CS 3410: RPC Chat System

Introduction

In this assignment, you will write a simple chat server and client. You will use the `net/rpc` package in the Go standard library to manage communications.

As a user, you will interact with the client. When you start it, you must supply it with your handle and an optional server address:

```
./chatclient alice
```

(this connects to localhost:3410). Or

```
./chatclient alice :3411
```

(this connects to localhost:3411). Or

```
./chatclient alice krypton.cs.dixie.edu
```

(this connecst to krypton.cs.dixie.edu:3411). Or

```
./chatclient alice krypton.cs.dixie.edu:3411
```

It will connect to the server and register you as a user. In addition, it will get a list of all users currently logged in, and present that list to you:

```
Hi alice, connecting to localhost:3410...
List of users currently online:
    jdoe
    bob
    eve
    gary
```

Then you can type commands. The client understands the following commands:

- `tell <user> some message`: This sends "some message" to a specific user.
- `say some other message`: This sends "some other message" to all users.
- `list`: This lists all users currently logged in.
- `quit`: this logs you out.
- `help`: list all of the recognized commands.
- `shutdown`: this shuts down the server.

In addition to responding to your commands, the client will fetch messages being sent to you and display them on the terminal.

The server

The server sits and waits for incoming RPC requests. It keeps a queue of messages for each user currently logged in, delivering them to the client when the client requests them.

When you start it, you can supply a port number and/or address to listen on, or you can accept the default (port 3410 on all network devices).

```
./chatserver
```

or

```
./chatserver -port=3411
```

It provides the following functions over RPC to clients:

- Register: This message is used when a new client logs in. The request includes a username, and the response is empty. The server should deliver a message that the user has logged in to the queue of everyone else currently logged in.
- List: This message is used to gather a list of all users currently logged in. The request has no parameters, and the response is a list of strings, one per user.
- CheckMessages: This message gathers all pending messages for a client. The request is a string (the username, as used to register earlier), and the result is a slice of strings. Each message is a string in the list.
- Tell: This message sends a message to a specific user. The request is a type containing the following:
  - User: A string with the name of the user sending the message.
  - Target: A string with the name of the user to send the message to.
  - Message: A string.

  The response is empty. If the target user does not exist, the server should deliver a message to the sender (adding it to his/her queue) but should otherwise behave as though it succeeded. The message should be formatted (by the server) as `"<Sender> tells you <Message>"`.
- Say: This message sends a message to all users currently logged in. The request is a record:
  - User: A string with the name of the user sending the message.
  - Message: A string.

  The response is empty. Note that the message should be delivered to all users, including the sender. It should be formatted (by the server) as `"<Sender> says <Message>"`.
- Logout: This message logs a user out. All queued messages are discarded, and the queue for that user is destroyed. The request is a username, the response is empty. The server should also deliver a message saying that the user logged out to the queue of every other user currently logged in.
- Shutdown: This message shuts the server down. In addition to tracking a message queue for each user, the server object should have a channel. The shutdown handler can put a value in that channel to indicate that the server should shutdown. Then in the main goroutine, wait for a value from the same channel, and return from main normally when a value is found.

The server maintains a queue of messages for every user currently logged in. A user queue is emptied after a CheckMessages transaction when all of the messages have been returned to that user.

The client

The client parses the command line and connects to the server with a Register message. It then enters a loop:

- Prompt the user and wait for him/her to type something. Use a buffered reader to get input from the keyboard. See the `bufio` package documenation for an example:
  - bufio.Scanner
- Parse the command. If it is a recognized command, form a request and send it to the server. The `strings` package has many useful functions to help with parsing the input.
  - `tell` and `say` send their respective messages to the server and do not print out any additional message to the user.
  - `list` prints out a nicely formatted list of users as reported by the server.
  - `quit` tells the user it is quitting, logs the user out, and quits.
  - `help` displays a list of recognized commands.
  - For an empty line, do not output anything in response to the command.

    - For an unrecognized command, print an error message and then print the help message.

- In addition to the main loop that is handling keyboard input, launch another goroutine to call CheckMessages and display the result to the console. You can do this one of two ways:

    1. Have CheckMessages on the server return immediately. The client calls it, displays the results, then sleeps for a second or so (using `time.Sleep`), then repeats.

    2. Have CheckMessages on the server block if there is nothing to return. Only when there is actually a message waiting can does it return anything. The client then calls it, reports the result, and immediately calls it again. This is a little trickier, as you will need additional coordination on the server.

Implementation details

- Make sure your Go environment is set up correctly. In particular, you should have `GOPATH` set, and you should be working in a directory somewhere below `$GOPATH/src`. Be sure to run `go fmt` once in a while to ensure that your code is well formatted.

- Command-line arguments are supplied as a `[]string` call `os.Args`. You can use it directly, or you can use the `flag` package to parse named command-line arguments. See its documentation for details.

- Be careful to check for all of the possible ways that a user can start the client and the server. When joining a hostname with a port, use `net.JoinHostPort`.

- For an example of how to use the `rpc` library, see Chapter 13 of “Network Programming with Go”:

    - Chapter 13: Remote Procedure Call

- You should be careful of *race conditions* in the server. The simplest way to do this is to make sure that only one goroutine at a time can access a queue. There is a straightforward way to do this using *actors*:

    1. Start a goroutine (the actor) that owns the server state (the struct that holds the message queues). All other goroutines will act on the state (querying it or making changes to it) by asking the actor to perform the action.

    2. Communicate with the actor through a channel. The actor will run in a forever loop pulling requests from the channel, performing the requested action, and then moving on to the next request.

    3. Rather than moving all of the code to make changes into the actor itself, have the channel that communicates with the actor accept functions as its input. The actor simply runs each function that it receives across the channel. The input to each function is the state object. For example:

        - An incoming request needs to get the messages for a user
        - It creates a function (using a function literal) that takes the state object as its input and performs the necessary actions.
        - It passes that function into the channel to the state actor.
        - The state actor pulls the function from the channel, and calls it by passing the state to it.
        - Because the function is a closure (created using a function literal), it runs with full access to the context in which it was defined, i.e., that of the request handler.

    4. Because only one function acts on the state at a time (it is always the actor goroutine peforming the action), there is no danger of race conditions.

    5. Define a type for the channel that sends data to the actor, and you can use it as the main object whose methods are exported via RPC. So your main types would be:

        - A struct or map that holds the server state
        - A function type that accepts the server state as input and uses it to satisfy a request
        - A channel of handler functions. This is the type that has methods to be exported via RPC.

    For an example of this approach, watch this talk by Dave Cheney

What to pass off

You should record a screencast tto demonstrate your programs to me. Set up at least four terminal windows:

one with a server and three with clients. Demonstrate all of the required functionality in a brief screencast.