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

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:
   3.1. FRAGMENT_COUNT
   3.2. FILE_PATH
   3.3. ACK_AND_SWITCH
   3.4. 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" 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:

$$x\text{^}32 + x\text{^}26 + x\text{^}23 + x\text{^}22 + x\text{^}16 + x\text{^}12 + x\text{^}11 + x\text{^}10 + x\text{^}8 + x\text{^}7 + x\text{^}5 + x\text{^}4 + x\text{^}2 + x + 1$$

(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	0b11111111111111111111111110010110
2nd bit shift	0b11111111111111111111111111001011
3rd bit shift	0b11111111111111111111111100101
XOR with the polynomial	0b11110010010001110111110011000101
4th bit shift	0b11110010010001110111111001100010
XOR with the polynomial	0b10010100100110110011110101000010
5th bit shift	0b10010100100110110011110101000001

6th bit shift	0b10010100100110110011110101000
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).

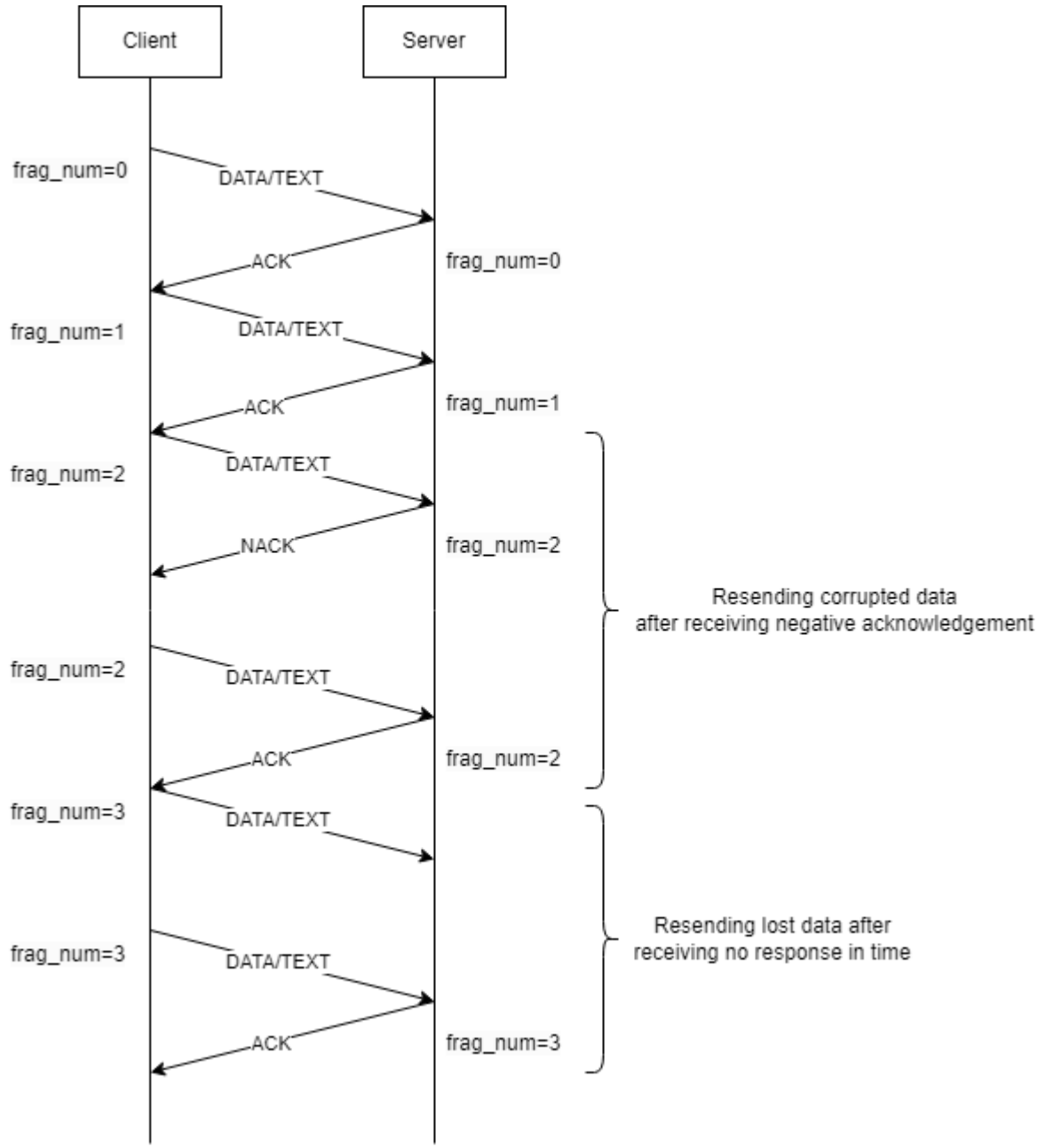

Figure 1

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.

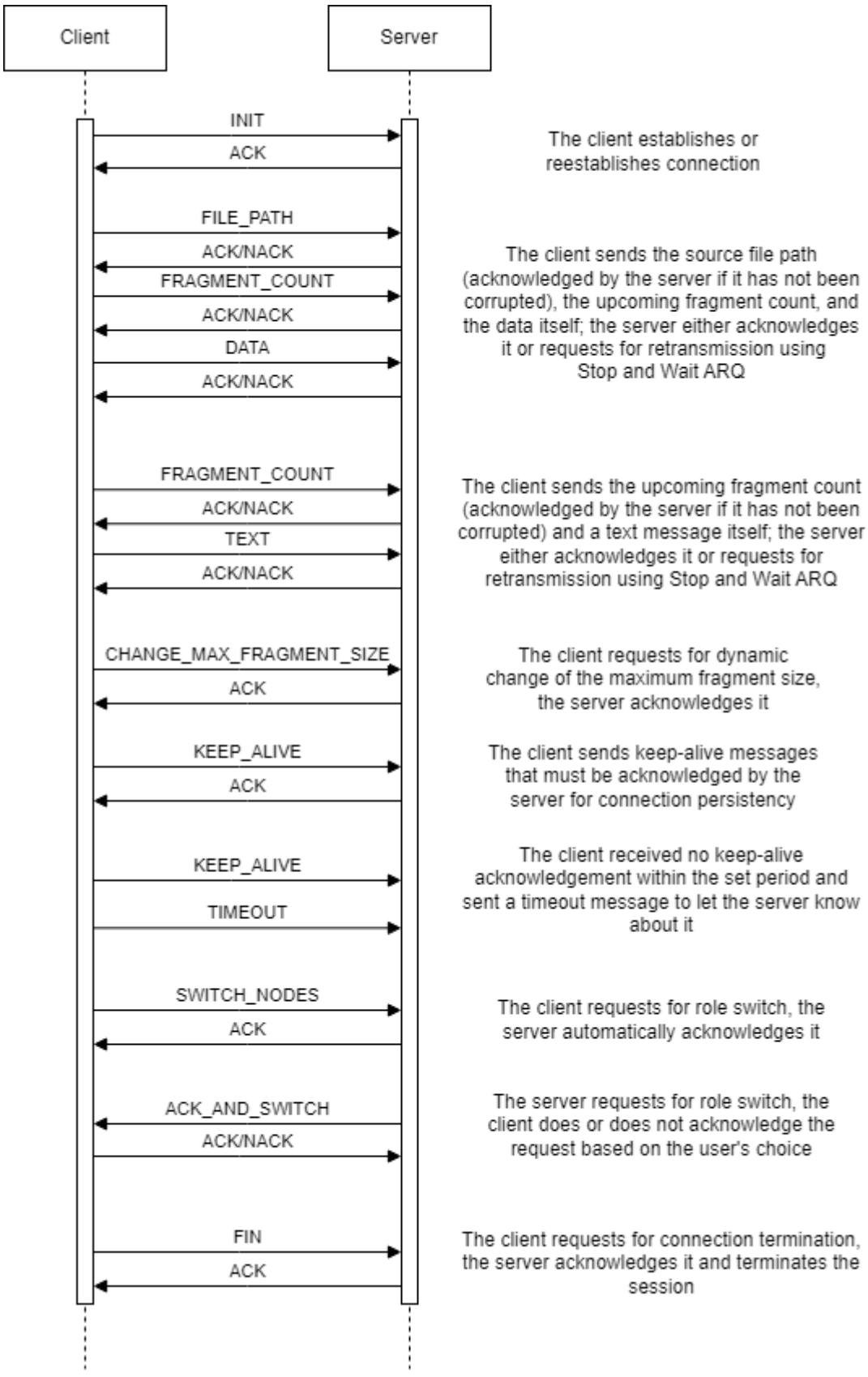
Client
Server
INIT
ACK
The client establishes or reestablishes connection
FILE_PATH
ACK/NACK
FRAGMENT_COUNT
ACK/NACK
DATA
ACK/NACK
The client sends the source file path (acknowledged by the server if it has not been corrupted), the upcoming fragment count, and the data itself; the server either acknowledges it or requests for retransmission using Stop and Wait ARQ
FRAGMENT_COUNT
ACK/NACK
TEXT
ACK/NACK
The client sends the upcoming fragment count (acknowledged by the server if it has not been corrupted) and a text message itself; the server either acknowledges it or requests for retransmission using Stop and Wait ARQ
CHANGE_MAX_FRAGMENT_SIZE
ACK
The client requests for dynamic change of the maximum fragment size, the server acknowledges it
KEEP_ALIVE
ACK
The client sends keep-alive messages that must be acknowledged by the server for connection persistency
KEEP_ALIVE
TIMEOUT
The client received no keep-alive acknowledgement within the set period and sent a timeout message to let the server know about it
SWITCH_NODES
ACK
The client requests for role switch, the server automatically acknowledges it
ACK_AND_SWITCH
ACK/NACK
The server requests for role switch, the client does or does not acknowledge the request based on the user's choice
FIN
ACK
The client requests for connection termination, the server acknowledges it and terminates the session

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).

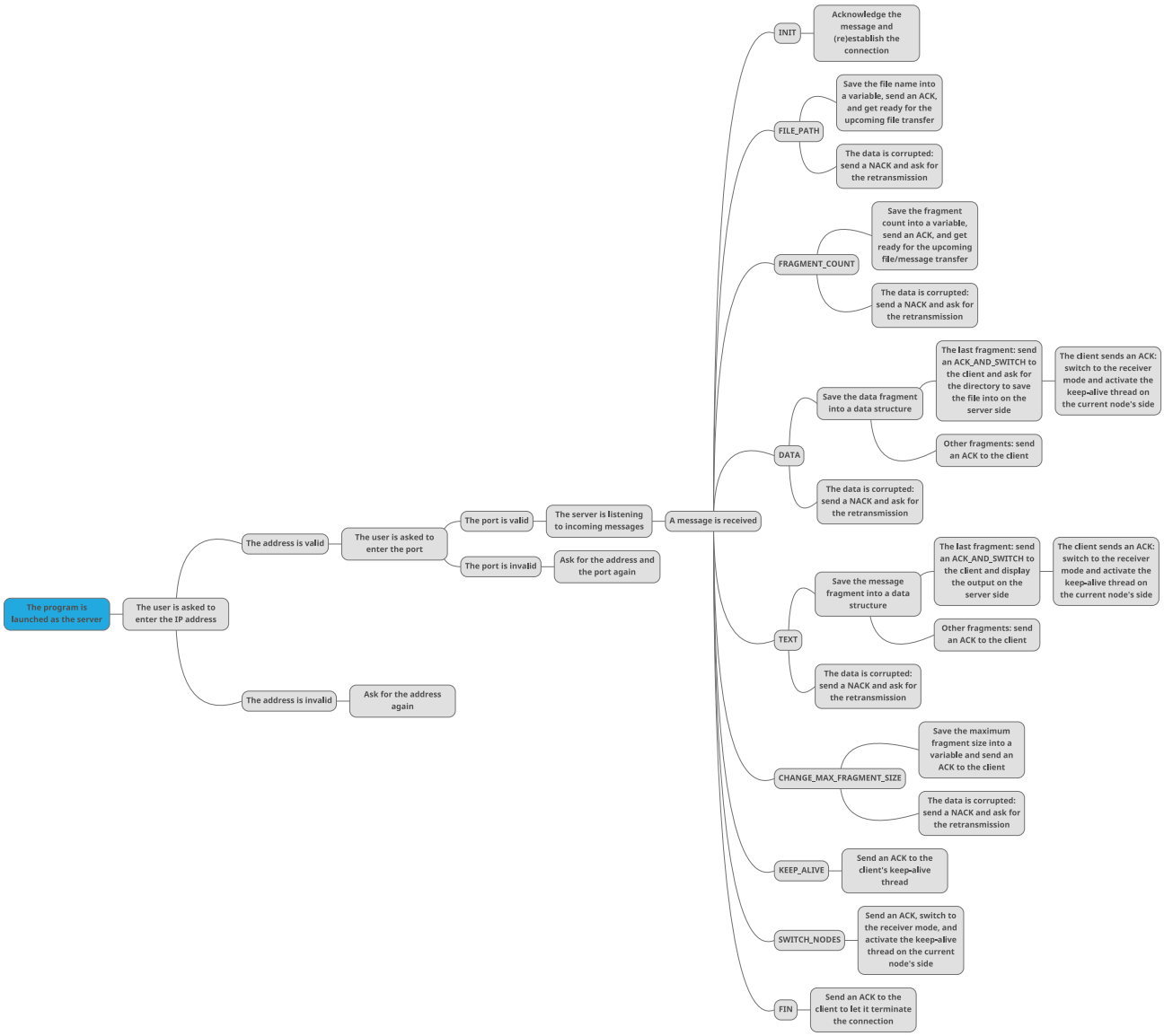

Figure 3

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).

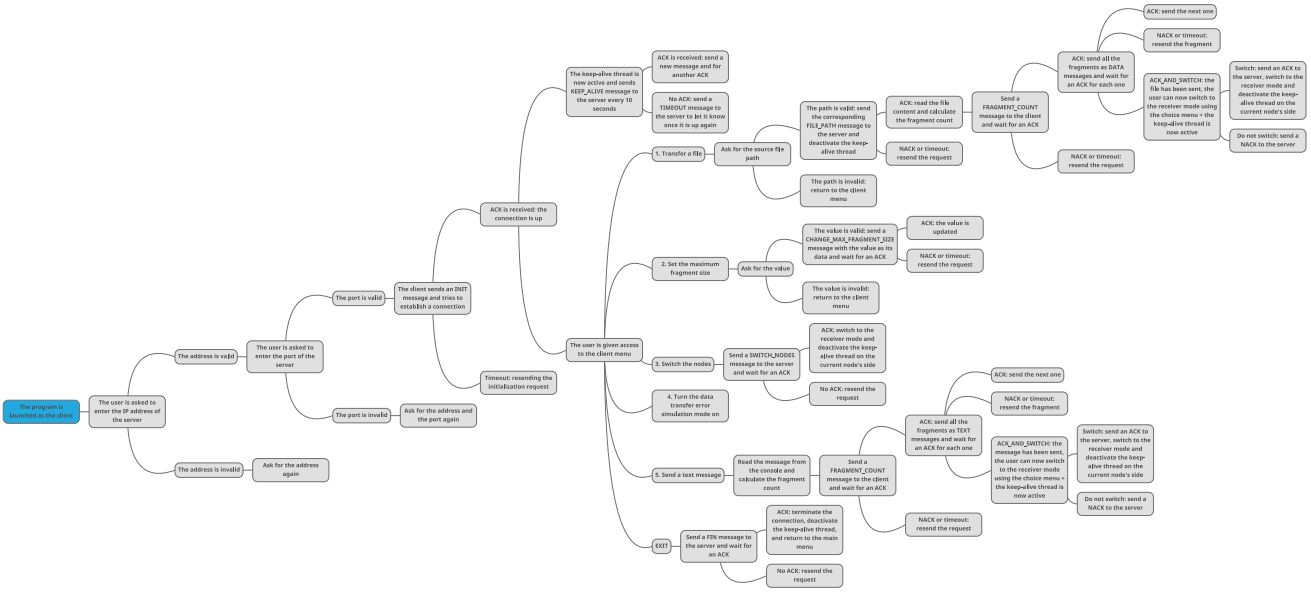

Figure 4

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

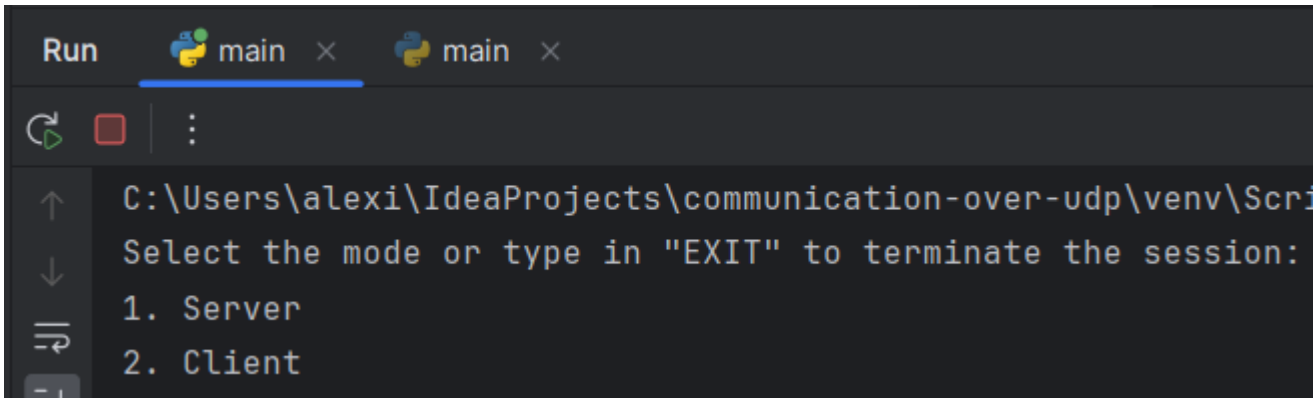

Figure 5. Start menu

```
Select the mode or type in "EXIT" to terminate the session:
1. Server
2. Client
1
Enter the server IP address: 127.0.0.1
Enter the server port: 128
Server address is 127.0.0.1:128
```

Figure 6. Server init

```
Select the mode or type in "EXIT" to terminate the session:
1. Server
2. Client
2
Enter the server IP address: 127.0.0.1
Enter the server port: 128
Connection init requested
Connection successfully initialized
Client address is 127.0.0.1:52460
```

Figure 7. Client init

```
Select the mode or type in "EXIT" to terminate the session:
1. Transfer a file
2. Set the maximum fragment size
3. Switch the nodes
4. Turn the data transfer error simulation mode on
5. Send a text message
```

Figure 8. Client menu

```
1
Enter your file path: t2.txt
File path sent: t2.txt
File path sent successfully
Fragment count (30) sent
Fragment count successfully received
Fragment 0 sent
Acknowledgement received for fragment 0
Fragment 1 sent
Acknowledgement received for fragment 1
Fragment 2 sent
Acknowledgement received for fragment 2
Fragment 3 sent
```

Figure 9. File transfer (client side)

```
Fragment 29 sent
Acknowledgement received for fragment 29
The server asks if you would like to switch the roles:
1. Yes
2. No
```

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

```
File path received: t2.txt
File path acknowledged
Fragment count received: 30
Fragment count acknowledged
Fragment 0 received
Fragment 0 acknowledged
Fragment 1 received
Fragment 1 acknowledged
Fragment 2 received
Fragment 2 acknowledged
```

Figure 11. File transfer (server side)

```
Fragment 28 received
Fragment 28 acknowledged
Fragment 29 received
Fragment 29 acknowledged
Enter the directory to save the file into: .
The file has been successfully saved!
Negative acknowledgement sent for node switch request
```

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

```
Keep-alive message received
Keep-alive message acknowledged
Keep-alive message received
Keep-alive message acknowledged
Keep-alive message received
Keep-alive message acknowledged
Keep-alive message received
Keep-alive message acknowledged
```

Figure 13. Keep-alive thread (server side)

```
2. Set the maximum fragment size
3. Switch the nodes
4. Turn the file transfer error simulation mode on
5. Send a text message
2
Enter your size in bytes: 128
Maximum fragment size (128) sent
Fragment size changed successfully
```

Figure 14. Maximum fragment size request (client side)

```
Max fragment size received: 128
Max fragment size acknowledged
```

Figure 15. Maximum fragment size request (server side)

```
Select the mode or type in "EXIT" to terminate the session:
1. Transfer a file
2. Set the maximum fragment size
3. Switch the nodes
4. Turn the file transfer error simulation mode on
5. Send a text message
3
Node switch requested
Starting the switch
Server address is 127.0.0.1:52460
Connection init received from 127.0.0.1:128
Connection init acknowledged
Keep-alive message received
Keep-alive message acknowledged
```

Figure 16. Node switch request (client side)

```
Node switch requested
Node switch request acknowledged
Connection init requested
Connection successfully initialized
Client address is 127.0.0.1:128
Select the mode or type in "EXIT" to terminate the session:
1. Transfer a file
2. Set the maximum fragment size
3. Switch the nodes
4. Turn the file transfer error simulation mode on
5. Send a text message
```

Figure 17. Node switch request (server side)

```
Enter your message: testesmsklflekmelfsmfskmlkmlfsekmfe
Fragment count (3) sent
Fragment count successfully received
Acknowledgement received for fragment 0
The message was not received correctly. Resending...
Acknowledgement received for fragment 1
Acknowledgement received for fragment 2
```

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

```
Fragment count received: 3
Fragment count acknowledged
Fragment 0 received: testesmsklflekmelfsmfskmlkmlfsekmfeslfskmelsfemtestesmsklflekmel
Fragment 0 acknowledged
Fragment 1 was not received correctly
Negative acknowledgement sent for 1
Fragment 1 received: fsmfskmlkmlfsekmfeslfskmelsfemtestesmsklflekmelfsmfskmlkmlfsekmf
Fragment 1 acknowledged
Fragment 2 received: eslfskmelsfem
Fragment 2 acknowledged
b'testesmsklflekmelfsmfskmlkmlfsekmfeslfskmelsfemtestesmsklflekmelfsmfskmlkmlfsekmfesl
```

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

```
Select the mode or type in "EXIT" to terminate the session:
1. Transfer a file
2. Set the maximum fragment size
3. Switch the nodes
4. Turn the file transfer error simulation mode on
5. Send a text message
exit
Connection termination requested
Connection successfully terminated
Select the mode or type in "EXIT" to terminate the session:
1. Server
2. Client
```

Figure 20. Connection termination (client side)

```
Termination request received
Termination request acknowledged
```

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

Wireshark Testing

udp.port == 2023

No.	Time	Source	Destination	Protocol	Length	Info
447	37.467250	192.168.0.179	192.168.0.115	UDP	50	64968 → 2023 Len=8 **INIT**
448	37.593672	192.168.0.115	192.168.0.179	UDP	69	2023 → 64968 Len=27 **ACK**
524	47.468055	192.168.0.179	192.168.0.115	UDP	50	64969 → 2023 Len=8 **KEEP_ALIVE**
525	47.525749	192.168.0.115	192.168.0.179	UDP	50	2023 → 64969 Len=8 **ACK**
662	53.838861	192.168.0.179	192.168.0.115	UDP	53	64968 → 2023 Len=11 **CHANGE_MAX_FRAGMENT_SIZE**
664	53.873938	192.168.0.115	192.168.0.179	UDP	50	2023 → 64968 Len=8 **ACK**
695	57.526291	192.168.0.179	192.168.0.115	UDP	50	64969 → 2023 Len=8 **KEEP_ALIVE**
696	57.560393	192.168.0.115	192.168.0.179	UDP	50	2023 → 64969 Len=8 **ACK**
788	63.231944	192.168.0.179	192.168.0.115	UDP	51	64968 → 2023 Len=9 **FRAGMENT_COUNT**
789	63.294742	192.168.0.115	192.168.0.179	UDP	50	2023 → 64968 Len=8 **ACK**
790	63.295008	192.168.0.179	192.168.0.115	UDP	56	64968 → 2023 Len=14 **TEXT**
791	63.301573	192.168.0.115	192.168.0.179	UDP	50	2023 → 64968 Len=8 **ACK_AND_SWITCH**
811	65.190772	192.168.0.179	192.168.0.115	UDP	50	64968 → 2023 Len=8 **NACK**
836	67.561111	192.168.0.179	192.168.0.115	UDP	50	64969 → 2023 Len=8 **KEEP_ALIVE**
837	67.593815	192.168.0.115	192.168.0.179	UDP	50	2023 → 64969 Len=8 **ACK**
959	77.639436	192.168.0.179	192.168.0.115	UDP	50	64969 → 2023 Len=8 **KEEP_ALIVE**
960	77.735758	192.168.0.115	192.168.0.179	UDP	50	2023 → 64969 Len=8 **ACK**
977	79.490614	192.168.0.179	192.168.0.115	UDP	79	64968 → 2023 Len=37 **FILE_PATH**
978	79.576395	192.168.0.115	192.168.0.179	UDP	50	2023 → 64968 Len=8 **ACK**
979	79.598383	192.168.0.179	192.168.0.115	UDP	54	64968 → 2023 Len=12 **FRAGMENT_COUNT**
980	79.604350	192.168.0.115	192.168.0.179	UDP	50	2023 → 64968 Len=8 **ACK**
981	79.604567	192.168.0.179	192.168.0.115	UDP	562	64968 → 2023 Len=520 **DATA**
982	79.609749	192.168.0.115	192.168.0.179	UDP	50	2023 → 64968 Len=8 **ACK**
983	79.609947	192.168.0.179	192.168.0.115	UDP	562	64968 → 2023 Len=520 **DATA**
984	79.614365	192.168.0.115	192.168.0.179	UDP	50	2023 → 64968 Len=8 **ACK**
985	79.614584	192.168.0.179	192.168.0.115	UDP	562	64968 → 2023 Len=520 **DATA**
986	79.622124	192.168.0.115	192.168.0.179	UDP	50	2023 → 64968 Len=8 **ACK**

Figure 22. Communication #1

9380	104.736664	192.168.0.179	192.168.0.115	UDP	562	64968 → 2023 Len=520 **DATA**
9381	104.741537	192.168.0.115	192.168.0.179	UDP	50	2023 → 64968 Len=8 **ACK**
9382	104.741837	192.168.0.179	192.168.0.115	UDP	562	64968 → 2023 Len=520 **DATA**
9383	104.747472	192.168.0.115	192.168.0.179	UDP	50	2023 → 64968 Len=8 **ACK**
9384	104.747742	192.168.0.179	192.168.0.115	UDP	67	64968 → 2023 Len=25 **DATA**
9385	104.754527	192.168.0.115	192.168.0.179	UDP	50	2023 → 64968 Len=8 **ACK_AND_SWITCH**
9482	114.478184	192.168.0.179	192.168.0.115	UDP	50	64968 → 2023 Len=8 **ACK**
9483	114.595380	192.168.0.115	192.168.0.179	UDP	50	2023 → 64968 Len=8 **INIT**
9484	114.595989	192.168.0.179	192.168.0.115	UDP	68	64968 → 2023 Len=26 **ACK**
9600	124.595239	192.168.0.115	192.168.0.179	UDP	50	2023 → 64968 Len=8 **FIN**
9601	124.595484	192.168.0.179	192.168.0.115	UDP	50	64968 → 2023 Len=8 **ACK**

Figure 23. Communication #1

258	20.337304	192.168.0.179	192.168.0.115	UDP	51	55126 → 2023 Len=9 **FRAGMENT_COUNT**
259	20.458338	192.168.0.115	192.168.0.179	UDP	50	2023 → 55126 Len=8 **ACK**
260	20.458621	192.168.0.179	192.168.0.115	UDP	66	55126 → 2023 Len=24 **TEXT**
261	20.464974	192.168.0.115	192.168.0.179	UDP	50	2023 → 55126 Len=8 **ACK**
262	20.465165	192.168.0.179	192.168.0.115	UDP	66	55126 → 2023 Len=24 **TEXT**
263	20.471127	192.168.0.115	192.168.0.179	UDP	50	2023 → 55126 Len=8 **NACK**
264	20.471387	192.168.0.179	192.168.0.115	UDP	66	55126 → 2023 Len=24 **TEXT**
265	20.478109	192.168.0.115	192.168.0.179	UDP	50	2023 → 55126 Len=8 **ACK**
266	20.478324	192.168.0.179	192.168.0.115	UDP	66	55126 → 2023 Len=24 **TEXT**
267	20.484660	192.168.0.115	192.168.0.179	UDP	50	2023 → 55126 Len=8 **ACK**
268	20.484967	192.168.0.179	192.168.0.115	UDP	62	55126 → 2023 Len=20 **TEXT**
269	20.490048	192.168.0.115	192.168.0.179	UDP	50	2023 → 55126 Len=8 **ACK_AND_SWITCH**
286	23.605967	192.168.0.179	192.168.0.115	UDP	50	55126 → 2023 Len=8 **NACK**
295	25.592378	192.168.0.179	192.168.0.115	UDP	50	55126 → 2023 Len=8 **FIN**
296	25.681015	192.168.0.115	192.168.0.179	UDP	50	2023 → 55126 Len=8 **ACK**

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