Project Case	
OOP using Java Project	
Periode Berlaku Semester Ganjil 2023/2024 ***Valid on*** *Odd Year 2023/2024*	**Software Laboratory Center** **Assistant Recruitment 24-1**

Note: Please focus on the main logic and main features!

(Splash screen and design are not scored)

Soal

Case

EF-RPC

EF is tired of mainstream game-type cases, so EF-RPC is born. EF-RPC is a master-slave **messaging system** designed to allow interaction in a simulated distributed environment. Central to its design is the **Master Application**, which oversees and upholds system integrity, tracking active clients, and maintaining a smooth communication flow. Clients are the heart of user interaction - enabling **message sharing**, **server switching**, and **private messaging**.

Moving away from standard game scenarios, this system promotes a unique way for clients to communicate in a simulated mutable multi-server environment, closely mimicking real-world chat-based systems.

EF encourages you to focus on building a system that achieves seamless, fault-tolerant, and straightforward communication. Emphasizing usability and smooth transactional operations between the master and slaves, this case invites you to reimagine messaging protocols.

Remember, **the use of regex in your work might negatively impact your score and aesthetics don't count**. Concentrate on your foundational logic and **use proper OOP concepts** such as **abstraction**, **encapsulation**, **polymorphism**, and **inheritance**. Ensure that your multi-thread design is **resilient** to **thread interference**.

- **Home Page**
  - This menu contains 3 menus, which are **Start as Master**, **Start as Client**, and **Exit**.
  - **Prompt** user to **input chosen menu**. **Validate** the input must be **between 1 and 3 inclusively**.

Figure 1. Home Page.

- If the user chooses **Start as Master** (**Menu 1**), then:
  - The program will check for a .tmp file called '**checkup.tmp**' inside of the '**datastore**' folder to **check** whether **there's another instance of the master** server running.
  - If the '**checkup.tmp**' file exists, then the program will **exit**.

```
Checking checkup file
Another master is running. Shutting down...
```

Figure 2. 'checkup.tmp' exists.

  - If not, then the program will create the '**checkup.tmp**' file inside of the '**datastore'** folder to mark that an instance of the master server is running. The file **must automatically be deleted** when the **program exits**.
  - Then the program will read all **.txt** files inside of the '**pubSub**' directory every **100 milliseconds** to **read packets** sent by the client-server to the master server.
  - The read packets will then be processed accordingly. Please use the **abstraction** and **polymorphism concepts** to construct these packet listeners. You can adjust the exact packet format to your liking.
    - Received Packets:
      - **Keep Alive** Packet: This packet will be used to **keep track of live clients**. Upon receiving this packet, update the client's keep alive timeout so that they're not considered dead. If a client doesn't send a keep-alive packet within **1000 milliseconds**, then the client will be considered **dead** and **will be removed from the client list**.
      - **Switch** Packet: This packet will be used when the **client requests to move** between channels. Upon receiving this packet, **check** whether the **requested channel exists**, if **yes** then **switch the client's channel** to the requested channel, **else** send a packet to tell the client that the requested channel does not exist.

- **WhereAmI** Packet: This packet will be used when the **client requests to get the current channel** that they're in. Upon receiving this packet, **send a packet to tell the sender client their current channel**.
- **Public Message** Packet: This packet will be used when the **client sends a normal message**. Upon receiving this packet, **broadcast the public message** to all clients **inside of the same channel** as the packet sender.
- **Private Message** Packet: This packet will be used when the **client sends a private message** to another client. Upon receiving this packet, the master will **check** whether the **target client is logged in**, if **yes**, then **send the private message to the target client**, **else** send a packet to tell the sender client that the target client is not found.
- **Login** Packet: This packet will be used when the **client sends a login request**. Upon receiving this packet, the **master** will **check whether the user is already registered** by checking the '**clients.txt**' file inside the '**datastore**' directory. If **yes**, then **the client will be added to the list of connected clients** and be **put inside of the default channel** but if the client is **already inside of the connected client list**, **send a packet** to tell the client that they're **already logged in**, **else if the credential is not valid**, the master will **send a packet** to tell the client that the inputted credentials is invalid.
- **Register** Packet: This packet will be used when the **client sends a register request**. Upon receiving this packet, the **master** will then **check whether the user is registered** by checking the '**clients.txt**' file inside the '**datastore**' directory. If **yes**, then the master will **send a packet** to tell the client that the **account is already registered**, **else** the master will **add** the client to the **list of connected clients**, and then **write a new entry** in the '**clients.txt**' file with the format of:

(CLIENT_NAME)-(PASSWORD)

- Sent Packets:
  - **Acknowledge** Packet: This packet will be sent as a **response to the keep-alive packet**.
  - **Generic** Packet: This packet will be sent as a **generic message** to the client.

- **Auth** Packet: This packet will be sent as a **success/fail message and its reason** to the client whenever they're **registering a new account or logging in**.
- **Public Message** Packet: This packet will be sent **to all other clients in the same channel** as the sender client as a **public message**.
- **Private Message** Packet: This packet will be sent **to the target client** as a **private message**.
- **Logout** Packet: This packet will be sent to clients whenever the **master reboots** and/or **needs the client to re-enter their login credentials**.

- If there's any packet sent by a client, but the client is **not** on the connected client list, force the client to **log out**.

- The program will also wait for the **user's input**.
  - If the user types **nothing** or any **unknown commands**, then the program will **show** an '**Unknown command**' **message** to the user.

```
Done (0.023s)! For help type "help" or "?"
--------------------------------------------------------------------------------
>>
Unknown command. For help type "help" or "?"
```

Figure 3. Master's unknown command message.

  - If the user types '**help**' or '**?**' then the **full help message** will be shown. The full help message will **show all available commands**, their **aliases**, and their **descriptions**.

```
--------------------------------------------------------------------------------
>> help

Available commands:

  > stop, exit, quit, end
    Stops the application.

  > help, ?
    Shows this help message.

  > channellist, channels, chlist
    Shows a list of all channels.

  > clientlist, clients, clist
    Shows a list of all connected clients.

  > addchannel, addchan, addch
    Adds a channel.

  > removechannel, removechan, removech
    Removes a channel.
```

Figure 4. Master's full help message.

- If the user types ‘**stop**’ or ‘**exit**’ or ‘**quit**’ or ‘**end**’ then the program will **exit**.

```
>> exit
Stopping application...

Shutting down MasterDatastore
Shut down 1 executor services
Shutting down PubSubManager
Shut down 1 runnable(s)
```

```
o---------------------------------------------o
|      Breaking and Overcoming Challenges     |
|    Through Courage Hardwork and Persistence |
o---------------------------------------------o
```

Figure 5. Master exiting.

- If the user types ‘**channelist**’ or ‘**channels**’ or ‘**chlist**’ then the program will **show all available channels** and the **count of clients** in that channel. By **default**, there must be **one channel** called ‘**lobby**’.

```
>> chlist
Channels:
  > lobby (1 connected)
```

Figure 6. Master ‘chlist’ command.

- If the user types ‘**clientlist**’ or ‘**clients**’ or ‘**clist**’ then the program will **show all connected clients** and **the channel that the client is on**.

```
>> clist
Connected clients:
  > EF [lobby]
```

Figure 7. Master 'clist' command.

- If the user types '**addchannel**' or '**addchan**' or '**addch**' then the program will **create and add a new channel** for the clients to connect. Make sure that there are **no duplicate channels**. The exact usage of the command would be (command) <channel name>. **If the user does not supply a channel name**, then the program will **show the correct command usage**.

```
>> addchannel
Usage: /addchannel <channel

>> addchannel slc
Channel added.

>> addchannel slc
Channel already exists.
```

Figure 8. Master 'addchannel' command.

- If the user types '**removechannel**' or '**removechan**' or '**removech**' then the program will **remove the selected channel**. Make sure that **the selected channel exists**, there is **no client in the selected channel**, and **the selected channel is not the default** 'lobby' **channel**. The exact usage of the command would be (command) <channel name>. **If the user does not supply a channel name**, then the program will **show the correct command usage**.

```
>> removechannel
Usage: /removechannel <channel>

>> removechannel lobby
Channel is a default channel.

>> removechannel slc
Channel is not empty.

>> removechannel slc
Channel removed.
```

Figure 9. Master 'removechannel' command.

- If the user chooses **Start as Client** (**Menu 2**), then:
  - The client will create a **.txt** file inside of the '**pubSub**' directory with a **randomly generated name** in the format of:

CLI-[0-9][0-9][0-9][0-9][0-9].txt

- Then the client will start **reading** its pubSub file every **100 milliseconds** to read incoming packets from the master client.
- The read packets will then be processed accordingly. Please use the **abstraction** and **polymorphism concepts** to construct these packet listeners. You can adjust the exact packet format to your liking.
  - Received Packet:
    - **Acknowledge** Packet: This packet will be used as a response to a keep-alive packet. This packet does **nothing**.
    - **Auth** Packet: This packet will be used as the **response of a login or a register packet**. Upon receiving this packet, if the packet says that the client is **allowed** to log in, **save their state to be logged in** so that they can execute other commands and **start sending a keep-alive packet** every **500 milliseconds**, **else** tell the client to **re-login or re-register**.

```
Authentication: FAILED - Client already registered.
Authentication: FAILED - Invalid credentials.
Authentication: FAILED - Client already logged in.
```

Figure 10. Client auth failure.

```
Authentication: SUCCESS - Client logged in successfully.
```

Figure 11. Client auth success.

    - **Generic** Packet: This packet is a versatile packet that is usually used for **simple response messages** such as to respond to the '**WhereAmI Packet'**, a denied and accepted '**Switch Packet**', and a denied '**Private Message Packet**'. Upon receiving this packet, **display** the received message.

```
You are in channel lobby

Channel does not exist!
```

Figure 12. Client generic message.

    - **Logout** Packet: This packet will be used when the master forces the client to log back in. Upon receiving this message, **update the client's state to be logged out** so that they must log back in.

```
You have been logged out!
```

Figure 13. Client logged out.

- **Private** Message Packet: This packet will be used when there is an **incoming private message** to the client. Upon receiving this message, **format** the received message in this format:

[(sender) -> YOU]: (message)

```
[slc -> YOU]: Hello!
```

Figure 14. A client is receiving a private message.

- **Public** Message Packet: This packet will be used when there's an **incoming public message**. Upon receiving this message, **format** the received message in this format:

(sender): (message)

```
slc: asd
slc: Hello World!
```

Figure 15. A client is receiving a public message.

- Sent Packet:
  - **Keep Alive** Packet: This packet will be sent **every 500 milliseconds** to indicate that the client is still alive.
  - **Register** Packet: This packet will be sent whenever the client **registers a new account**.
  - **Login** Packet: This packet will be sent whenever the client is **logging in**.
  - **Public Message** Packet: This packet will be sent whenever the client is **sending a public message**.
  - **Private Message** Packet: This packet will be sent whenever the client is **sending a private message** to another client.
  - **Switch Request** Packet: This packet will be sent whenever the client requests **to move to another channel**.

- **Where Am I** Packet: This packet will be sent whenever the client is **requesting to get the channel they're currently on**.

- **After processing** the read packets, **clear the txt file** to save storage and avoid duplicate packet reading.
- The program will also wait for the **user's input**.
  - Commands in the client **must begin** with a '**/**'
  - If the user **does not input anything**, **only** a '**/**', or an **unknown command**, then the program will **show** an '**Unknown command**' **message** to the user.

```
>>
Unknown command. For help type "help" or "?"
>> /
Unknown command. For help type "help" or "?"
>> /asd
Unknown command. For help type "help" or "?"
```

Figure 16. The client's unknown command message

  - If the user types **without** a '/' in the **beginning**, then the program will assume that the user is sending a **public message**. **Ensure** that the user's state is **logged in before sending** a public message, if yes, **send a packet** to the **master**, else, **display** a message.

```
Please login first!
>>
```

Figure 17. The client sends a message before logging in.

```
slc: Hello world!
```

Figure 18. A client sending a public message.

  - If the user types '/**help**' or '/**?**' then the **full help message** will be shown. The full help message will **show all available commands**, their **aliases**, and their descriptions.

```
Available commands:

  > /stop, /exit, /quit, /end
    Stops the application.

  > /register, /reg
    Registers the client with the master.

  > /login, /l
    Logs the client in.

  > /pm, /privmsg, /privatemessage, /msg
    Sends a private message to a client.

  > /whereami, /checkchannel, /channel
    Shows the current channel you are in.

  > /switch, /goto
    Switches to another channel.

  > /help, /?
    Shows this help message.
```

Figure 19. Client's help message.

- If the user types '/**stop**' or '/**exit**' or '/**quit**' or '/**end**' then the program will **exit**.

```
Stopping application...

Shutting down PubSubManager
Shut down 1 runnable(s)
```

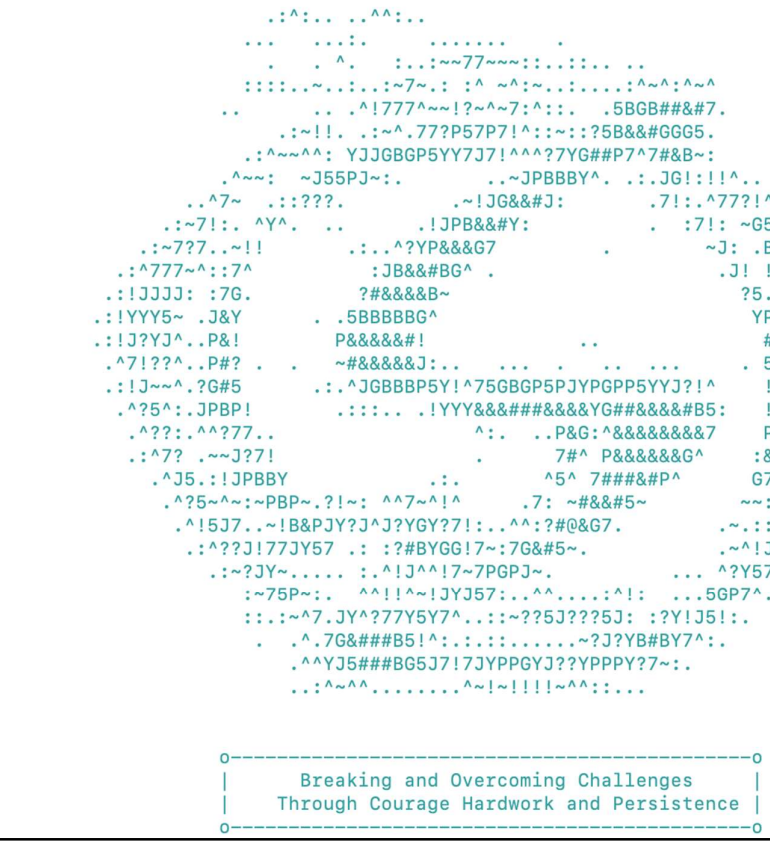

Figure 20. Client exiting.

- If the user types '**/register**' or '**/reg**' then the program will **send a packet** to the master that the client is **requesting to register a new account**. The exact usage of the command would be /(command) <username> <password> <confirm password>. **If the user does not supply enough arguments**, then the program will **show the correct command usage**. Also, make sure that the **password** is **the same** as the **confirmed password**, otherwise **shows** an error message. Lastly, make sure that the user's state **must be logged out** to execute this command.

```
Usage: /register <name> <password> <confirmPassword>

Passwords do not match.
```

Figure 21. Client 'register' command.

- If the user types '**/login**' or '**/l**' then the program will **send a packet** to the master that the client is **requesting to login**. The exact usage of the command would be /(command) (username) (password). **If the user does not supply enough arguments**, then the program will **show the correct command usage**. Make sure that the user's state **must be logged out** to execute this command.

```
Usage: /login <name> <password>
```

Figure 22. Client 'login' command.

- If the user types '**/pm**' or '**/privatemessage**' or '**/msg**' or '**/privatemsg**' then the program will **send a packet** to master that the client is **sending a private message**. The exact usage of the command would be /(command) (targetClientName) (message). **If the user does not supply enough arguments**, then the program will **show the correct command usage**. Make sure that the user's state **must be logged in** to execute this command.

```
Usage: /msg <clientName> <message>
```

Figure 23. Client 'message' command.

```
[YOU -> 241]: Hello!
```

Figure 24. Client messaging another client.

- If the user types '**/whereami**' or '**/checkchannel**' or '**/channel**' then the program will **send a packet** to the master that the client is requesting to **get their current channel**. The exact usage of the command would be /(command). Make sure that the user's state **must be logged in** to execute this command.
- If the user types '**/switch'** or '**/goto'** then the program will **send a packet** to the master that the client is requesting to **switch their current channel**. The exact usage of the command would be /(command) (targetChannelName). **If the user does not supply enough arguments**, then the program will **show the correct command usage**. Make sure that the user's state **must be logged in** to execute this command.

```
Usage: /switch <channelName>
```

Figure 25. Client 'switch' command.

- If the user chooses **Exit** (**Menu 3**), then the program will exit.

```
                    .:^:.. ..^^:..
                 ...   ...:.     .......    .
                   .   . ^.   :..:~~77~~~::..::.. ..
                 ::::..~..:..:~7~.: :^ ~^:~..:....:^~^:^~^
              ..      .. .^!777^~~!?~^~7:^::.  .5BGB##&#7.
                   .:~!!. .:~^.77?P57P7!^::~::?5B&&#GGG5.
                 .:^~~^^: YJJGBGP5YY7J7!^^^?7YG##P7^7#&B~:
              .^~~:  ~J55PJ~:.       ..~JPBBBY^. .:.JG!:!!^..
           ..^7~  .::???.         .~!JG&&#J:       .7!:.^77?!^..
        .:~7!:. ^Y^.  ..      .!JPB&&#Y:          .  :7!: ~G5?!:.
     .:~7?7..~!!       .:..^?YP&&&G7         .        ~J: .BP7J!:.
   .:^777~^::7^          :JB&&#BG^ .                  .J! !YYYJ?^.
  .:!JJJJ: :7G.         ?#&&&&B~                        ?5..^!~~!!:.
 .:!YYY5~ .J&Y      . .5BBBBBG^                          YP  :?J!~!^.
 .:!J?YJ^..P&!        P&&&&&#!             ..              #7 .^55?^~:
  .^7!??^..P#? .  .   ~#&&&&&J:..   ...  .   ..  ...     . 5#..:J57~!:.
  .:!J~~^.?G#5      .:.^JGBBBP5Y!^75GBGP5PJYPGPP5YYJ?!^    !&:.:??~?~:.
   .^?5^:.JPBP!       .::::.. .!YYY&&&###&&&&YG##&&&&#B5:   !&:.^~!5J~.
    .^??:.^^?77..                  ^:.  ..P&G:^&&&&&&&&7    PG ~7:JP7:.
    .:^7? .~~J?7!                  .      7#^ P&&&&&&G^    :&^ 7?5?!:.
      .^J5.:!JPBBY            .:.       ^5^ 7###&#P^      G7..7G5?!:.
      .^?5~^~:~PBP~.?!~: ^^7~^!^     .7: ~#&&#5~        ~~:.:PY~:.
       .^!5J7..~!B&PJY?J^J?YGY?7!:..^^:?#@&G7.        .~.::^Y!^.
        .:^??J!77JY57 .: :?#BYGG!7~:7G&#5~.           .~^!J5!:.
          .:~?JY~..... :.^!J^^!7~7PGPJ~.          ... ^?Y57:.
            :~75P~:.  ^^!!^~!JYJ57:..^^....:^!:  ...5GP7^.
            ::.:~^7.JY^?77Y5Y7^..::~??5J???5J: :?Y!J5!:.
             .  .^.7G&###B5!^:.:.::......~?J?YB#BY7^:.
                .^^YJ5###BG5J7!7JYPPGYJ??YPPPY?7~:.
                ..:^~^^........^~!~!!!!~^^::...


          o-------------------------------------------o
          |      Breaking and Overcoming Challenges     |
          |    Through Courage Hardwork and Persistence |
          o-------------------------------------------o
```

Figure 26. Program exiting.

- **Diagrams**

To **aid** you in how the master-client **communication protocol** works in this application, you can **refer to the diagrams below**. The diagrams are made **based on John Satzinger's theory**.

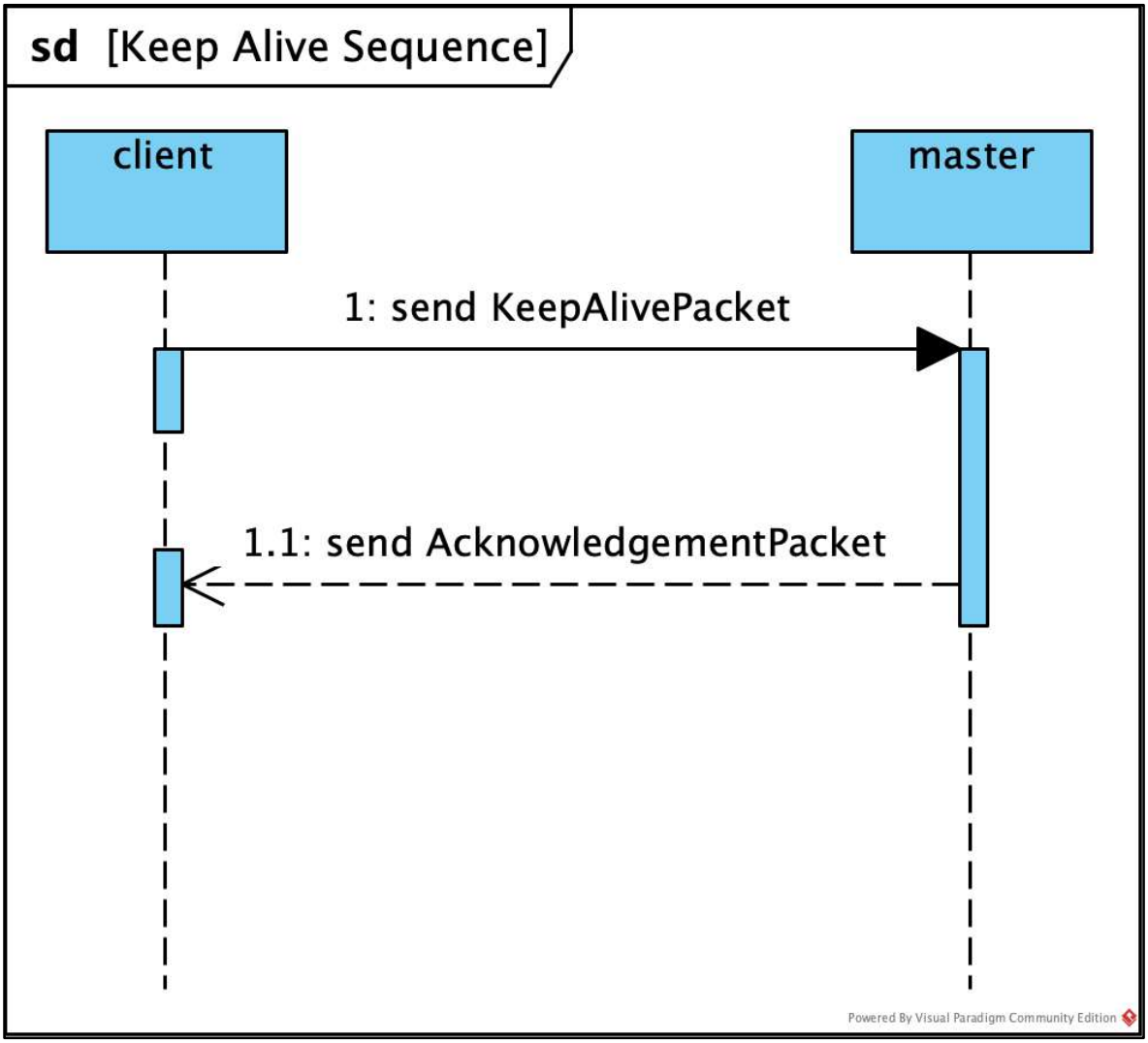

Figure 27. Keep Alive Messaging Protocol.

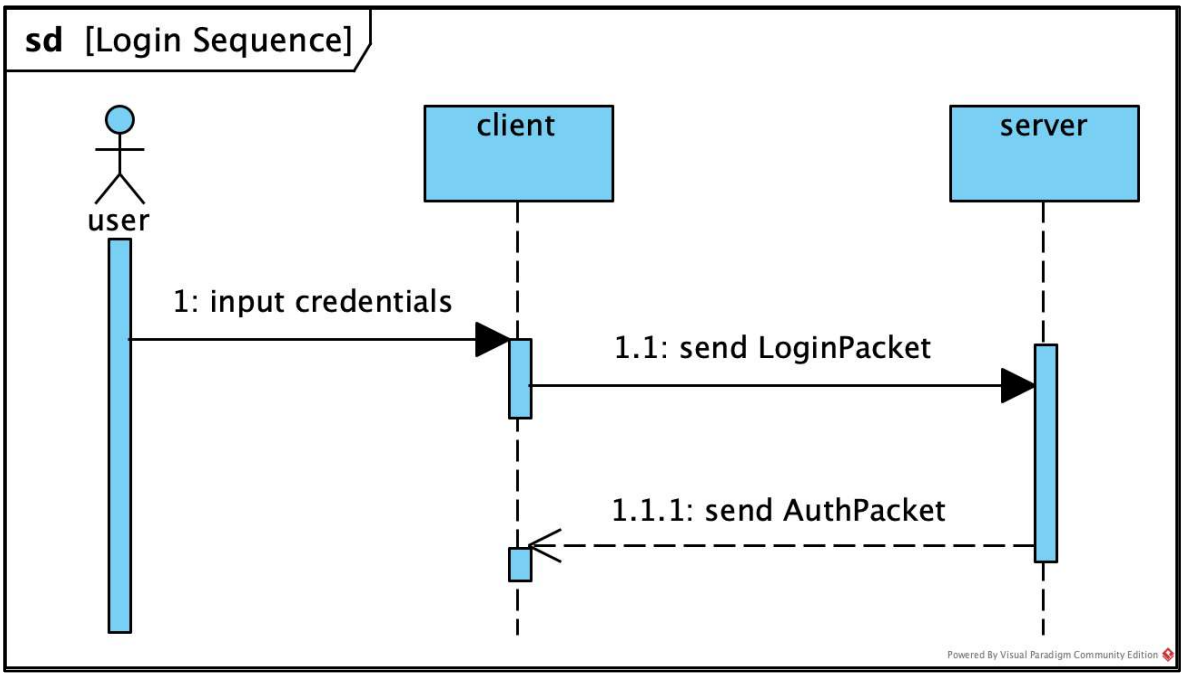

Figure 28. Login Messaging Protocol.

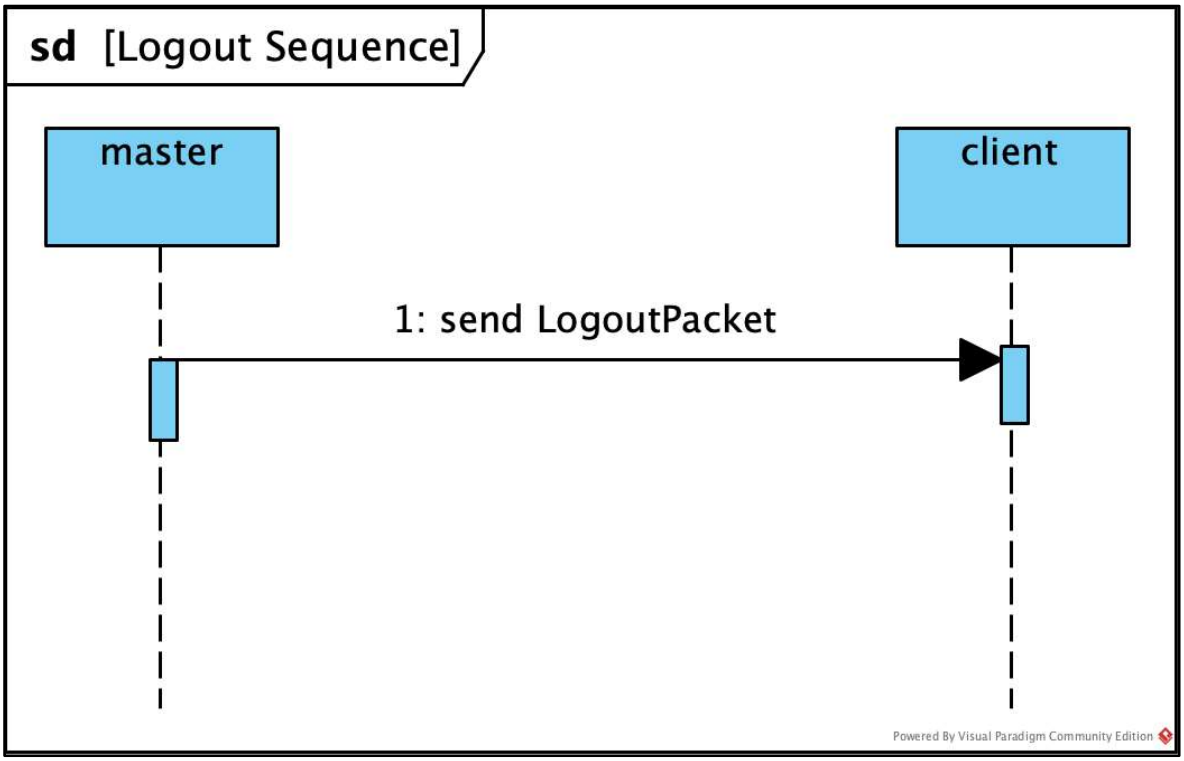

Figure 29. Logout Messaging Protocol.

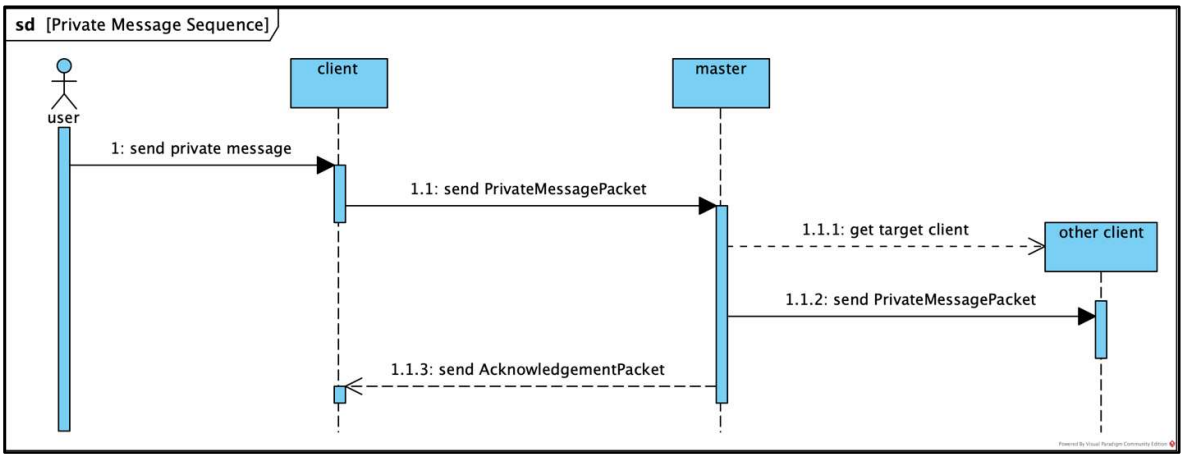

Figure 30. Private Messaging Communication Protocol.

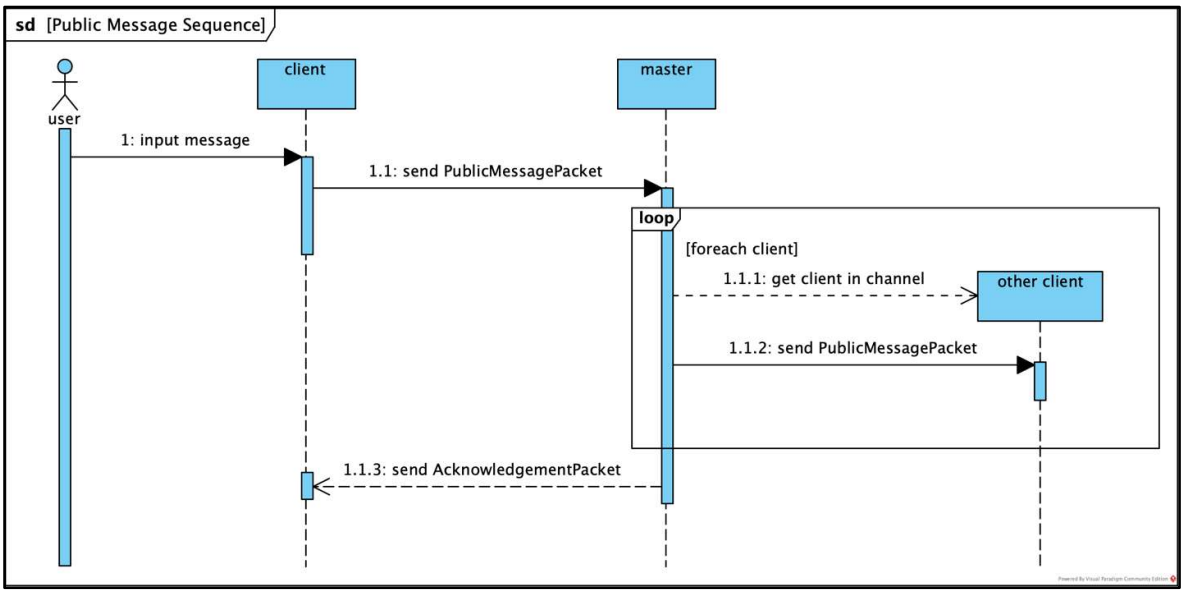

Figure 31. Public Messaging Communication Protocol.

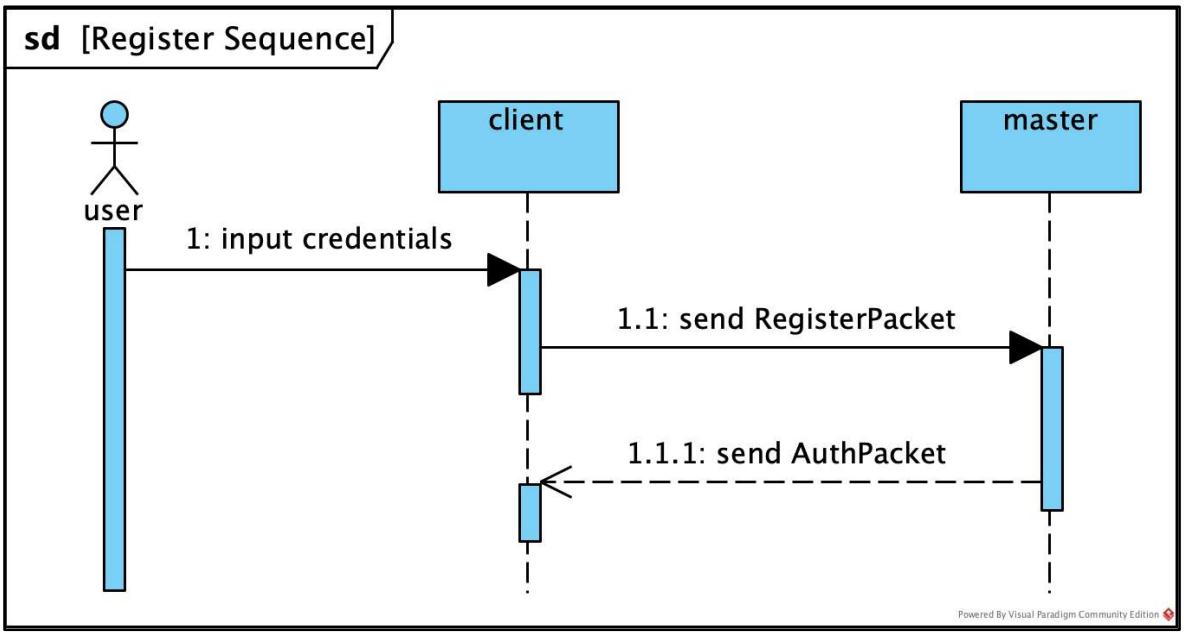

Figure 32. Register Communication Protocol.

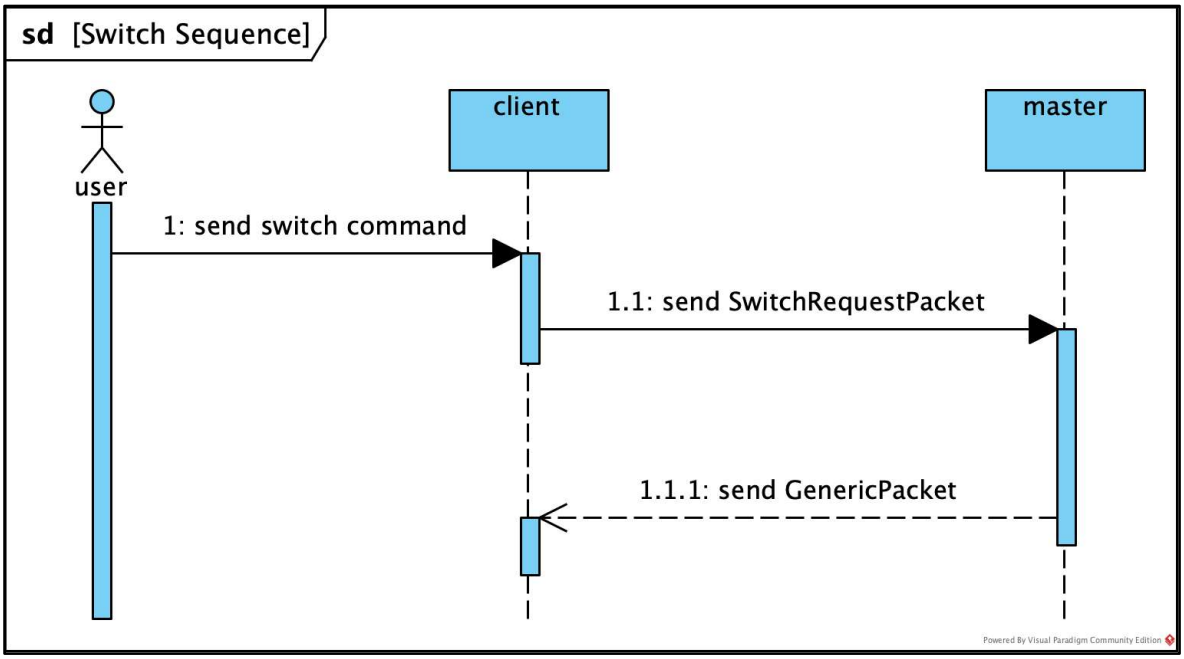

Figure 33. Switch Communication Protocol.

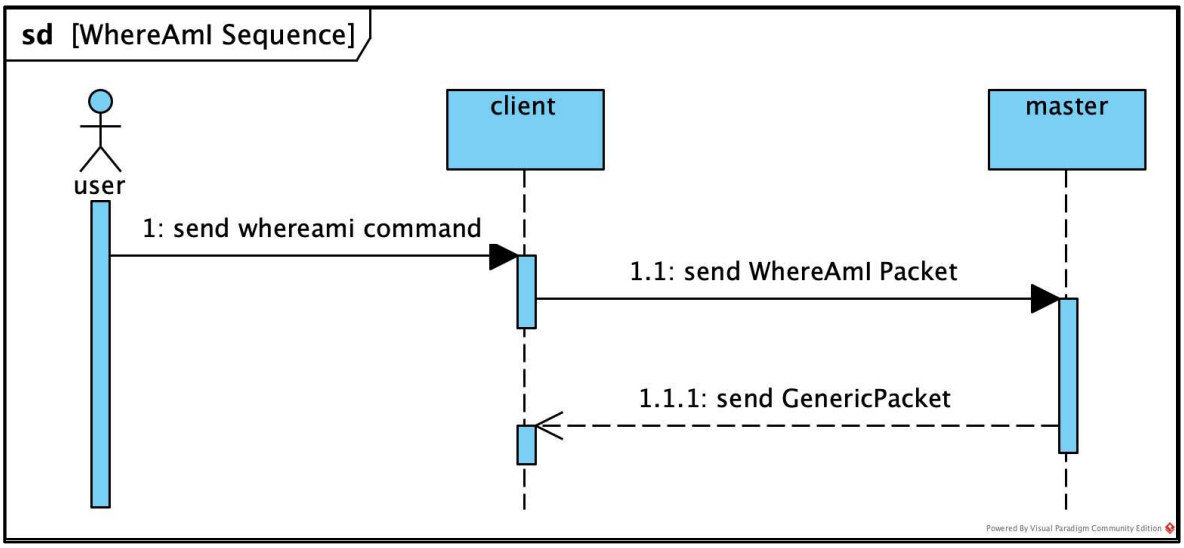

Figure 34. WhereAmI Communication Protocol.

Please run the JAR file to see the sample program.