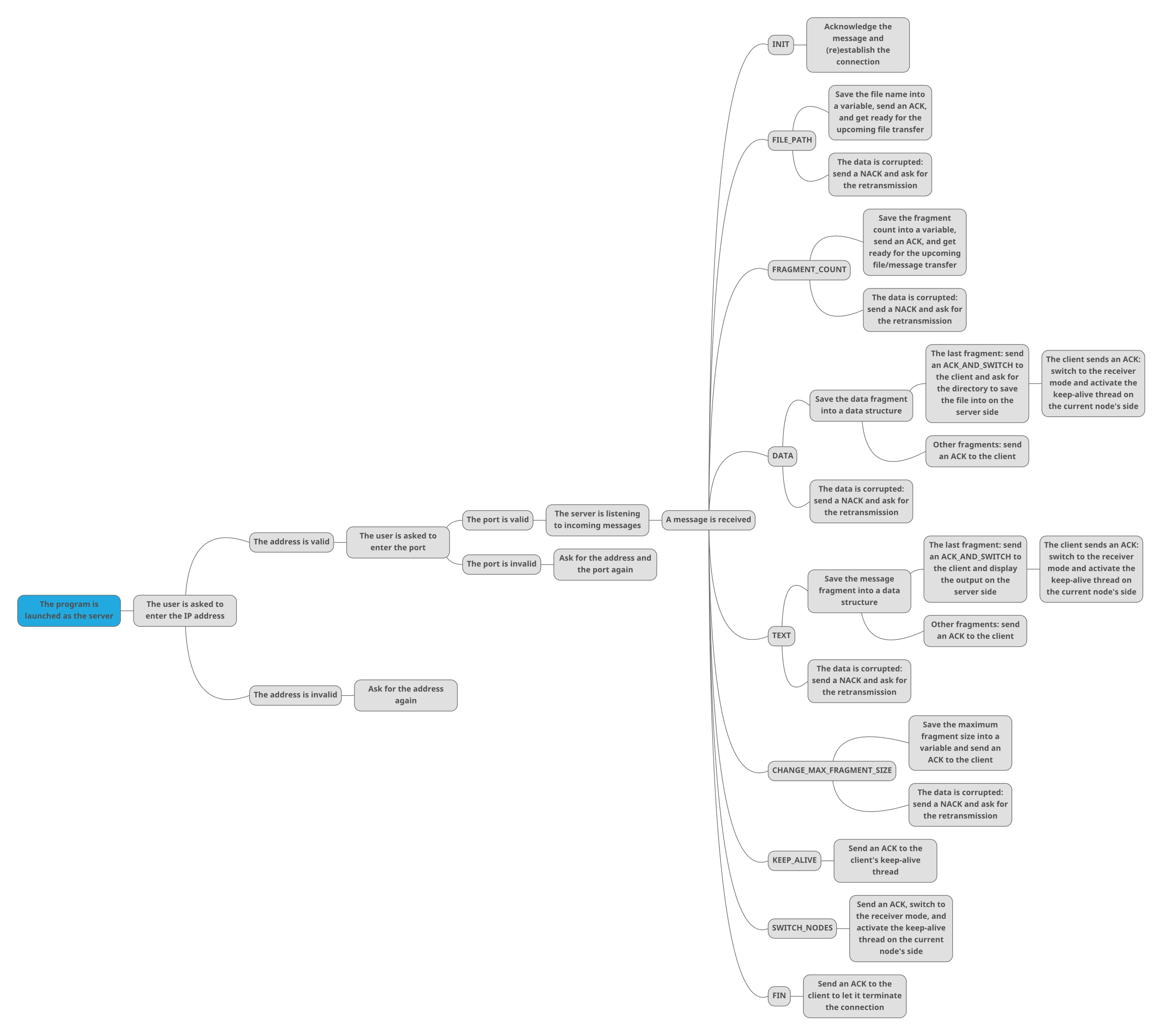


Figure 3

## Client Implementation

After the server has been activated, the client node can be launched as well. The user must enter the IP address and the port of the running server node. Once they do, the client sends an INIT message to the server and waits for an ACK while also activating its keep-alive thread. As soon as the connection is up, the user is given access to the client menu. The rest of the options are illustrated by the diagram below (the full-sized version can be found in the project directory).

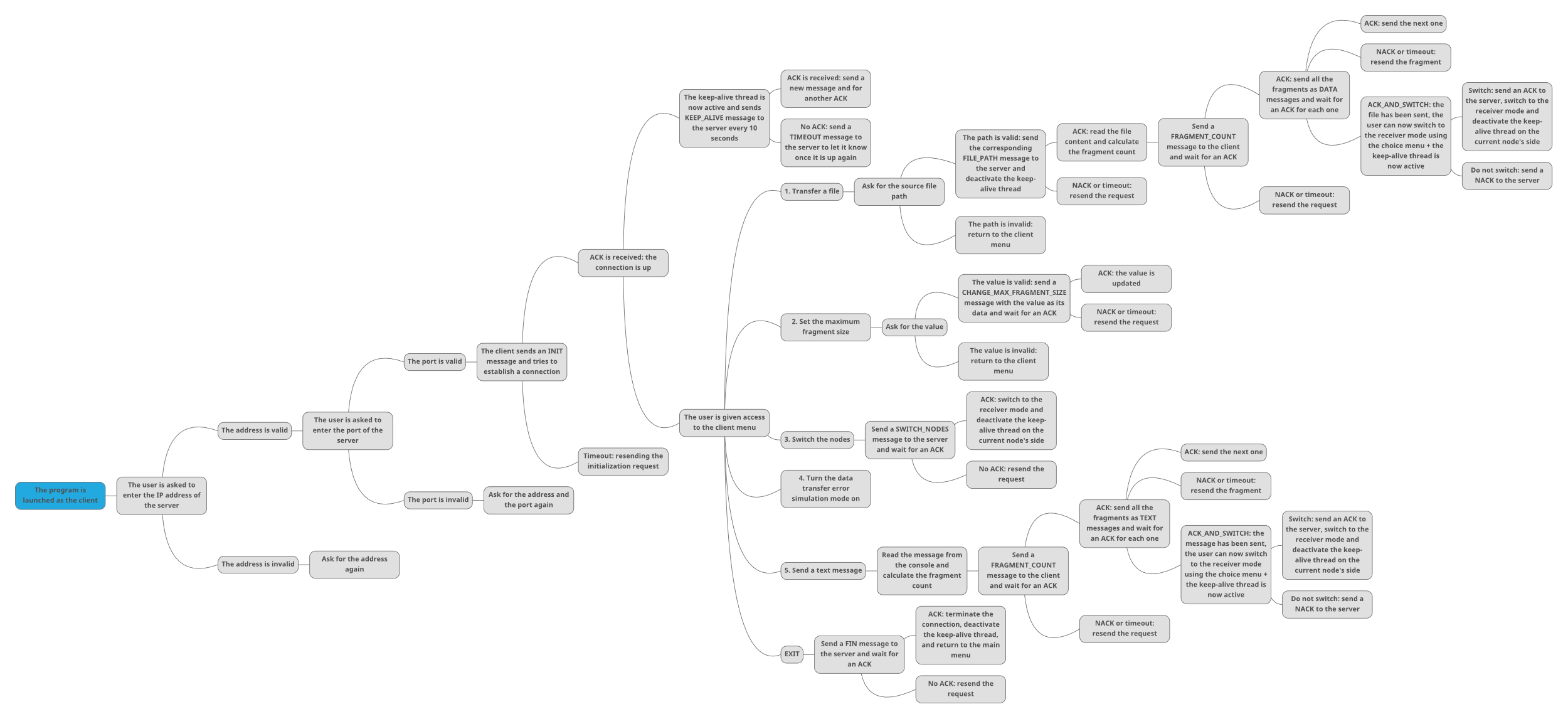


Figure 4

## Switch Implementation

In order to demonstrate how my switch implementation works, I would like to use a snippet of Python-like pseudocode. Basically, the code for the two modes is incorporated into both nodes; therefore, what mode a particular node is currently in is managed by certain flags (specifying whether a node should be running in the opposite mode at the moment).

*# client.py*client\_socket = ...  
server\_mode = False  
  
def client():  
 if !server\_mode:  
 while !server\_mode:  
 send\_messages(client\_socket)  
  
 if switch:  
 server\_mode = True  
 break  
 if server\_mode:  
 client()  
 else:  
 while server\_mode:  
 listen\_to\_messages(client\_socket)  
  
 if switch:  
 server\_mode = False  
 break  
 if !server\_mode:  
 client()

*# server.py*server\_socket = ...  
client\_mode = False  
  
def server():  
 if !client\_mode:  
 while !client\_mode:  
 listen\_to\_messages(server\_socket)  
  
 if switch:  
 client\_mode = True  
 break  
 if client\_mode:  
 server()  
 else:  
 while client\_mode:  
 send\_messages(server\_socket)  
  
 if switch:  
 client\_mode = False  
 break  
 if !client\_mode:  
 server()

## User Interface

The entire project has been implemented as a console program where the node currently on the sending side is constantly waiting for user input.

### Examples

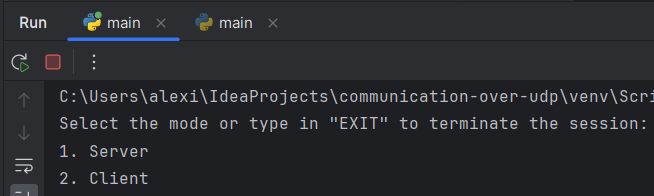


Figure 5. Start menu

A computer screen with white text and numbers

Description automatically generated

Figure 6. Server init

A screen shot of a computer

Description automatically generated

Figure 7. Client init

A screen shot of a computer error

Description automatically generated

Figure 8. Client menu

A screen shot of a computer

Description automatically generated

Figure 9. File transfer (client side)

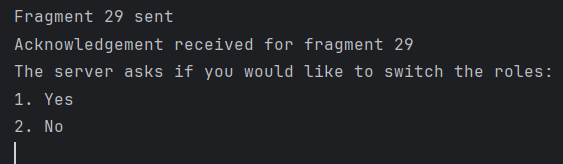


Figure 10. File transfer + switch request confirmation (client side)

A screenshot of a computer screen

Description automatically generated

Figure 11. File transfer (server side)

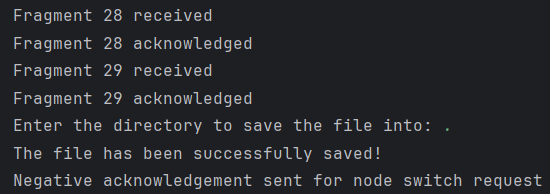


Figure 12. File transfer and saving + switch request response received (server side)

A screen shot of a computer screen

Description automatically generated

Figure 13. Keep-alive thread (server side)

A screenshot of a computer program

Description automatically generated

Figure 14. Maximum fragment size request (client side)

A black background with white text

Description automatically generated

Figure 15. Maximum fragment size request (server side)

A screenshot of a computer program

Description automatically generated

Figure 16. Node switch request (client side)

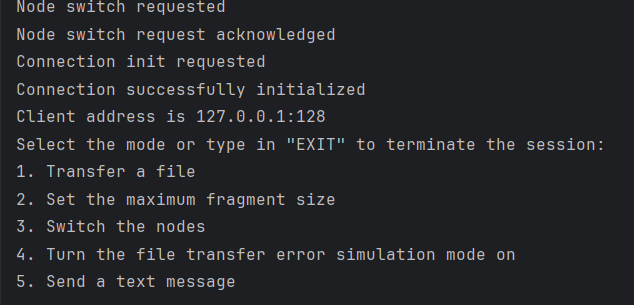


Figure 17. Node switch request (server side)

A screen shot of a computer

Description automatically generated

Figure 18. Data transfer error simulation mode (client side)

A screenshot of a computer screen

Description automatically generated

Figure 19. Data transfer error simulation mode (server side)

A screenshot of a computer program

Description automatically generated

Figure 20. Connection termination (client side)

A black background with white text

Description automatically generated

Figure 21. Connection termination by the client (server side)

## Wireshark Testing

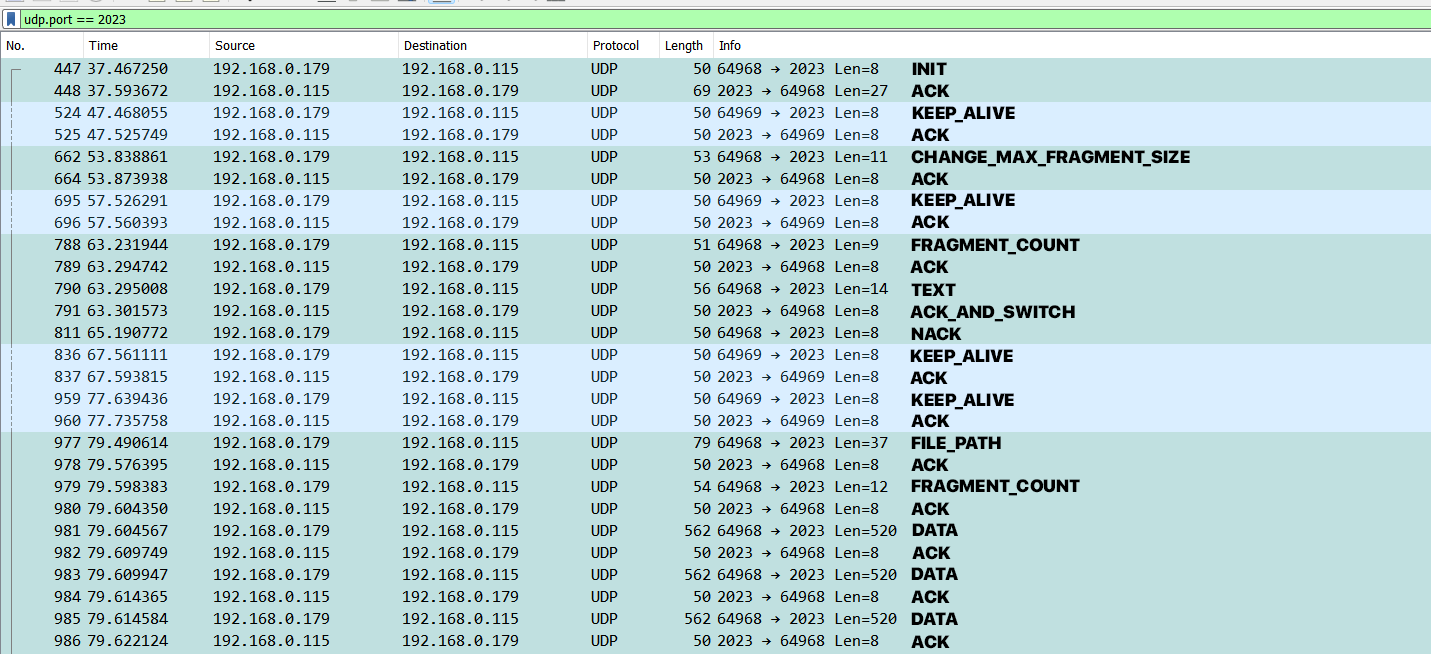


Figure 22. Communication #1

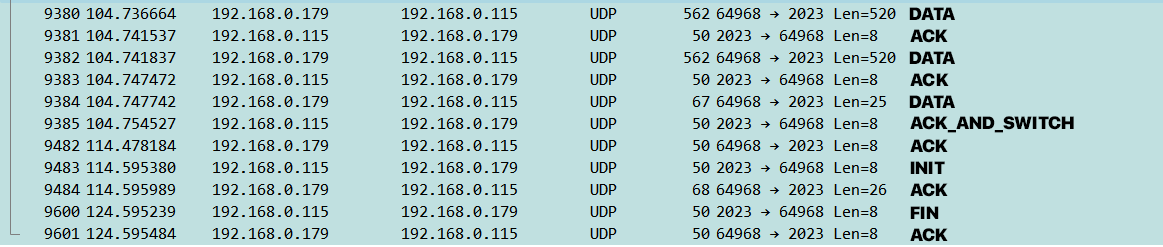


Figure 23. Communication #1

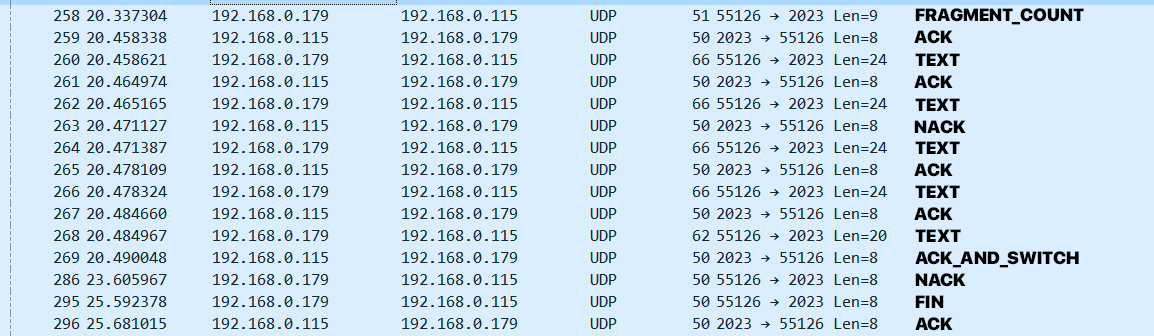


Figure 24. Communication #2 (data transfer error simulation mode on)