Distributed Systems Report

You should provide a description of your protocol design. It should include a list of the commands, what they do and the rationale behind each design decision. You should also write pseudocode that describes how your server will handle the various commands, in what order, etc.

Protocol Overview

There are two parts for this protocol to run, the server and the client. Once the server is up and running, a client is able to join and connect to the server. Multiple clients can join the server at the same time. Upon connection the user is prompted with a welcome and a message advising the user to register their name with the server, so that they may be able to use the messaging system. They can alternatively use HELP or QUIT commands too, as they do not need to register to use these commands. Once the user registers using a valid username they are welcomed and can use other commands in the system. The user is able to communicate with other users who are also connected to the server, either through private message or messaging all users at once. The messages that are sent from the server to the receiving user are done so by parsing the input by the sending client, and using the clients socket object to send the message. These socket objects act as a unique ID for each user and can be used to identify them and send messages to others, allowing users to communicate with each other.

This works with the client side taking the user's input and sending it over to the server. The server is then responsible for processing any commands but also ensures that the user has entered valid input. The server will validate each command to check for specific parameters and will send a message back to the user if incorrect input has been given. The server also keeps a server log of all requests on the system, including user commands, the number of clients connected, new users who register, and when a user disconnects. Through this design the server controls and processes almost everything and is able to give sensible output back to the user based on what has been requested. As I have learnt from the lectures this client-server interaction is known as request-reply behaviour.

After all clients are done using the system they can QUIT using the command and exit. The server will acknowledge when each client is disconnected and once all users have quit, the server will remain active, in case anyone else joins again.

Design Decisions

One key design decision I made was allowing only certain commands available to users who have not registered. I decided it seemed user friendly to allow users to be able to access HELP/HELPLIST to see available commands or information about them. This made sense as a new user may not be familiar with the commands so it would be useful for them to understand more of how the system works before registering a user name.

One command that the user can do is see all active users on the system at a given time, so that they can see who they can message. However I also included another command which will allow the user to check if a specific user is active, as there may be a large amount of users connected to the server. Instead of searching all of them they can check if the one user they want to message is active and message them if they are.

I have also decided to include a command called, HIDE. This command lets a user not be seen as an active user on the system, whilst still being able to receive messages from other users. I included this as a user may want to receive messages but not be able to chat, so other users would not try message them, until they become unhidden. This approach is taken from realistic messaging systems, such as discord and PSN messaging, therefore it seemed a valid and useful idea to implement.

A decision I made to not implement something was chat history of users. Although it would be useful to keep a track of this, per user, I decided against it as given the current implementation of sending individual messages across to multiple users, to me it felt that it was not necessary. Instead the server just keep a log of all messages and command requests made by the users, rather than each user being able to see certain chats they are having with another user, as that would become quite messy in this implementation given the clients all interact to the server and other users through a single terminal.

Validation

Another decision was to do with the validation of the user input. The server handles various different validations mainly to ensure each command is entered correctly so that server can follow the correct appropriate action. The main validation is with regards to user registration.

Firstly I allow the user to only create a username with one word, no spaces, otherwise they are prompted by and error message to try again. The user cannot create a name that has already been chosen by another user, currently active on the system, and once a user is registered they cannot try register again. The server will display an appropriate message to each of these situations. Once registered the user is also then allowed to use the additional commands, such as message users or see active users. This check is made by the server to ensure the user is registered before allowing them to perform these commands.

Command list

For my protocol there is many commands that the user Is able to enter each with a specific format, and parameters. Below is the list of commands, their explanation and their required format.

Command	**Purpose**	**Format**
'REGISTER'	Allows user to register a name on the system	'<REGISTER>' '<username>'
'MESSAGEALL'	Send message to all active users on system	'<MESSAGEALL>' '<message>'
'MESSAGE'	Send direct message to specified user	'<MESSAGE>' '<username>' '<message>'
'ALLUSERS'	Displays list of all active users	'<ALLUSERS>'
'HELPLIST'	Lists all commands and their format to user	'<HELPLIST>'
'HELP'	Information about commands and their purpose	'<HELP>'
'CHECK'	Lets the user check if a specific user is active on the system	'<CHECK>' '<username>'
'HIDE'	Lets the user hide, so they are not seen as active on the system by other users	'<HIDE>'
'QUIT'	Allows the user to exit the program	'<QUIT>'

Command order

The order in which the users input is checked for a command is crucial to allowing the user to correctly perform commands. The protocol uses multiple if statements to check user input and respond to the user accordingly. The first commands it checks for is the commands that are required to be registered before using, such as message and check users etc. This seemed like the most logical order as these commands will be frequently used by the users, meaning it would be more efficient to check for them first. This therefore will improve the performance and speed of the protocol as there will be less if statements checking for the commands. After this the register command is checked for as this seems like an appropriate place as it wont be called as frequently, as a user only registers once.

The next commands are the HELP, HELPLIST and QUIT commands, as these commands do not require a user to be registered to be used therefore again makes sense to check for these commands next. The check after this is a validation check to see if a user is not registered and tries to enter any command they are not allowed to use, this seemed sensible to include to prevent users from misusing the system, potentially leading to errors.

Pseudocode

Below shows the pseudocode I used to design my protocol. For the function onStart, I initialised my server variables which will be a counter for the number of clients connected and a dictionary holding the users name and instance (socket) as a key/value pair. It also declares that the server has started, which will be displayed in the server log. For onConnect and onDisconnect I have initialised user attributes and outputted the number of clients connected, which updates even when a client disconnects. I have also included that onDisconnect will remove a users name from the users list after they disconnect.

For onMessage I have included if statements for each of the commands on the system. As mentioned earlier I chose the order of them carefully to ensure the user uses the system as I have designed, meaning they will have to register to use all commands but can access help and quit commands without doing so. I have also added in some places where I will validate the user input, I have kept it brief at this stage for the most obvious forms of validation. However for the actual implementation I will validate more thoroughly as some instances may be required which aren't as easy to see form a high level.

Finally the onRegister function is one I created myself to allow users to be registered on the server. Here I will validate the username chosen and ensure its not being used already and that the given name is only one word. Once validated it will store the name and associated socket object into the userlist dictionary to hold all users active on the system.

```
function onStart():
    # Initialise server variables
    SET userCount = 0
    SET users = {}
    OUTPUT "Server started"

function onMessage(socket, message)
    # Split input to get command and parameter
    SET command, parameter = message.split()

    IF command == "MESSAGEALL" AND registered == TRUE
        SET message = parameter
        for user in userlist
            user.send(message)

    ELIF command == "MESSAGE" AND registered == TRUE
        SET user, message = parameter.split()
        user.send(message)

    ELIF command == "ALLUSERS" AND registered == TRUE
        for user in userlist
            OUTPUT user.name

    ELIF command == "CHECK" AND registered == TRUE
        IF user in userlist
            OUTPUT "User is active"
        ELSE
            OUTPUT "User is not active"

    ELIF command == "HIDE" AND registered == TRUE
        IF hidden == FALSE
            SET hidden = TRUE
        ELSE
            SET hidden = FALSE

    # Call register function
    ELIF command == "REGISTER"
        VALIDATE user not registered yet
        SET name = parameter
        CALL onRegister(name)

    # Help and quit commands are available to users before registering
    ELIF command == "HELP"
        OUTPUT commandsInfo

    ELIF command == "HELPLIST"
        OUTPUT commandslist

    ELIF command == "QUIT"
        OUTPUT "Quitting program"
        CALL onDisconnect()

    # User should register before using certain commands
    # Messaging specific or all users needs registering
    IF (command != "REGISTER" OR "HELP" OR "HELPLIST" OR "QUIT") AND name == NULL
        OUTPUT "Please register first!!"

    ELSE
        OUTPUT "invalid command, try HELP"
```

```
function onConnect()
    SET name = NULL
    SET hidden = FALSE
    SET registered = FALSE
    INCREMENT userCount
    OUTPUT "Connected"
    OUTPUT userCount

function onDisconnect()
    userlist.remove(name)
    DECREMENT userCount
    OUTPUT "Disconnected"
    OUTPUT userCount

function onStop()
    # Reset server variables
    SET userCount = 0
    SET users = {}
    OUTPUT "Server Stopped"

# For register add new user name to user list
# Name is stored with socket
function onRegister(name)
    VALIDATE name is one word
    VALIDATE name does not already exist
    userlist[name] = socket
    SET registered = TRUE
    OUTPUT "New user registered"
```

Describe in detail how the current implementation could be extended/improved to allow subgroups of users to be created and messages to be sent only to members of a subgroup. (3 marks)

To extend this implementation to include group chat feature, you would have to be able to store sub groups of users. In this protocol I stored users into a dictionary with their name and object as a key/value pair, therefore to create a subgroup of users I would create another dictionary holding whichever users are within the sub group. This new dictionary would be created as a server variable, and could store all sub groups as nested dictionaries within the dictionary holding all groups. This would then mean whichever person messages within a group, the protocol would first check if a user is apart of the group, then loop through all values in the subgroup dictionary and send the messages to them.

Implementing a feature like this would mean the protocol would need to handle more command requests from the user. Examples of such commands could be, CREATE, ADD, REMOVE, LEAVE, MAKEADMIN, MEMBERS, as well as another MESSAGE. These commands could work as shown below:

- CREATE – would be used to create a new group and allow the user to choose a suitable name for the group.
- ADD – would allow a user from within the group to add another user to the group
- REMOVE – would allow a user to remove another user form the group
- LEAVE – would let a user leave the group
- MAKEADMIN – this could let one user make another user have admin rights to potentially give them power to add or remove participants
- MEMBERS – this would let you view all members currently participating with in the sub group
- MESSAGE – This could be modified to take different parameters that the original message command, instead taking group as one of its parameters instead of the specified user.

In addition to the new data structures and commands used to implement subgroups, it would also need to be able to store chat history. As mentioned earlier my current implementation does not include chat history, so to include sub groups would mean that this would also need to be implemented, so that each of the users are able to see a clear record of all messages sent within the sub group, and who from. This is a realistic approach to how many messaging systems handle their own group chats, such as WhatsApp and Facebook, therefore it would be sensible to do so in this case as well. This can be done by again storing data into a dictionary, storing user and their message as the key/value pair instead so that every message stored has the sending user associated with it. This can then be displayed to all users by looping through the data and outputting it to each user to see the chat history.

There would also be some changes need in the onConnect and onDisconnect functions. onConnect may include any additional attributes that may need to be initialised such as whether a user is an admin etc. In onDisconnect, it would need to ensure that when a client disconnects from the server they are also removed from the group chat, by removing them from the dictionary holding the members of the group.

Is the protocol you created for this exercise stateless? Explain why or why not, or explain why this concept may not be applicable in the context of this application. (3 marks)

During my own research and knowledge I have gained from the lectures I would say this protocol is **not** stateless. Stateless communication between a client and a server means the system treats each request made by the client as an independent transaction to be completed, and that the server disconnects from the client and forgets that they connected. This is not how this protocol works, instead in this implementation the server explicitly remembers the client and does so by storing their socket objects whilst they are connected. These objects hold data about the client on the server, rather than the client storing its own information, meaning that it cannot be stateless as only this central server will be able to communicate with the client once connected. If it was stateless the client would store its own data and would be able to communicate with any other server to perform a request.

As well as storing the socket objects of each client, this protocol requires the client to be dependant on the server to perform its requests. Because the server holds the clients socket data, for all connected clients, it will also hold all session data as well, such as information about the requests, the messages all will be in the server log. In a stateless protocol the client would not be so dependant on the server and would be able to hold enough information to connect to any server, rather than a specific server only, therefore this clearly shows that this protocol is stateful instead of stateless.

One final thing is how simple the client side is in comparison to the server side. In this implementation the client is only responsible to taking input and sending the request to the server. The server holds all the logic including validation for client input, and parsing the commands before responding to the request. This type of design is consistent with a protocol that is stateful.