Ex2-Report_f85709jw

The protocol is designed to be as simple as possible while still allowing for verification of whether a request has been met or if there was an error. The commands can be thought of in two categories. Commands sent to the server and then responses sent back to clients from the server which also follow a certain pattern. The list of the commands are seen below:

Protocol Commands

Commands to Server

`register_user [user screen name]`
`delete_user`
`list_users`
`send_message [recipient] [message]`
`quit`

Responses to clients

`message [message data]`
`server_success`
`server_error [error message]`

Analysis of Each Command

`register_user`

This command takes as input a screen name which cannot contain a space, must be unique (not the same as any other registered user), and cannot be 'everyone'. The command registers a user on the server, storing the screen name and the socket in a hashmap. This is useful later for sending messages, where we can figure out which client to send a message to by decoding a screen name to a socket. The screen name is also stored as a socket variable for easy access later.

The reason for not allowing a space in the name is to make parsing other commands easier, a name with a space like 'john smith' might later be interpreted by another command as `[command] john smith [other parameter]` where 'john' is the first parameter and 'smith' is the second and the other parameter is now the third parameter, or smith would be lumped in with the other parameter.

The reason for requiring uniqueness is to help in the case of needing to send a message to someone, if there are two 'bob's then if we send a message to bob how are we to know which bob? We could send it to both but that's probably not what the sender intends and could leak private information.

The reason for not allowing 'everyone' is that it is a special word used to indicate whether a message is to be send to everybody. Allowing a user to be called 'everyone' would conflict with this special functionality.

The functionality allows an already registered client to register again, this is essentially equivalent to a name change. The previous entry in the hashmap is deleted.

If one of the requirements aren't met then a `server_error` is returned along with the details. If everything works then a `server_success` is returned.

Pseudocode

```
users <- new hashmap

if parameter contains ' ' or is "everyone" or is empty or in users then
        socket.send("server_error Invalid username")
else then
        delete users[socket.name]
        users[parameter] <- socket
        socket.name <- parameter
        socket.send("server_success")
```

`delete_user`

This command takes no parameters, it deletes the entry of the hashmap discussed above that corresponds to the client that sent the request, it also deletes the socket variable name. The reason for not taking a user as a parameter is to ensure that a client can only delete its own registration and not others, this improves security.

If everything works then a `server_success` is returned.

Pseudocode

```
delete users[socket.name]
delete socket.name
socket.send("server_success")
```

`list_users`

This command simply returns a list of all the users as a `message` which is discussed later. Also a `server_success` is returned.

Pseudocode

```
if length of users = 0 then
        socket.send("message There are no registered users")
else then
        socket.send("message The registered users are:" concat users.keys)
        socket.send("server_success")
```

`send_message`

This takes two parameters, the recipient of the message and the message itself. In the case where the recipient is 'everyone' the message is sent to every registered user as a `message` by looping through the hashmap, and then a `server_success` is returned.

If the recipient is anything other than 'everyone', a check is done to ensure the recipient is a registered user, if not a `server_error` is returned. If the recipient is registered, a hashmap lookup is done on the recipient to find the socket corresponding to the user and the message is sent as a `message`. A `server_success` is returned.

pseudocode

```
receiver, message <- parameter.partition(' ')

if receiver = "everyone" then
        for each name, socket in users
                if user not sender then
                        socket.send("message " + name + " -> everyone: " + message)
        socket.send("server_success")
else then
        if receiver not in users then
                socket.send("server_error Receiver is not a registered user")
        else then
                users[receiver].send("message " + socket.name + " -> " + receiver + ": "
+ message)
                socket.send("server_success")
```

`quit`

Quit is a very simple command that takes no parameters to indicate to the server that the client would like to disconnect. The server responds with a `server_success` to indicate to the client that it is safe to stop the connection with the server.

pseudocode

```
socket.send("server_success")
```

Responses

`message`

A way of sending arbitrary strings to the client

Example

```
socket.send("message Hello, World!")
```

`server_success`

A simple message send to the client to indicate that the request to the server was successful.

Example

```
socket.send("server_success")
```

`server_error`

`server_error` contains an additional message afterwards to describe the error that has occurred.

Example

```
socket.send("server_error Something went wrong!")
```