Communication over UDP

Contant

Assignment

Design and implement an application using a custom protocol over the User Datagram Protocol (UDP) of the transport layer of the TCP/IP network model. The application will enable communication between two participants in a local Ethernet network, i.e., the transfer of text messages and any file between computers (nodes).

The application consists of two parts: transmitting and receiving. The sending node sends the file to another node in the network. It is assumed that there is data loss in the network. If the sent file is larger than the user-defined maximum fragment size, the sending party breaks the file into smaller parts - fragments, which it sends separately. The user must be able to set the maximum fragment size so that they are not fragmented again on the lower layer.

If the file is sent as a sequence of fragments, the destination node reports information about the message and whether the message was transferred without errors. When the entire file is received on the destination node, it will display a message about its reception and the absolute path where the received file was saved.

The application must include communication error checking and re-requesting of erroneous fragments, including both positive and negative acknowledgement. After starting the program, the communicator automatically sends a packet to maintain the connection every 5 seconds until the user ends the connection. We recommend solving via self-defined signalling messages and separate thread.

The application must fulfil the following functions (minimum):

1. The application must be implemented in C/C++ or Python using libraries for working with UDP socket, compilable, and executable in classrooms. We recommend using the Python socket module, C/C++ libraries sys/socket.h for Linux/BSD and winsock2.h for Windows. Other libraries and functions to work with sockets must be approved by the trainee. Libraries for working with IP addresses and ports can also be used in the application: arpa/inet.h a netinet/in.h.
2. The application must work with data optimally (e.g., do not store IP addresses in 4x int).
3. When sending a file, it must allow the user to specify the destination IP and port.
4. The user (just on the transmitter side) must be able to choose the max fragment size and change it dynamically during the program run before sending message/file (no for keep-alive packets).
5. Both communicating parties must be able to display:
   a) the name and absolute path to the file on the given node,
   b) the size and number of fragments, including the total size of the message/file.
6. Possibility of simulating a transmission error by sending at least 1 erroneous fragment during file transfer (an error is purposefully inserted into the data part of the fragment, that is, the receiving party detects an error during transmission).
7. The receiving party must be able to notify the sender of the correct and incorrect delivery of fragments. If a fragment is delivered incorrectly, the application will ask to re-send the damaged data.
8. The possibility to send a 2MB file and in that case save them on the receiving side as the same file, while the user only enters the path to the directory where it should be saved.
9. The application must be organized so that both communicating nodes can switch between transmitter and receiver functions without restarting the application (one side sends a switching message, receives an ACK from the other side, and the nodes automatically switch); the application does not have to (but can) be a transmitter and receiver at the same time. Submitted by: Proposal of a solution Demonstration of the solution in accordance with the presented proposal When presenting the solution, the ability to implement a simple functionality in the exercise is a condition of the assessment.

Design of the program and communication protocol

1. Header structure

Flags (1B)	Sequence number of the fragment(4B)	Fragment size(4B)	Checksum(4B)	Data (1459B)

The header has following components:

1. Flags – represents the type or purpose of the message and has these possible values:

   1 – start connection
   2 – end of connection
   3 – fragment received
   4 – fragment rejected
   5 – keep alive
   6 – sending file
   7 – sending text
   8 – file name
   9 – connection confirmation

2. Sequence number – represents the sequence number of the fragment.
3. Fragment size – indicates the size of the message and fragment being sent.
4. Checksum method – calculated using CRC32 algorithm. Used for error checking and ensure data integrity.
5. Data – represents all data being sent. The max size of the data fragment is 1459B and was calculated as 1500B – IP header – UDP header – my header = 1500B – 20B – 8B – (1+4+4+4)B = 1459B.

2. Description of the used checksum method and ARQ operation

1. Description of checksum method

This program's checksum technique uses zlib CRC32 algorithm to guarantee data integrity while it's being transmitted. The algorithm provides an effective way to detect errors by computing a 32-bit checksum for a given data collection. When data is transferred, a checksum is appended to the header, which enables the recipient to confirm data integrity by recalculating the checksum after receiving the data.

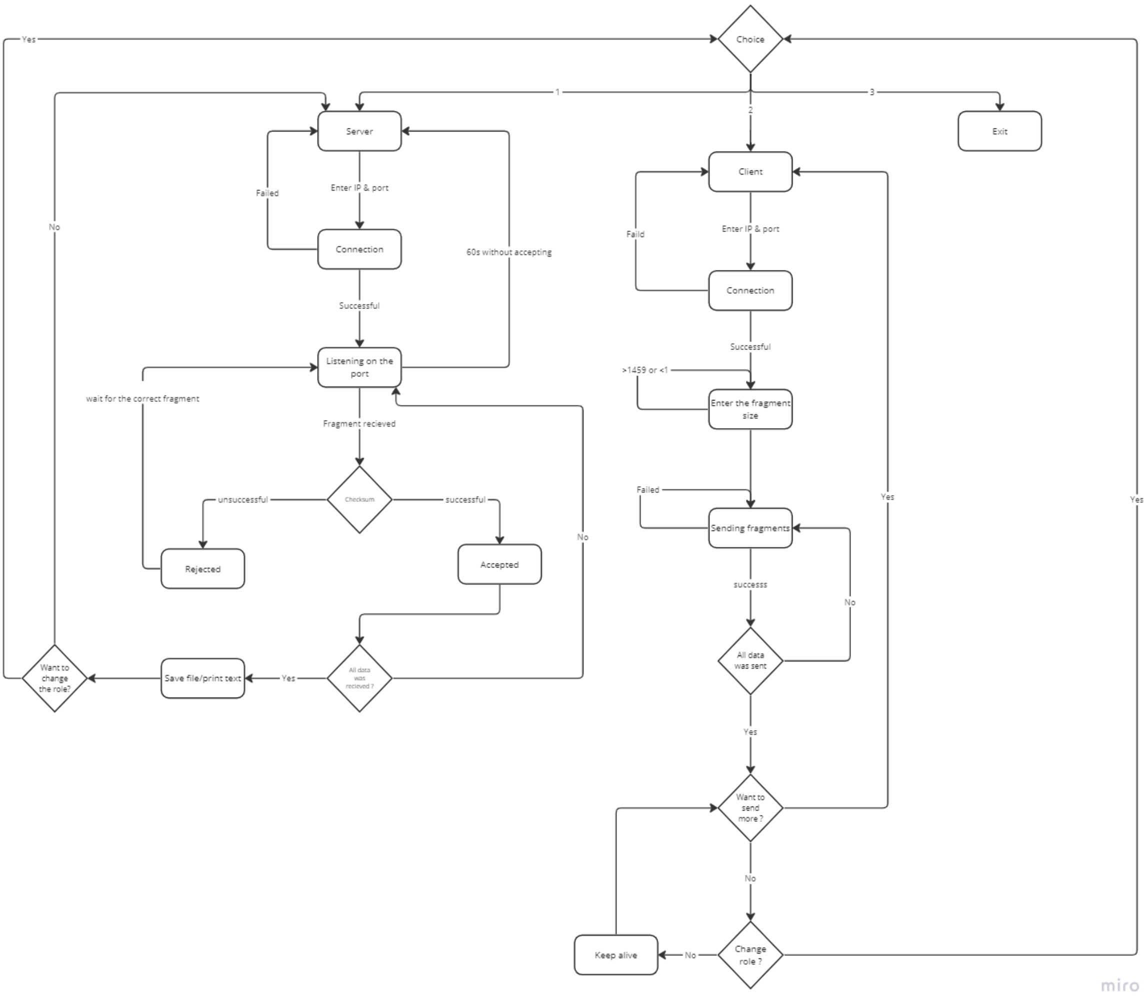
2. Description of the used ARQ operation

In the Stop-and-Wait method, the sender sends a single data frame and then waits for an acknowledgment (ACK) from the receiver. If the receiver successfully receives the frame, it sends an ACK back to the sender. If the sender does not receive an ACK within a specified time, it assumes that the frame was lost or corrupted and retransmits it. This process continues until the receiver successfully acknowledges the frame. The Stop-and-Wait ARQ method helps prevent errors and ensures the orderly flow of data in network communication.

3. Methods for maintaining the connection

   Keep alive function - keep alive flag is transmitted with every 5 second segment. The connection is lost if sent flag didn't receive a response in more than 60 seconds. If not, the user has the option to break the connection by providing a specific variable as an input.

4. Sequence diagram of communication processing on both nodes
   https://miro.com/welcomeonboard/YUVJOEdnVjhENnI3eWpFUDVMdnpvU1JabEtBYVFGTENPVzdTSGFlVnJNR0MzUnNTYUNiY0xMeXNOczhTeWZ4cXwzNDU4NzY0NTY1NjQ0MTg0NTA3fDI=?share_link_id=55056489355



5. Description of individual parts of the source code

Warning! As far as it is only suggestion of the possible solution, the source code can be changed in the future. The description of some function is represented as its prototype and new method, libraries or classes can be added to the final version.

1. Libraries:

socket – used for creating and managing new sockets
threading – used for ARQ method
zlib – used for crc32
time – used for keep alive method

2.Functions:

flags() – describes the meaning of given flag value
checksum() – calculates the crc32 checksum for a given data string
check_fragment_size() – ensures the size of the fragment is within a valid range
create_header() – creates a header dictionary for a message
handle_client() – handles a new client connection, extracting information from the header and performs checksum validation
send() – sends a message to the server with a custom header
start() – initialize the client socket, takes user input for a message and sends it to the server.
client() – initializes the client socket, takes user input for a message, and sends it to the server

Changes to the design of the program and communication protocol

1. Header structure

Flags (1B)	Sequence number of the fragment(3B)	Fragment size(2B)	Checksum(4B)	Num of fragments(3B)	Data (1459B)

The header was modified the following way:

1. The size of seq num was decreased from 4 to 3B (now the max number of fragments to be send is 2^24-1 = 16777215 fragments).
2. As far as fragment size is limited by the data size that is 1459B, it could not have a value larger than that. 2B could be equal to the max value of 2^16-1 = 65535 (>1459)
3. Number of fragments is a new header part that uses 3B or the header and carries number of fragments to be send.

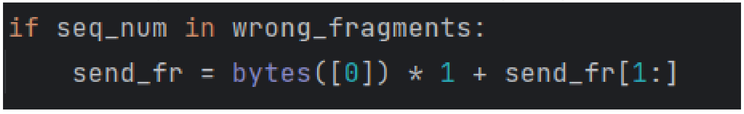
4. Flags has size of 1B and could have 2^8 – 1 = 255 different values. Flags are defined in the program the following way:

00000001	Start of the connection
00000010	End of the connection
00000011	Fragment received
00000100	Fragment rejected
00000101	Keep alive
00000110	File
00000111	Message
00001000	Switch
00001001	Connection confirmed

ARQ method and checksum remains the same. In methods of maintaining connection time without answer was decreased to 30 seconds instead of 60 seconds.

Main methods used in the project

1. Error simulation:
   The program simulates errors in the fragments extracting the data from the sending packet and changing its first byte to "0". Taking more than 6 bits guarantee that checksum 32 will detect the error in the data with 99,9% possibility. The sequence number of the fragments to be damaged are initialized in "wrong fragments" array.



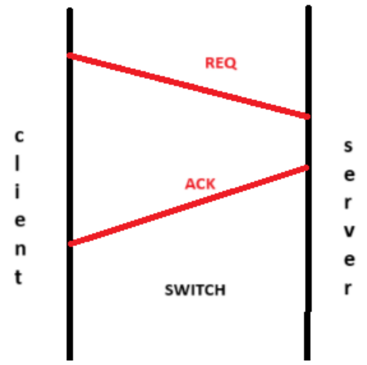
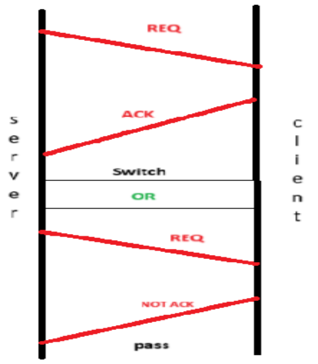
```
if seq_num in wrong_fragments:
    send_fr = bytes([0]) * 1 + send_fr[1:]
```

pic1 – error simulation.

2. Keep alive:
   Keep alive is a method that allows to heck if the connection between client and server is active while running the program. The function that allows to maintain the connection runs on a separate thread that allows it to be functional while running the main program. However, when the user starts to send either message or file the thread stops that is helpful in preventing the conflicts between ACK that server sends to client. If the time of keep alive runs out the program will take the user back to the main menu. Another way, if keep alive was interrupted due to send/ receive request the timer will stop and the thread is waiting for the final packet to be send.

3. Switch

Switch function is implemented so that both client and server could send a switch request and the roles will automatically change. From the client side the option to send the choice request will be given immediately after establishing the connection and then after sending data the user must choose either to change the role or stay in the same mode as previously. From the server side the option will appear only after receiving all the data from the client.

Client - > Server	Server - > Client
	

How does client and server work ?

In this part of documentation two functions that are a considerable amount of whole communication will be described and their functionality will be explained.

- Server()

Server function is a function that listens to and wait for any request to be accepted or rejected.

Example:

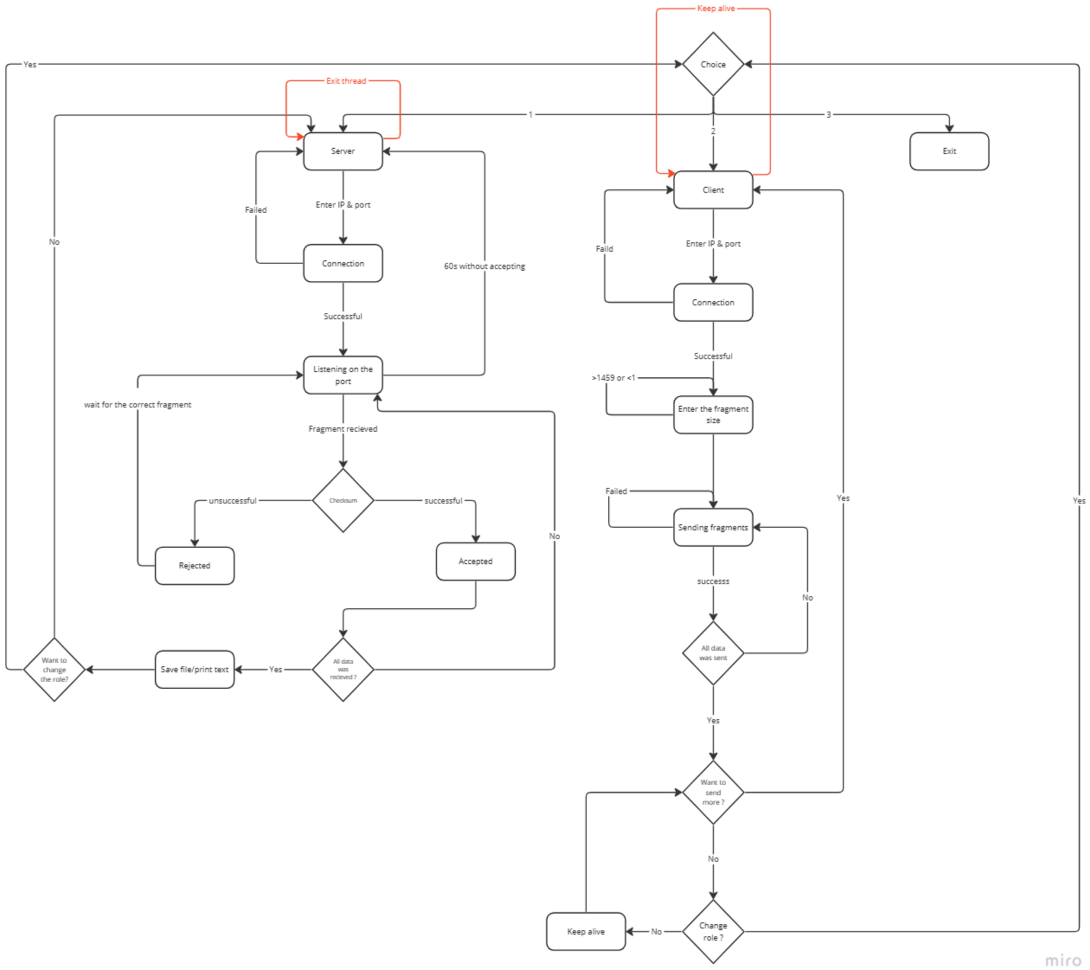
After confirming the connection and receiving the flag that a message will be send to the client, firstly, keep alive function will stops. Then the program extract information about current seq number of the fragment and number of fragments to be send. Then the data part of received packet will be trough checksum verification. From now on there are two possible scenarios: accept the fragment or reject it. If checksum verification was successful message part will be decoded and appended to the final message. Accept flag will be send back to the client and the cycle will continue till all fragments are received. If data didn't make checksum verification the reject flag will be send.

When receiving the file algorithm remains the same. The only difference is that after receiving every single fragment it will ask for the absolute path to save the file (input example: "C:\Users\Username\Desktop\). The file name will be sent to the server with the 0th seq number and the file will be saved to that directory with the same filename as it was for the client.

After receiving either message or file a switch request is send to the client. User (on the client's side can receive or reject the request). Is confirmation flag received user and server will automatically switch or continue in its current mode if reject was sent.

- Client()
  Client is a function that is interactive with the user. After establishing the connection 3 choices will be given rather send a message, file, or exit.
  Example:
  If a user chose to send a message keep alive will stop and the user will be asked to enter a message to send. After receiving the message and fragment size, data will be spitted to small messages to fit the size of one fragment. If a message to be send is smaller than chosen by user fragment size or last fragment can be send in a smaller packet the message "Fragment size was decreased to {fragment_size} for the last fragment" will appear. Right after that, number of fragments will be counted and sending messages will start (could be with or without simulating error). The client than will wait for ACK from the server`s side and send it again if a fragment was rejected. After sending all packets keep alive will turn on and the process repeats.
  Sending file is mostly the same. The important note about the file that it can be send only from cwd.

It is well seen that the main structure of the program hasn't changed and mostly in everything is the same as in the suggestion of the solution. The changes were in adding separate threads (they are red in the diagram)

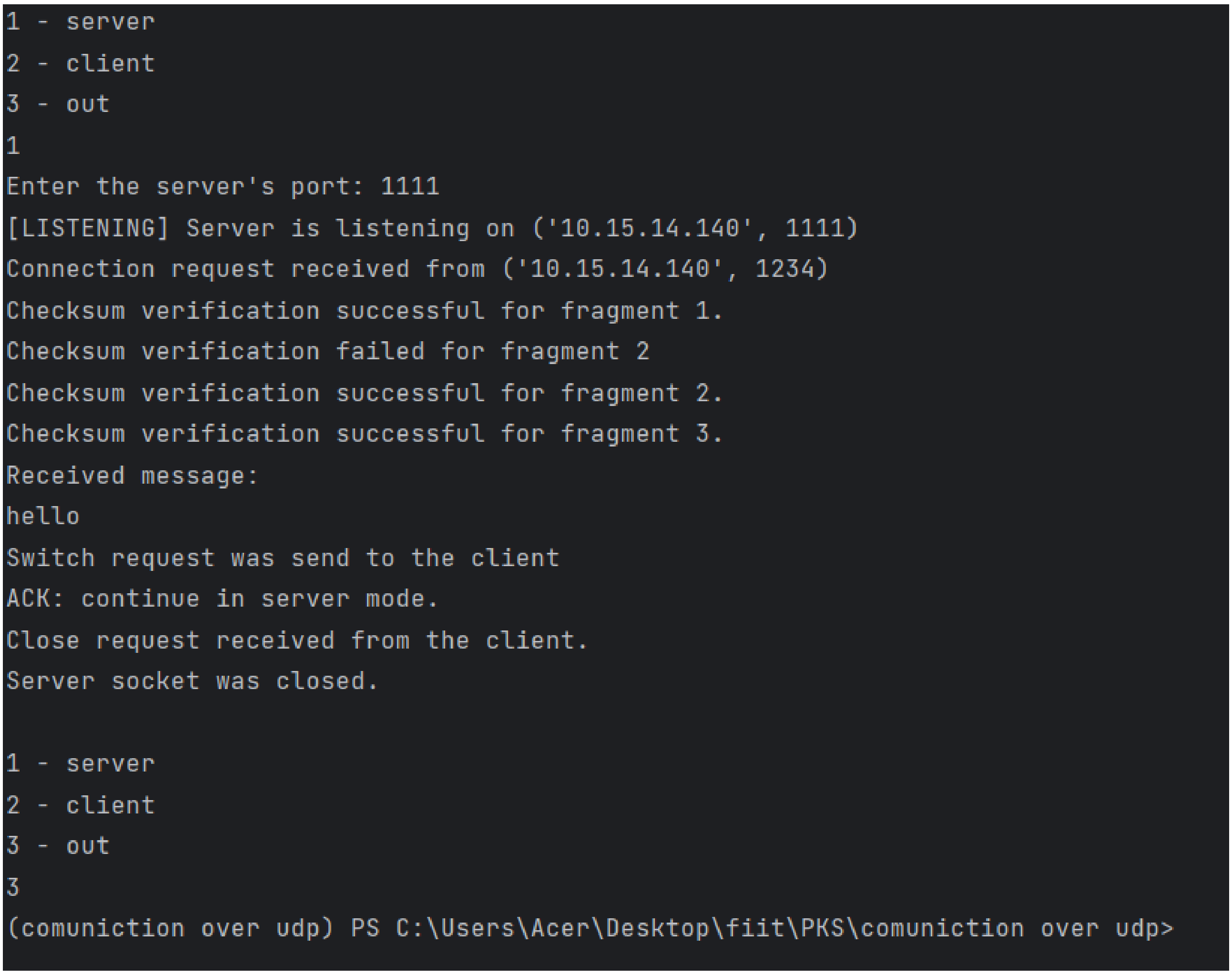


User Interface

In this part the user interface in some testing scenarios will be shown. All scenarios are simulating error on the second fragment.

1. Sending a message, no switch from any side

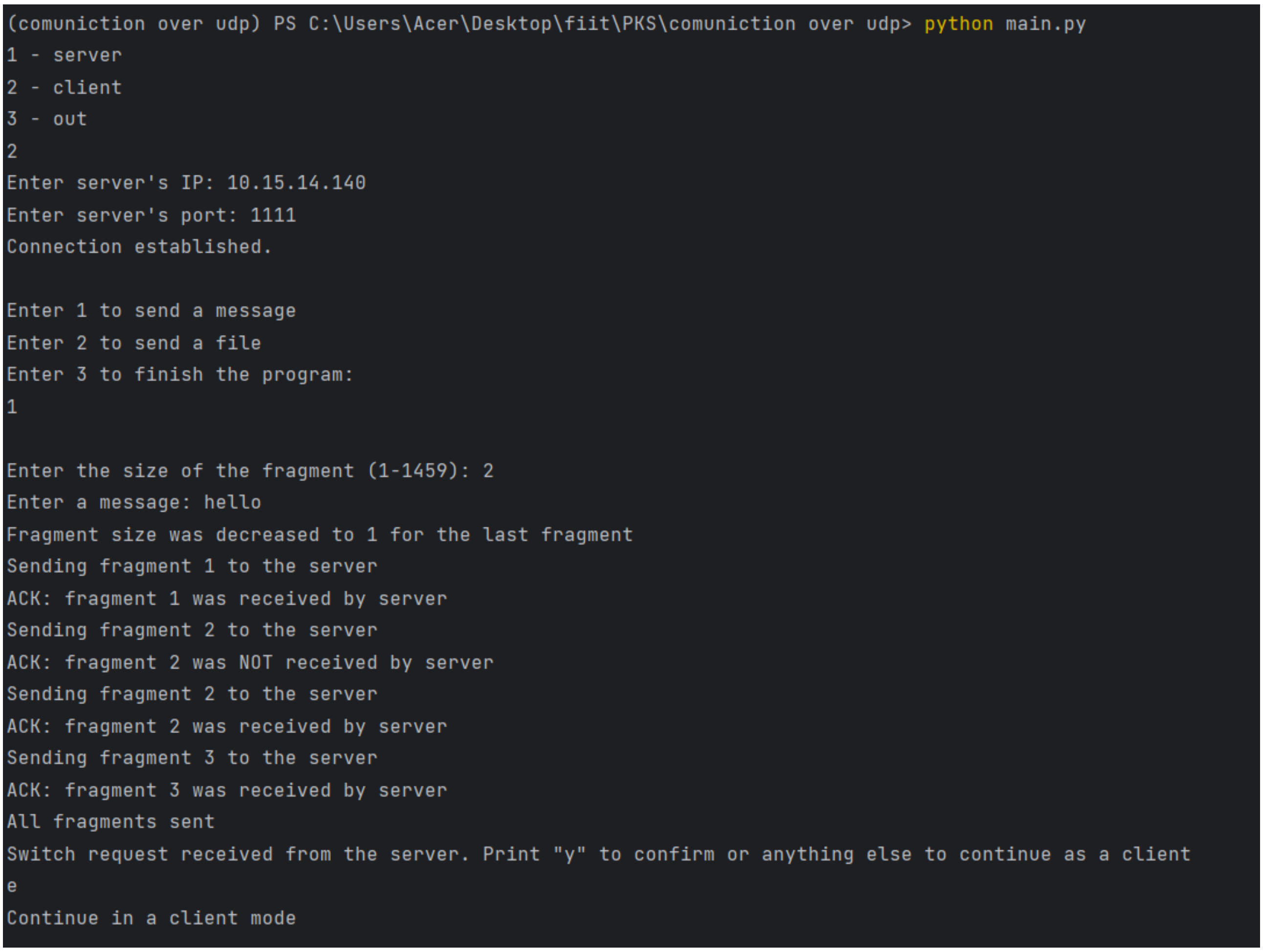
Server:



```
1 - server
2 - client
3 - out
1
Enter the server's port: 1111
[LISTENING] Server is listening on ('10.15.14.140', 1111)
Connection request received from ('10.15.14.140', 1234)
Checksum verification successful for fragment 1.
Checksum verification failed for fragment 2
Checksum verification successful for fragment 2.
Checksum verification successful for fragment 3.
Received message:
hello
Switch request was send to the client
ACK: continue in server mode.
Close request received from the client.
Server socket was closed.

1 - server
2 - client
3 - out
3
(comuniction over udp) PS C:\Users\Acer\Desktop\fiit\PKS\comuniction over udp>
```

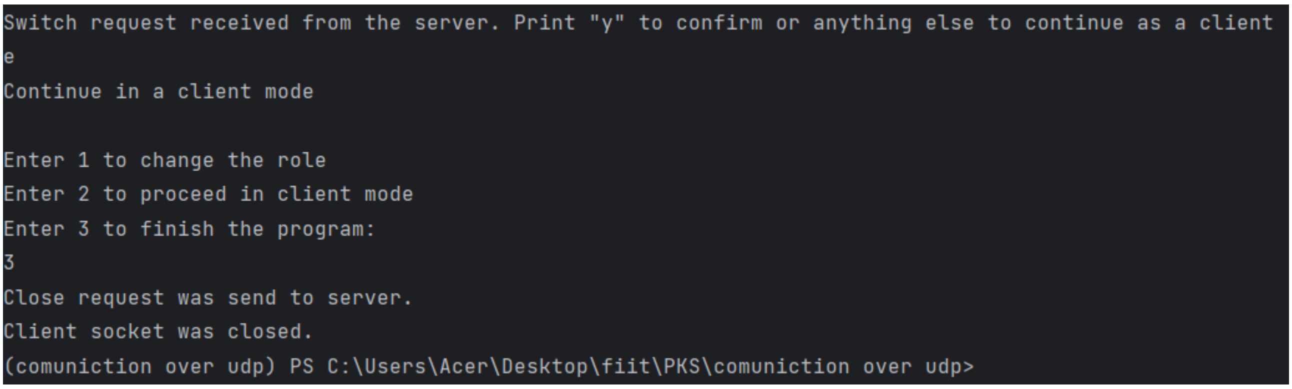
Client:



```
(comuniction over udp) PS C:\Users\Acer\Desktop\fiit\PKS\comuniction over udp> python main.py
1 - server
2 - client
3 - out
2
Enter server's IP: 10.15.14.140
Enter server's port: 1111
Connection established.

Enter 1 to send a message
Enter 2 to send a file
Enter 3 to finish the program:
1

Enter the size of the fragment (1-1459): 2
Enter a message: hello
Fragment size was decreased to 1 for the last fragment
Sending fragment 1 to the server
ACK: fragment 1 was received by server
Sending fragment 2 to the server
ACK: fragment 2 was NOT received by server
Sending fragment 2 to the server
ACK: fragment 2 was received by server
Sending fragment 3 to the server
ACK: fragment 3 was received by server
All fragments sent
Switch request received from the server. Print "y" to confirm or anything else to continue as a client
e
Continue in a client mode
```

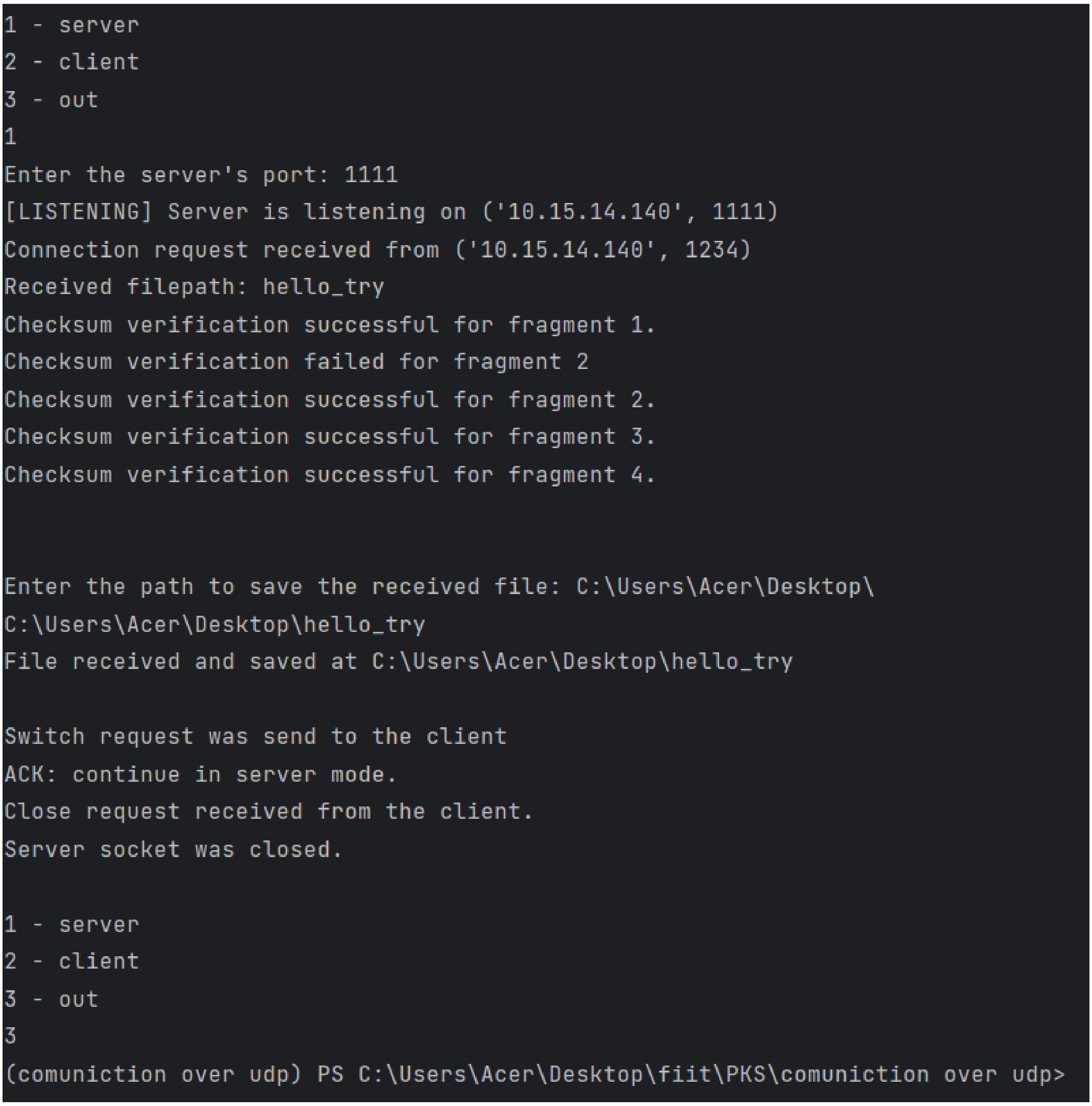


```
Switch request received from the server. Print "y" to confirm or anything else to continue as a client
e
Continue in a client mode

Enter 1 to change the role
Enter 2 to proceed in client mode
Enter 3 to finish the program:
3
Close request was send to server.
Client socket was closed.
(comuniction over udp) PS C:\Users\Acer\Desktop\fiit\PKS\comuniction over udp>
```

2. Sending a file, no switch from any side

Server:



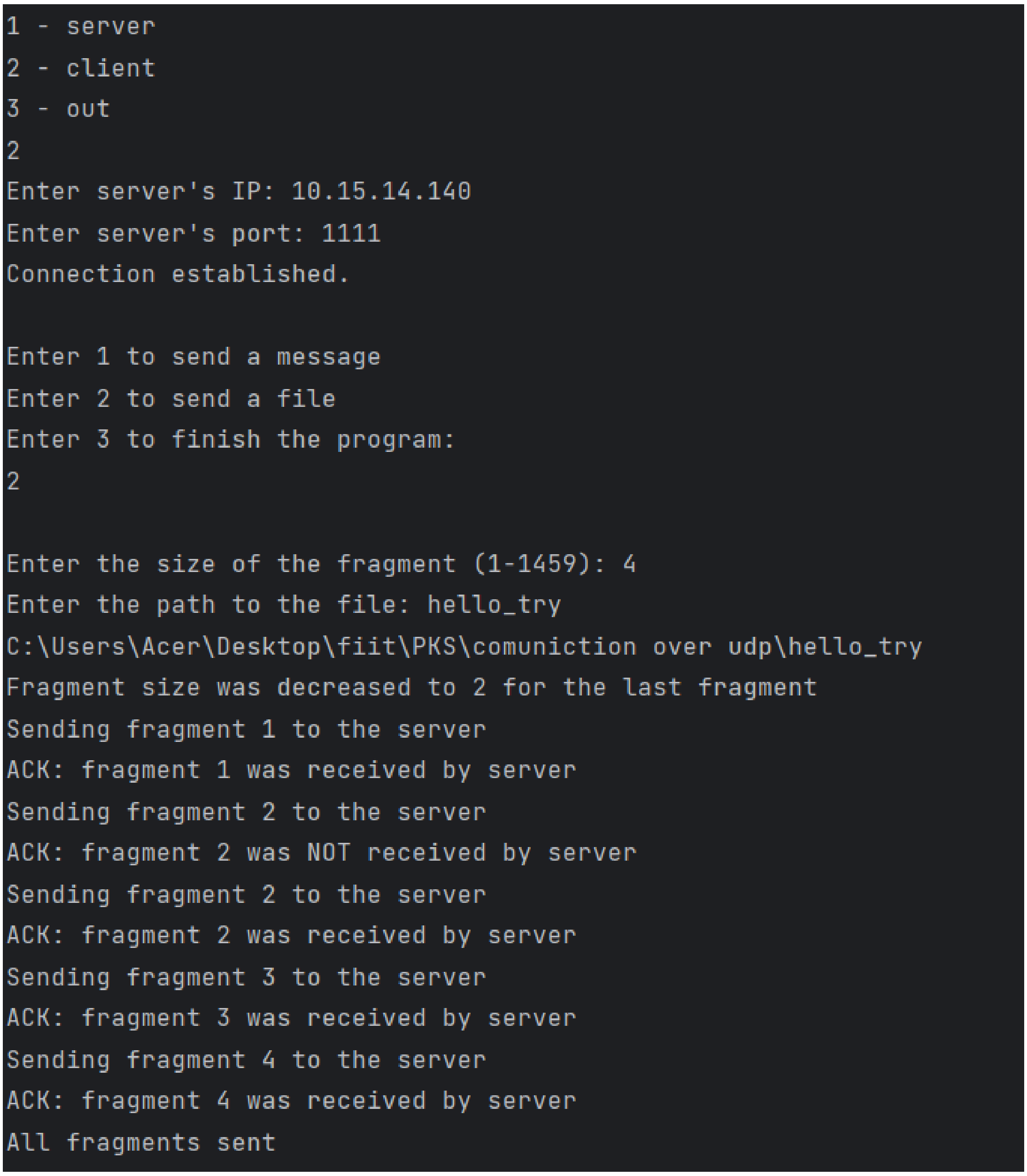
```
1 - server
2 - client
3 - out
1
Enter the server's port: 1111
[LISTENING] Server is listening on ('10.15.14.140', 1111)
Connection request received from ('10.15.14.140', 1234)
Received filepath: hello_try
Checksum verification successful for fragment 1.
Checksum verification failed for fragment 2
Checksum verification successful for fragment 2.
Checksum verification successful for fragment 3.
Checksum verification successful for fragment 4.


Enter the path to save the received file: C:\Users\Acer\Desktop\
C:\Users\Acer\Desktop\hello_try
File received and saved at C:\Users\Acer\Desktop\hello_try

Switch request was send to the client
ACK: continue in server mode.
Close request received from the client.
Server socket was closed.

1 - server
2 - client
3 - out
3
(comuniction over udp) PS C:\Users\Acer\Desktop\fiit\PKS\comuniction over udp>
```

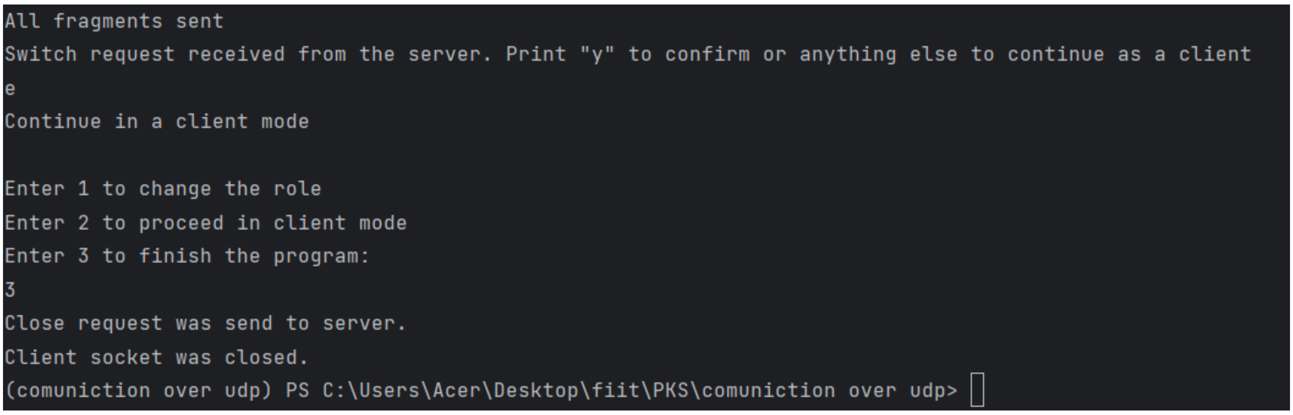
Client:



```
1 - server
2 - client
3 - out
2
Enter server's IP: 10.15.14.140
Enter server's port: 1111
Connection established.

Enter 1 to send a message
Enter 2 to send a file
Enter 3 to finish the program:
2

Enter the size of the fragment (1-1459): 4
Enter the path to the file: hello_try
C:\Users\Acer\Desktop\fiit\PKS\comuniction over udp\hello_try
Fragment size was decreased to 2 for the last fragment
Sending fragment 1 to the server
ACK: fragment 1 was received by server
Sending fragment 2 to the server
ACK: fragment 2 was NOT received by server
Sending fragment 2 to the server
ACK: fragment 2 was received by server
Sending fragment 3 to the server
ACK: fragment 3 was received by server
Sending fragment 4 to the server
ACK: fragment 4 was received by server
All fragments sent
```

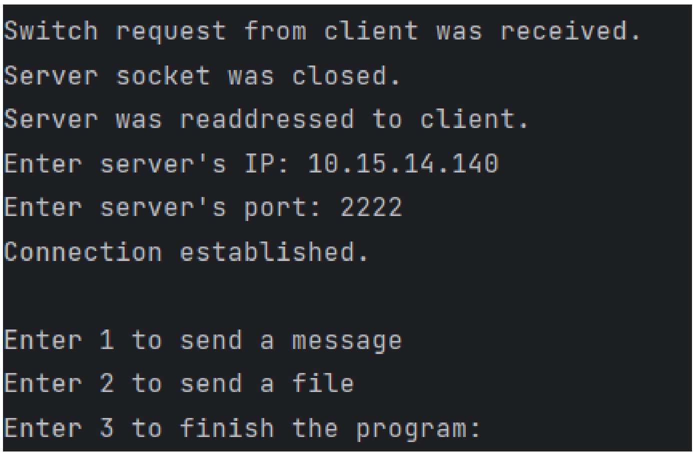


```
All fragments sent
Switch request received from the server. Print "y" to confirm or anything else to continue as a client
e
Continue in a client mode

Enter 1 to change the role
Enter 2 to proceed in client mode
Enter 3 to finish the program:
3
Close request was send to server.
Client socket was closed.
(comuniction over udp) PS C:\Users\Acer\Desktop\fiit\PKS\comuniction over udp> 
```

3. Switch request send from the clint to server:

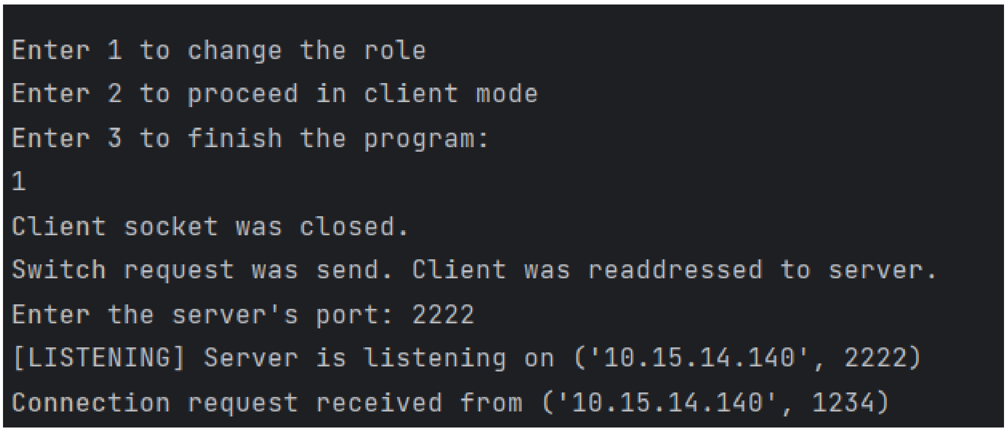
Server:



```
Switch request from client was received.
Server socket was closed.
Server was readdressed to client.
Enter server's IP: 10.15.14.140
Enter server's port: 2222
Connection established.

Enter 1 to send a message
Enter 2 to send a file
Enter 3 to finish the program:
```

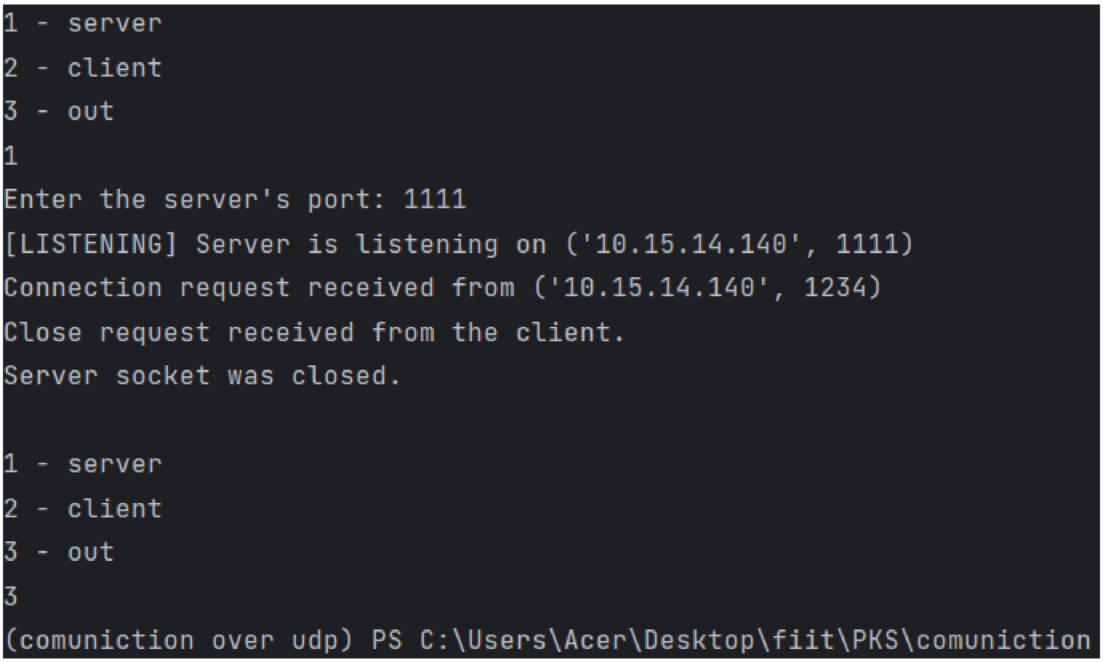
Client:



```
Enter 1 to change the role
Enter 2 to proceed in client mode
Enter 3 to finish the program:
1
Client socket was closed.
Switch request was send. Client was readdressed to server.
Enter the server's port: 2222
[LISTENING] Server is listening on ('10.15.14.140', 2222)
Connection request received from ('10.15.14.140', 1234)
```

4.Keep alive terminates the connection:

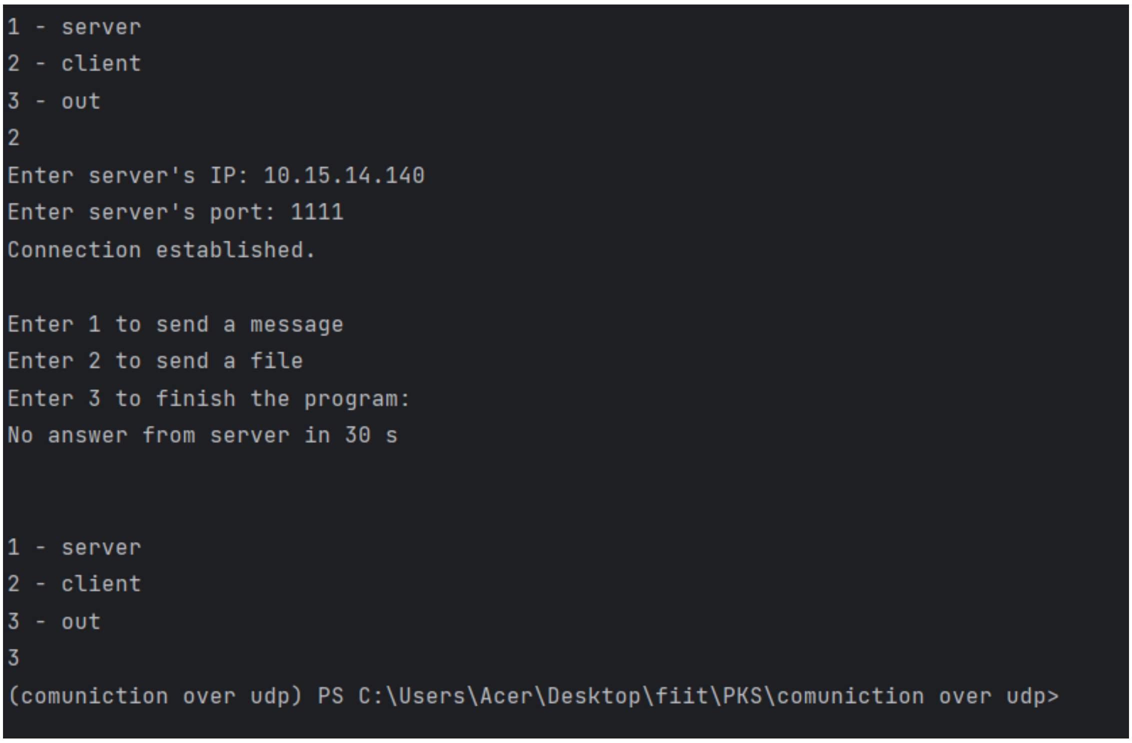
Server:



```
1 - server
2 - client
3 - out
1
Enter the server's port: 1111
[LISTENING] Server is listening on ('10.15.14.140', 1111)
Connection request received from ('10.15.14.140', 1234)
Close request received from the client.
Server socket was closed.

1 - server
2 - client
3 - out
3
(comuniction over udp) PS C:\Users\Acer\Desktop\fiit\PKS\comuniction
```

Client:



```
1 - server
2 - client
3 - out
2
Enter server's IP: 10.15.14.140
Enter server's port: 1111
Connection established.

Enter 1 to send a message
Enter 2 to send a file
Enter 3 to finish the program:
No answer from server in 30 s


1 - server
2 - client
3 - out
3
(comuniction over udp) PS C:\Users\Acer\Desktop\fiit\PKS\comuniction over udp>
```

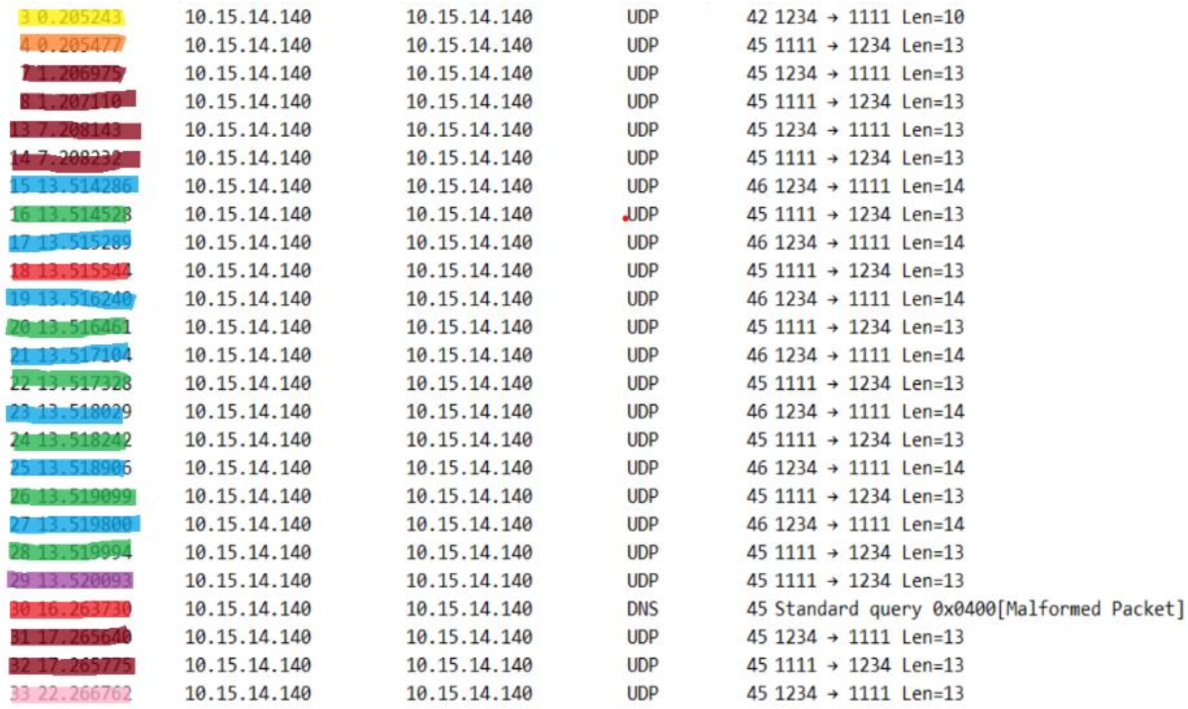
Wireshark capture

The color legend:

🟨	Start of the connection
🟪	End of the connection

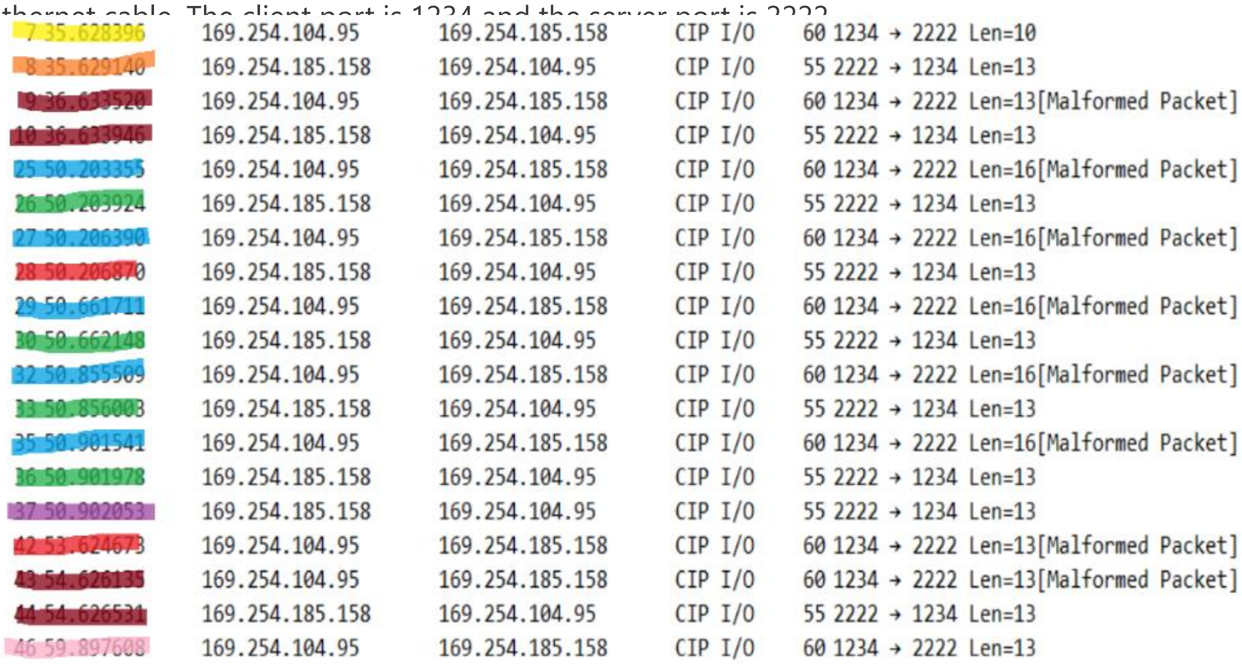
🟩	Fragment received
🟥	Fragment rejected
🟫	Keep alive
-	File
🟦	Message
🟪	Switch
🟧	Connection confirmed

1.This is an example of captured communication. The client port is 1234 and the server port is 1111. The communication was captured on the Wi-Fi.



```
3 0.205243    10.15.14.140    10.15.14.140    UDP    42 1234 → 1111 Len=10
4 0.205477    10.15.14.140    10.15.14.140    UDP    45 1111 → 1234 Len=13
7 1.206975    10.15.14.140    10.15.14.140    UDP    45 1234 → 1111 Len=13
8 1.207110    10.15.14.140    10.15.14.140    UDP    45 1111 → 1234 Len=13
13 7.208143   10.15.14.140    10.15.14.140    UDP    45 1234 → 1111 Len=13
14 7.208232   10.15.14.140    10.15.14.140    UDP    45 1111 → 1234 Len=13
15 13.514286  10.15.14.140    10.15.14.140    UDP    46 1234 → 1111 Len=14
16 13.514528  10.15.14.140    10.15.14.140    UDP    45 1111 → 1234 Len=13
17 13.515289  10.15.14.140    10.15.14.140    UDP    46 1234 → 1111 Len=14
18 13.515544  10.15.14.140    10.15.14.140    UDP    45 1111 → 1234 Len=13
19 13.516240  10.15.14.140    10.15.14.140    UDP    46 1234 → 1111 Len=14
20 13.516461  10.15.14.140    10.15.14.140    UDP    45 1111 → 1234 Len=13
21 13.517104  10.15.14.140    10.15.14.140    UDP    46 1234 → 1111 Len=14
22 13.517328  10.15.14.140    10.15.14.140    UDP    45 1111 → 1234 Len=13
23 13.518029  10.15.14.140    10.15.14.140    UDP    46 1234 → 1111 Len=14
24 13.518242  10.15.14.140    10.15.14.140    UDP    45 1111 → 1234 Len=13
25 13.518906  10.15.14.140    10.15.14.140    UDP    46 1234 → 1111 Len=14
26 13.519099  10.15.14.140    10.15.14.140    UDP    45 1111 → 1234 Len=13
27 13.519800  10.15.14.140    10.15.14.140    UDP    46 1234 → 1111 Len=14
28 13.519994  10.15.14.140    10.15.14.140    UDP    45 1111 → 1234 Len=13
29 13.520093  10.15.14.140    10.15.14.140    UDP    45 1111 → 1234 Len=13
30 16.263730  10.15.14.140    10.15.14.140    DNS    45 Standard query 0x0400[Malformed Packet]
31 17.265640  10.15.14.140    10.15.14.140    UDP    45 1234 → 1111 Len=13
32 17.265775  10.15.14.140    10.15.14.140    UDP    45 1111 → 1234 Len=13
33 22.266762  10.15.14.140    10.15.14.140    UDP    45 1234 → 1111 Len=13
```

2. This is similar experiment with sending message, but the connection was established using Ethernet cable. The client port is 1234 and the server port is 2222.



```
7 35.628396    169.254.104.95     169.254.185.158    CIP I/O    60 1234 → 2222 Len=10
8 35.629140    169.254.185.158    169.254.104.95     CIP I/O    55 2222 → 1234 Len=13
9 36.633520    169.254.104.95     169.254.185.158    CIP I/O    60 1234 → 2222 Len=13[Malformed Packet]
10 36.633946   169.254.185.158    169.254.104.95     CIP I/O    55 2222 → 1234 Len=13
25 50.203355   169.254.104.95     169.254.185.158    CIP I/O    60 1234 → 2222 Len=16[Malformed Packet]
26 50.203924   169.254.185.158    169.254.104.95     CIP I/O    55 2222 → 1234 Len=13
27 50.206390   169.254.104.95     169.254.185.158    CIP I/O    60 1234 → 2222 Len=16[Malformed Packet]
28 50.206870   169.254.185.158    169.254.104.95     CIP I/O    55 2222 → 1234 Len=13
29 50.661711   169.254.104.95     169.254.185.158    CIP I/O    60 1234 → 2222 Len=16[Malformed Packet]
30 50.662148   169.254.185.158    169.254.104.95     CIP I/O    55 2222 → 1234 Len=13
32 50.855509   169.254.104.95     169.254.185.158    CIP I/O    60 1234 → 2222 Len=16[Malformed Packet]
33 50.856003   169.254.185.158    169.254.104.95     CIP I/O    55 2222 → 1234 Len=13
35 50.901541   169.254.104.95     169.254.185.158    CIP I/O    60 1234 → 2222 Len=16[Malformed Packet]
36 50.901978   169.254.185.158    169.254.104.95     CIP I/O    55 2222 → 1234 Len=13
37 50.902053   169.254.185.158    169.254.104.95     CIP I/O    55 2222 → 1234 Len=13
42 53.624673   169.254.104.95     169.254.185.158    CIP I/O    60 1234 → 2222 Len=13[Malformed Packet]
43 54.626135   169.254.104.95     169.254.185.158    CIP I/O    60 1234 → 2222 Len=13[Malformed Packet]
44 54.626531   169.254.185.158    169.254.104.95     CIP I/O    55 2222 → 1234 Len=13
46 59.897608   169.254.104.95     169.254.185.158    CIP I/O    60 1234 → 2222 Len=13
```

Conclusion

In conclusion, socket programming is used in the provided Python code to construct a basic UDP communication system. A server and a client make up the system, which enables file and message sharing between them. The code is divided into functions, each of which has a distinct function. Multithreading is used to manage the user input and the main communication loop concurrently. The server can handle switch requests from the client, receive and respond to messages, listen for incoming connections, and store files that are received from the client. In addition, if the client sends a close request, the server can close gracefully.

The client, on the other hand, is made to connect to the server, send switch requests to switch between the client and server's responsibilities, and transfer files or text messages to the server. To maintain the stability of the connection, the client also has a keep-alive feature that allows it to end the program gently upon request. A unique header format that contains fields like the flag, sequence number, fragment size, checksum, and number of fragments is used for communication between the server and the client. Using the zlib library, a straightforward checksum verification method preserves the data's integrity.

The code offers the bare minimum for communication, but it may be improved upon in a few areas, including error handling, user interface, and possible multithreading synchronization problems.