Assignment 6

Student Name: Filipovych Marharyta

Group: ПКСП-1

Github Link: *https://github.com/VY-Assignments/client-server-chat-app-MarharytaFilipovych*

Task: Design and implement a multi-room chat application using TCP connections. The goal is to create a robust chat system that allows users to communicate in different rooms and even share files with other members within one room.

Solution description:

The server and the client applications are organized into several classes and structs. They both have almost identical structs **Receiving** and **Sending**, which are responsible for receiving and sending data accordingly.

The **Sending** struct provides several static methods for sending different types of information:

- **sendOneByte**: Sends a single byte (char) to the client or server (depending on where this struct is located). It is widely used while sending TLVs and sometimes for confirmation purposes.
- **sendIntegerValue**: Sends an integer value in network byte order to ensure compatibility across different systems. Used for sending the length of the messages and the size of the file.
- **sendMessage**: used for sending text messages. This method has a slight difference in these apps. In the Server, it is already implemented to send the message as a TLV (first send a one-byte tag, then the data length, then the data itself). While in the Client it is implemented to send the length and the data only (without tags, they are handled separately).
- **sendFile**: Sends a file in binary format in chunks of 1024 bytes. Both sendFile programs' methods differ slightly in implementation, but the overall purpose is the same. Th details of sending files are described in the protocol.

The **Receiving** also provides several static methods for receiving data of various types:

- **receiveInteger**: Receives an integer value representing the size of the message or file.
- **receiveMessage**: Receives a text message of various lengths. First it receives its length with the help of the receiveInteger method, then the message data itself.
- **receiveOneByte**: Receives a single byte (char) used as a tag to classify the type of data sent and returns this byte as a char value.

- **receiveFile**: Receives a file in binary format in a 1024-byte chunk approach. Then, it saves the file in the dedicated database directory.

Server specifics overview:

The **Client** struct represents each connected client. It stores the client's **socket descriptor**, **name**, and the room number to which the client is currently assigned. Clients are dynamically created the moment a new connection is accepted, and they are uniquely identified by their socket and name, ensuring that each user is easily distinguishable during communication.

The **Message** struct serves as the fundamental unit for message exchange. Each message consists of the **message content**, the **socket descriptor** of the sender, and a **tag byte** that indicates the message type. The struct also contains an optional field called **add_info**, which carries additional information, such as the file name during file transfers.

The core of the chat functionality revolves around the **Room** class. Each room represents a separate virtual chat space where clients can communicate independently. The class manages an **unordered set of client sockets** (I wanted to ensure each client's uniqueness within the same room and fast search), a **message queue** that temporarily stores incoming messages, and two **mutexes** to synchronize access to unordered set of clients and message queue. Additionally, each room uses a **condition variable** to notify the room's manager thread whenever new messages arrive. The primary method of the class is **broadcastMessage**, which continuously monitors the message queue and distributes messages to all clients in the room, except the sender, or sometimes only to the sender (it depends on the tag). It sends the messages only when they are available and when there are clients in the room. These conditions are monitored with the help of the condition variable. The **addClient** method is responsible for adding new clients to the room and notifying **broadcastMessage** method about their arrival. The **takeClientAway** method removes clients and sends departure notifications. Each room is operated on its separate dedicated thread.

The **Chat** class acts as the central manager. This class maintains a **vector of unique pointers** to **Room** objects created at startup. The **createRooms** method is called during initialization to set up the rooms. It constructs room objects that create a separate thread for each room management. When a new client connects, the server first verifies the connection through the **isGreetingSuccessful** method, which checks if the client sends the expected greeting message: "Hello, server! This is client:)". If the message is valid, the client proceeds to the registration phase.

The registration process consists of three steps: requesting the client's **name** via the **getName** method, receiving the selected **room number** through **getRoom**, and finally assigning the client to the corresponding room. This entire process is handled by the **registerClient** method. Once registered, the server listens for incoming data through the handleClient **method which calls the handleIncomingData** method to processes client messages based on their tag. Text messages, room change requests, file transfer requests, and file acceptance or rejection are handled according to the tag byte. Clients can dynamically switch between rooms using the **changeRoom** method.

File transfers are managed by the **handleTag3** method. When a client initiates a file transfer, the server receives the file, stores it in the **database directory**, and notifies other clients about the availability of the file. Clients then receive a prompt asking if they want to accept the file. If they agree, the server transfers the file to them in chunks. Their disagreement or agreement is indicated within the tag 0x04.

The **closeClient** method ensures that each client is properly disconnected by closing the socket and removing the client from the room, as well as notifying other clients within that room about this client departure.

The **main server loop** continuously listens for incoming connections and spawns a new **thread** for each client, calling the **handleClient** method. The server runs indefinitely, accepting and managing clients until the program is terminated.

Client specifics overview:

The registration process is encapsulated in the **Registration** class. Once the connection is established, the client sends an initial greeting message to the server - "Hello, server! This is client:)". The server responds with a 0x06 tag that either accepts or rejects the client. If the client is rejected, the socket simply closes, and the program terminates. Then, the client is asked to enter a name, which will also be used as the folder name to store received files. If the clients have the same names, their folders are still different. That is because inside the name-folder the folder named after client's socket number is created. It is here, where the database for a separate client lies. Then the client must select the room number where it wants to join. If the room number is invalid, (e.g. the number is not a digit), the user will be prompted about that immediately and asked for the room number again. When everything is fine, the server confirms that registration is complete.

The **Communication** class is the central part of the client application and manages the entire communication between the client and the server. Its primary purpose is to enable simultaneous message exchange and user input processing using 2 threads and synchronization mechanisms. The class ensures that text messages and file transfers are handled smoothly without blocking the console. It operates on two threads — one dedicated to listening for incoming server messages (receiver thread, created additionally) and another for processing user input and sending data (main thread). These threads work independently but are carefully synchronized using mutexes and condition variables to prevent race conditions. Synchronization between the two threads is achieved through mutexes and condition variables. The **cv_input** condition variable is used to pause user input during file transfers temporarily. In contrast, the **cv_response** condition variable ensures the client waits for the user's answer before proceeding with the file exchange.

The client communicates with the server through a socket, passed as a constant reference to the class to prevent unnecessary copies.

The **sendSimpleMessage** method sends simple messages to the server, combining the message content with the correct tag. If the user requests to change the room, the changeroom method verifies the input using the **InputParser** class and sends the request to the server. The

method then pauses the input thread using a condition variable until the server confirms the room switch.

When the user wants to send a file, the **sendFile** method validates the file path and then sends the file in small binary chunks of 1024 bytes. It temporarily blocks user input to avoid interference during the file download, then writes the received binary chunks to the file. After the transfer, the method prints whether the file was successfully received. Each file request is broadcast to every client in the current room, and the client waits for the user's confirmation using the **handleTag2** method. The **getResponse** method processes the user's decision, which sends either the file name or a rejection message back to the server.

Received file names are stored in the files vector. This vector acts as a queue, temporarily holding the names of files that the client might receive from the server. The file names are pushed into the queue when the server sends a file request, asking the client if they want to receive the file. This happens inside the **handleTag2** method, where the client first receives the question message from the server, followed by the file name.

If the user wants to receive a file and type something like "yes", the **getResponse** method retrieves the first element from the queue and sends the file name back to the server. After the file is successfully received, the name is popped from the queue.

Incoming messages are processed in the **receiveMessages** method, which runs in a loop until the client disconnects. The method continuously listens for new bytes from the server. Based on the received tag, the message is either printed directly to the console, prompts the user for confirmation, or triggers a file download.

For the input part, specific rules are followed, which are printed at the start. The input must always start with a number indicating the message's purpose, except for the cases when the user must answer the question. If the user wants to send a simple text message, he/she must enter 1 followed by the text. They must type 2 and the desired room number to change the room. If they want to send a file, they must type 3 and the full path to it either in quotes or without them.

Another essential part of the client is the InputParser class. This class parses user input, verifies correctness, and extracts the necessary information from the input. It checks if the user entered the correct file path, if the file exists, or if the file is empty. It also helps with text transformation and ensures that answers like "yes" or "yep" are correctly interpreted.

The client is designed to exit gracefully. When the user types "EXIT" (either in lower or upper case) into the console, the client sends a special byte 0x05 to the server, notifying that it wants to leave. Then, the socket closes, and both threads are stopped as well. This is how I achieved FIN-ACK.

Protocol description

Important: The server supports only Latin.

Important: The server always sends **TLVs**. That means it first sends a **1-byte tag**, identifying the message's purpose, then sends the length of this message via **4-byte integer value**, and only after that the **message itself**. Structure:

TAG LENGTH VALUE

1 byte 4 bytes length value bytes (31 in case the message is 31 characters long)

The files are sent similarly. However, some crucial differences need to be outlined. First, the server sends the tag (the list of tags is specified below). Then, it sends the 4-byte integer value, identifying the file size to be sent. Afterward, it starts to send files in 1024-byte chunks.

Server tags:

- 0x01 – simple messages from your neighbors (clients within your room)
- 0x02 – a question from one of your neighbors asking if you want to receive a file. Important! First, you receive TLV with the tag 0x02 and the question. Then, immediately after, the name of the file is sent via TLV marked with the tag 0x04.
  0x02 length question -> 0x04 length file_name
- 0x03 – file is going to be sent
- 0x04 – the name of the file is specified via this tag
- 0x05 – the protocol was ignored (The protocol was ignored), or the connection is about to be closed (Bye client!).
- 0x06 – messages from the server itself. These can be observed during the registration process and confirmation messages.

Registration stage:

The client sends the length of the initial greeting message. It must be 31 characters long.

A client MUST send the message: "Hello, server! This is client:)".

Important: if the client's greeting message differs, the server sends tag 0x05, followed by the message length and the message - "The protocol was ignored!"

The server sends tag 0x06.

The server sends the length of the message.

The server sends the message asking for the client's name: "Tell me your name, please!"

The client must send the length of its name.

The client must send the name.

The server sends tag 0x06.

The server sends the length of the message.

The server sends a message greeting the client and asking for the room the client wants to join: "Hello, NAME! This is the server. What is the room number?"

The client sends the room number in 1 byte.

The server sends 0x01 tag if the room number is valid or 0x00 tag if not. The server will wait and receive 1-byte values from the client until it gets the correct room number.

Server sends the message, which identifies the successful end of the registration process:

"Successfully registered with a name NAME-SOME_ASSIGNED_NUMBER in the room ROOM_NUMBER."

P.S. SOME_ASSIGNED_NUMBER – this is the value (number) of the client's socket.

This is the end of the registration process. Now, the client stays in a particular room and can communicate with other clients within this room. For successful communication, the client, from now on, must send messages in the **TLV** format.

Client tags:

0x01 – simple messages

Example: 0x01 10 Hi friend!

0x02 – change room

Example: 0x02 1 3

P.S. 1 is the length of the message, and "3" is the value sent (the room number).

Important:

If the room number sent by the client is invalid (such a room does not exist or the client already stays in this room), the server will send:

1) **0x06** tag
2) The message length
3) **"Invalid room number!"**

The client must ensure the value sent can be cast to an integer, meaning the message contains only digits.

If the value sent identifies one of the existing rooms (is correct), the server sends:

- To former neighbours:

1) **0x01** tag
2) The message length
3) **"NAME-SOCKET_NUMBER is leaving the room"**

- To new neighbours:

1) **0x01** tag
2) The message length
3) **"NAME-SOCKET_NUMBER joined"**

- To this client:

1) **0x06** tag
2) The message length
3) **"You successfully entered the desired room!"**

0x03 – send file

The client sends tag 0x03.

The client sends the **file name length**.

The client sends the **file name**.

The client sends a **4-byte integer** value identifying the **file size**.

The client sends file **content in 1024-byte chunks**.

Important:

If the server did not receive the file due to some unexpected issues, the server sends:

1) **0x06** tag
2) Message length
3) **"File was not sent due to some unexpected issues!"**

If the server receives the file, the server sends:

1) **0x06** tag
2) Message length
3) **"File was sent!"**

0x04 – The client's response to the question about its desire to receive the file from one of the neighbours. The client must send:

0x04 2 no – **"no"** if it declines this file

0x04 length filename – if it wishes to receive this file, it should send this **file name**

0x05 – this tag should be sent if the client desires to close the connection.

Connection termination:

The client sends **0x05** 1-byte tag to the server.

The server sends the message:

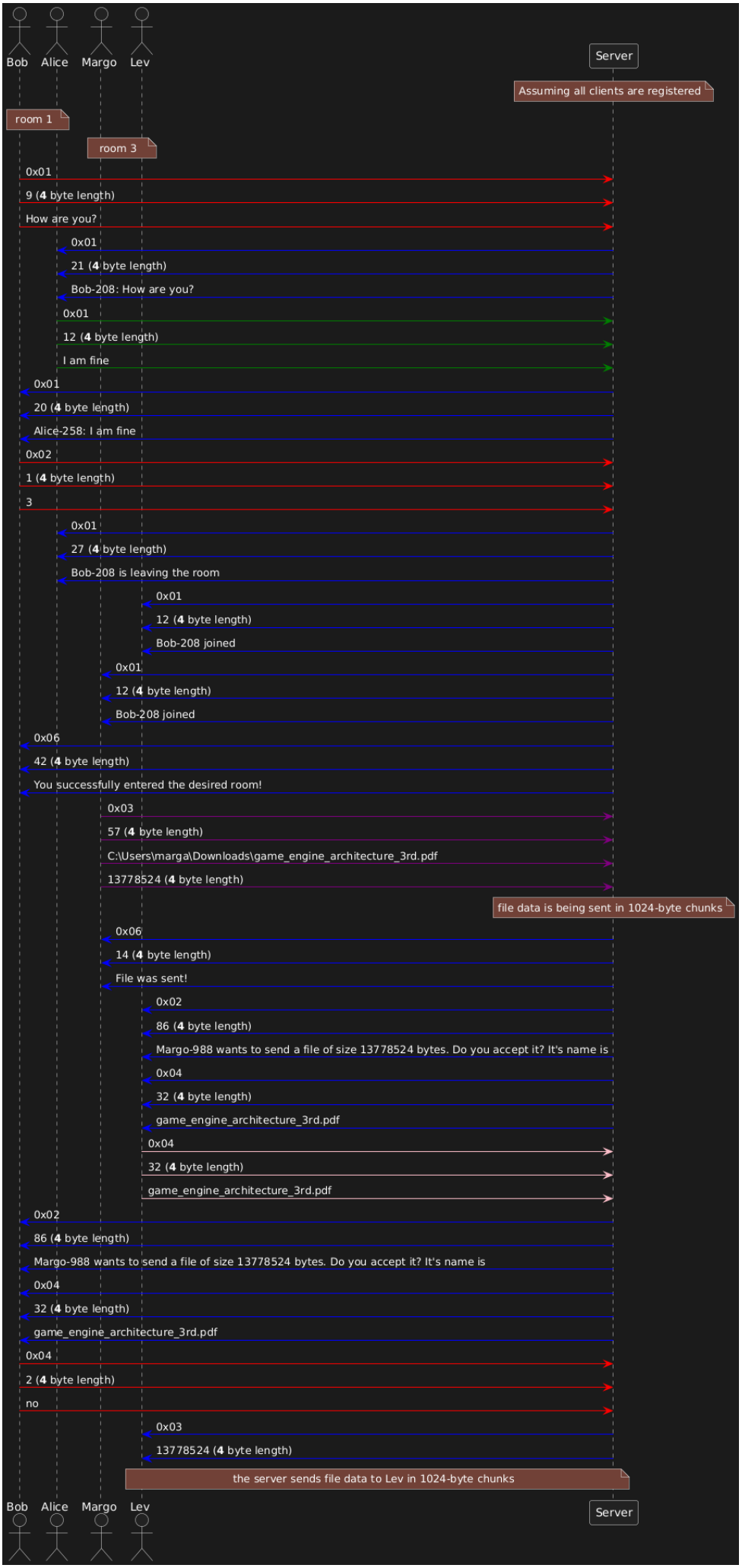
- To this client:

1) **0x05**
2) Message length
3) **"Bye client!"**

- **To former neighbours:**

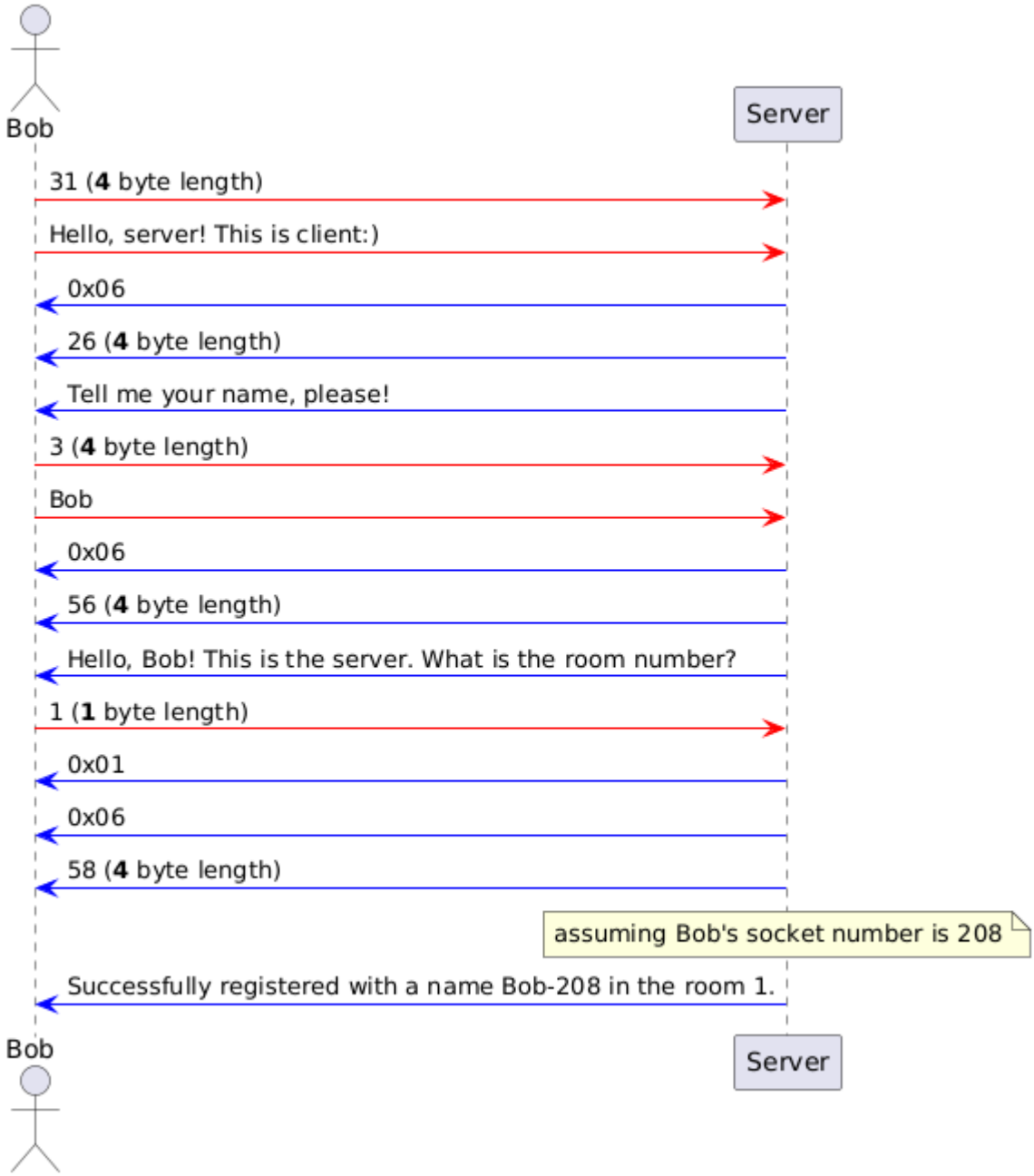
1) **0x01** tag
2) The message length
3) **"NAME-SOCKET_NUMBER is leaving the room"**

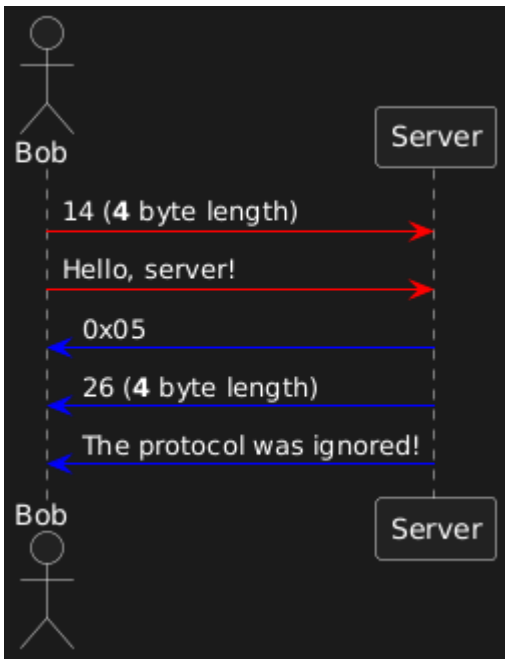
THE CONNECTION IS CLOSED.



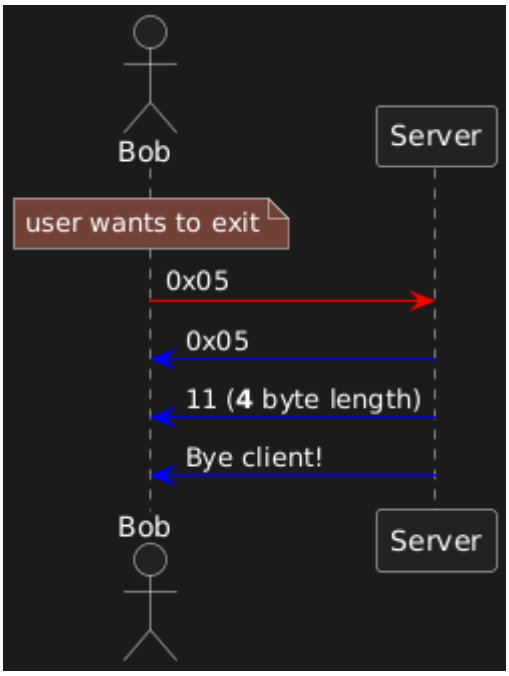
Multiple clients' communication

Successful registration





Protocol was ignored



Graceful termination

Testing:

I launched the server and 4 clients simultaneously.

Server screen:

```
Server listening on port 12345
Client with socket 276: Hello, server! This is client:)
Client with socket 260: Hello, server! This is client:)
Client with socket 296: Hello, server! This is client:)
Client with socket 308: Hello, server! This is client:)
Client olena joined ROOM 0
Client Tolya joined ROOM 0
Client Kate joined ROOM 0
Client Bunya joined ROOM 1
Tolya from room 0: hello
olena from room 0: hello Tolik!
Kate from room 0: hi guys
Kate from room 0: how are you?
olena from room 0: good
Tolya from room 0: not so good, i am tired....
Bunya from room 1: i am alome...
Tolya from room 0: 2
Client Tolya left ROOM 0
Client Tolya joined ROOM 1
olena from room 0: 2
Client olena left ROOM 0
Client olena joined ROOM 1
olena from room 1: C:/Users/marga/Downloads/game_engine_architecture_3rd.pdf
Tolya from room 1: game_engine_architecture_3rd.pdf
Bunya from room 1: game_engine_architecture_3rd.pdf
Kate from room 0: 2
Client Kate left ROOM 0
Client Kate joined ROOM 1
Kate from room 1: C:/Users/marga/Downloads/Practical assignment 2.pdf
Bunya from room 1: Practical assignment 2.pdf
Tolya from room 1: Practical assignment 2.pdf
olena from room 1: no
olena from room 1: let's end the converation
```

```
olena from room 1: 7
olena from room 1: 4
Client olena left ROOM 1
Client olena joined ROOM 3
Client olena left ROOM 3
olena-308 disconnected
Client Bunya left ROOM 1
Bunya-276 disconnected
Client Kate left ROOM 1
Kate-260 disconnected
Client Tolya left ROOM 1
Tolya-296 disconnected
```

Clients and databases:

```
Waiting for a server to accept us...
Tell me your name, please!
Kate
Hello, Kate! This is the server. What is the room number?
1
Successfully registered with a name Kate-260 in the room 1.
Rules:
You should enter such numbers before your input depending on the purpose:
* 1 - simple message
* 2 - change room: 2 ROOM_NUMBER
* 3 - send file: 3 PATH
Tolya-296: hello
olena-308: hello Tolik!
1 hi guys
1 how are you?
olena-308: good
Tolya-296: not so good, i am tired....
Tolya-296 is leaving the room
olena-308 is leaving the room
2 2
You successfully entered the desired room!
3 "C:\Users\marga\Downloads\Practical assignment 2.pdf"
File was sent!
olena-308: let's end the converation
olena-308 is leaving the room
Bunya-276 is leaving the room
exit
```

```
Waiting for a server to accept us...
Tell me your name, please!
olena
Hello, olena! This is the server. What is the room number?
1
Successfully registered with a name olena-308 in the room 1.
Rules:
You should enter such numbers before your input depending on the purpose:
* 1 - simple message
* 2 - change room: 2 ROOM_NUMBER
* 3 - send file: 3 PATH
Tolya-296 joined
Kate-260 joined
Tolya-296: hello
1 hello Tolik!
Kate-260: hi guys
Kate-260: how are you?
1 good
Tolya-296: not so good, i am tired....
Tolya-296 is leaving the room
2 2
You successfully entered the desired room!
3 "C:\Users\marga\Downloads\game_engine_architecture_3rd.pdf"
File was sent!
Kate-260 joined
Kate-260 wants to send a file of size 200741 bytes. Do you accept it? It's name is Practical assignment 2.pdf
no
1 let's end the converation
2 7
Invalid room number!
2 4
You successfully entered the desired room!
```

```
C:\Margo\Parallel and clien-se
Waiting for a server to accept us...
Tell me your name, please!
Tolya
Hello, Tolya! This is the server. What is the room number?
6
Choose another room number:
1
Successfully registered with a name Tolya-296 in the room 1.
Rules:
You should enter such numbers before your input depending on the purpose:
* 1 - simple message
* 2 - change room: 2 ROOM_NUMBER
* 3 - send file: 3 PATH
Kate-260 joined
1 hello
olena-308: hello Tolik!
Kate-260: hi guys
Kate-260: how are you?
olena-308: good
1 not so good, i am tired....
2 2
You successfully entered the desired room!
olena-308 joined
olena-308 wants to send a file of size 13778524 bytes. Do you accept it? It's name is game_engine_architecture_3rd.pdf
yes
File received
Kate-260 joined
Kate-260 wants to send a file of size 200741 bytes. Do you accept it? It's name is Practical assignment 2.pdf
yes
File received
olena-308: let's end the converation
olena-308 is leaving the room
Bunya-276 is leaving the room
Kate-260 is leaving the room
exit
```

```
Waiting for a server to accept us...
Tell me your name, please!
Bunya
Hello, Bunya! This is the server. What is the room number?
2
Successfully registered with a name Bunya-276 in the room 2.
Rules:
You should enter such numbers before your input depending on the purpose:
* 1 - simple message
* 2 - change room: 2 ROOM_NUMBER
* 3 - send file: 3 PATH
1 i am alome...
Tolya-296 joined
olena-308 joined
olena-308 wants to send a file of size 13778524 bytes. Do you accept it? It's name is game_engine_architecture_3rd.pdf
yes
File received
Kate-260 joined
Kate-260 wants to send a file of size 200741 bytes. Do you accept it? It's name is Practical assignment 2.pdf
yes
File received
olena-308: let's end the converation
olena-308 is leaving the room
```

```
PS C:\Margo\Parallel and clien-server programming\Assignment 6\Client\database> tree /f
Folder PATH listing for volume OS
Volume serial number is 7A33-E6AD
C:.
├───Bunya
│   └───284
│           game_engine_architecture_3rd.pdf
│           Practical assignment 2.pdf
│
├───emma
│   └───172
├───Kate
│   └───236
├───olena
│   └───244
└───Tolya
    └───260
            game_engine_architecture_3rd.pdf
            Practical assignment 2.pdf
```

```
PS C:\Margo\Parallel and clien-server programming\Assignment 6\Server\database> tree /f
Folder PATH listing for volume OS
Volume serial number is 7A33-E6AD
C:.
    game_engine_architecture_3rd.pdf
    Practical assignment 2.pdf

No subfolders exist
```

Testing server unexpected termination:

```
Waiting for a server to accept us...
Tell me your name, please!
emma
Hello, emma! This is the server. What is the room number?
3
Successfully registered with a name emma-240 in the room 3.
Rules:
You should enter such numbers before your input depending on the purpose:
* 1 - simple message
* 2 - change room: 2 ROOM_NUMBER
* 3 - send file: 3 PATH
The server terminated unexpectedly!

C:\Margo\Parallel and clien-server programming\Assignment 6\Client\x64\Debug\Client.exe (process 41644) exited with code
 1 (0x1).
Press any key to close this window . . .
```

```
Server listening on port 12345
Client with socket 240: Hello, server! This is client:)
Client emma joined ROOM 2

C:\Margo\Parallel and clien-server programming\Assignment 6\Server\x64\Debug\Server.exe (process 49096) exited with code
 -1073741510 (0xc000013a).
Press any key to close this window . . .
```

Testing client graceful exit:

```
Server listening on port 12345
Client with socket 272: Hello, server! This is client:)
Client Roma joined ROOM 0
Client Roma left ROOM 0
Roma-272 disconnected
```

```
Waiting for a server to accept us...
Tell me your name, please!
Roma
Hello, Roma! This is the server. What is the room number?
1
Successfully registered with a name Roma-272 in the room 1.
Rules:
You should enter such numbers before your input depending on the purpose:
* 1 - simple message
* 2 - change room: 2 ROOM_NUMBER
* 3 - send file: 3 PATH
exit
Bye client!

C:\Margo\Parallel and clien-server programming\Assignment 6\Client\x64\Debug\Client.exe (process 11116) exited with code
 0 (0x0).
Press any key to close this window . . .
```

From the user's perspective:

I designed and implemented the best chat application you could ever imagine. It works smoothly and provides the best user experience since all output is coloured accordingly. Hence, the user can easily understand what is happening on the screen.

The system supports up to **20 concurrent clients** and divides them into **4 separate rooms** where they can communicate independently. These numbers can be easily adjusted as one wishes.

Client interaction follows a structured protocol. When a client connects, they send a greeting message, and the server responds by requesting the client's username. Once the name is provided, the server asks for the desired room number. After joining the selected room, clients can exchange messages, request file transfers, switch rooms, or disconnect at any time. More details below.

The clients can communicate and send each other different messages if they stay in the same room. The user should specify the number 1 and the message to send something. The clients can easily change the room if the former one becomes annoying. To do that, the user must type the number 2, followed by the room number he/she wishes to enter. The user cannot send messages until he/she receives a confirmation from the server about successfully entering the desired room. Other clients always receive a relevant message if a newbie joins or someone leaves.

The user can send a file to the neighbors. Alternatively, the user gets a question at any point in time asking if he/she wants to receive a file. The user must type "YES," "Y," "YEAH," or "YEP" (it can be in both lower and upper cases) to receive or some other word for no. While the file is sent, the user can neither receive nor send messages.

If the user wants to exit, he/she must type “exit” either in lower or upper cases.

Conclusion:

Use my app in everyday life and enjoy it.