28112-lab2-S-MultipleDistributionCh

Bekhruz Suleymanov
a81060bs
24th Of March 2023

A protocol design for a chatroom application that allows users to send messages to each other, block and unblock other users, and quit the chatroom. The design includes several commands to implement these functionalities. The use of the '@' symbol to highlight a command variable in the code improves the quality of the code and allows for better differentiation between commands and messages. The design includes error handling to prevent users from performing invalid actions, such as blocking themselves or blocking a user who is not connected to the chatroom. The registration step ensures that users are registered before they can use the chatroom. Overall, the design is intended to provide a user-friendly, efficient, and secure chatroom experience.

Compiled on March 24, 2023

1 Introduction

The protocol design is for a chatroom application that allows users to send messages to each other, block and unblock other users, and quit the chatroom. The design includes several commands to implement these functionalities. The use of the '@' symbol to highlight a command variable in the code improves the quality of the code and allows for better differentiation between commands and messages. The design includes error handling to prevent users from performing invalid actions, such as blocking themselves or blocking a user who is not connected to the chatroom. The registration step ensures that users are registered before they can use the chatroom. Overall, the design is intended to provide a user-friendly, efficient, and secure chatroom experience.

Pseudo Code

```
onMessage:
    rawMessage = receive message
    command = extract command from rawMessage
    Message = extract message from rawMessage
    if registered == True:
        if command == "@everyone":
            send message to all connected users

        elif command == "@":
            list all connected users

        elif command starts with "@block" (followed by username):
            check that the user is not blocking himself
            check if the user has provided username as argument
            check that the username exists
            check if the user has been already blocked
            if the user has not been blocked:
                block the user

        elif commands start with "@unblock":
            check if the user is blocked
            if blocked:
                unblock
        elif command == "quit":
            close connection

        elif command starts with "@":
            check if sender or receiver has blocked one another.
            if no:
                recipient = extract recipient from command
                if recipient is a connected user:
                    send message to recipient
                else:
                    send error message to user
                    ask the user to type the command again
        else:
            send error message to user, saying that the command that he has typed is wrong.
    else:
        print(Register first!)
        registerThisUser()
```

List of Commands

The above protocol design is for a chatroom application that allows users to send messages to each other, block and unblock other users, and quit the chatroom. The design includes several commands to implement these functionalities.

Why use '@' for command variables?

I think it removes unnecessary ambiguity as a user can easily differentiate between a message and a command.

List of commands

The following commands are available in the chatroom application:

- `@everyone`: Send a message to all connected users.
- `@username`: Send a private message to a specific user.
- `@`: List all the connected users in the chatroom.
- `@block username`: Block a specific user.
- `@unblock username`: Unblock a specific user.
- `@quit`: Leave the chatroom.
- `@help`: Display a list of all available commands.

The protocol design also includes error checking to ensure that users cannot perform invalid actions. For example, users cannot block themselves or block a user who is not connected to the chatroom. Additionally, the design includes a registration step to ensure that users are registered before they can use the chatroom.

Bugs and Issues Found in the Current Program

The current program contains several bugs that could potentially cause issues in future development. In this report, we will highlight and discuss five of these bugs.

1. Firstly, we have noticed that the registration process allows users to input multiple parameters for their username. However, only the first parameter is used as the username. For instance, if a user registers with the name "Jake Jr.", only "Jake" will be passed as the username. This inconsistency in the registration process could lead to confusion and errors in the system.
2. Secondly, we would like to draw attention to the fact that the program currently lacks a robust error-handling system. In the event of an internal error occurring, users may not be notified that the server is down. This could potentially result in significant problems, including a loss of customer trust.
3. Thirdly, the current implementation of the server uses a dictionary to store usernames. While this is a suitable implementation for small-scale businesses, it may result in a slow registration process as the number of users grows rapidly.
4. Fourthly, we have found that the 'Client' class, which is designed to intercept incoming messages if another user is in the middle of raw input, does not function as intended. Despite attempts to fix this issue by introducing a queue, the solution was deemed unsatisfactory due to the potential for messaging to become mechanical and non-instant.
5. Lastly, we have not tested the server's capacity, which could potentially impact its performance. While theoretically, the capacity should not affect the server's performance, this is an area of concern. We have attempted to create a bash script to test the server's capacity. However, we were unable to run it due to the approaching deadline.

```
#!/bin/bash

#Define the number of times to run the script
for i in {1..100}
do
# Add 100 usernames for registration
usernames=("user1" "user2" "user3" "user4" "user5" "user6" "user7" "user8" "user9" "user10"
"user11" "user12" "user13" "user14" "user15" "user16" "user17" "user18" "user19" "user20"
"user21" "user22" "user23" "user24" "user25" "user26" "user27" "user28" "user29" "user30"
"user31" "user32" "user33" "user34" "user35" "user36" "user37" "user38" "user39" "user40"
"user41" "user42" "user43" "user44" "user45" "user46" "user47" "user48" "user49" "user50"
"user51" "user52" "user53" "user54" "user55" "user56" "user57" "user58" "user59" "user60"
"user61" "user62" "user63" "user64" "user65" "user66" "user67" "user68" "user69" "user70"
"user71" "user72" "user73" "user74" "user75" "user76" "user77" "user78" "user79" "user80"
"user81" "user82" "user83" "user84" "user85" "user86" "user87" "user88" "user89" "user90"
"user91" "user92" "user93" "user94" "user95" "user96" "user97" "user98" "user99" "user100")


# Loop through the usernames and register them
for username in "${usernames[@]}"
do
    python myClient.py localhost 8080
    python myClient.py -u "$username"
done
```