GopherChat

Created by: \ Lucas Kivi - kivix019

Implementation

All commands are read in immediately and some protocol checks are performed. These commands are then given to the client to run. Most of the commands are ready to for transfer to the server immediately, but some require further modification prior to issuing the command. For example: users cannot send messages without being logged in, so the user must log in, then the logged in username is attached to the command before it is sent.

Registration, Login, and Logout

All three are encapsulated within a `DataStore` class. This class maintains a file of registered users. It queries from the file on registration and login events and writes to it when a registration attempt is cleared.

Currently logged in users exist as a `Profile` in a vector. `Profile`s also contain queued messages. This will be detailed in the **Send** section.

Whenever a user is succesfully logged in the client starts 11 TCP connections, 1 is a UI connection for user messages and the other 10 are for file transfers.

Sending Messages

Sending direct and public messages, both anonymous and named, works like this: * `CommandData` is sent to server with message details * Server builds translates the `CommandData` into `MsgDta` * `MsgData` is sent to any recipients via their UI connections * The server responds to the sender with a `ResponseData` that represents the message status

Sending Files

Sending a direct or public file works like this: * Read contents file into memory on client and build a `CommandData` * Send `CommandData` to server * Server translates the `CommandData` int a `MsgData` * Server sends the `CommandData` containing the file to each recipient. * Server responds to the sender with a `ResponseData` that represents the file transfer status.

List

Listing all current users is simple. The server queries the data store for all signed in users, generates a data `ResponseData` structure to contain them, and returns it to the client.

Delay

The client's use a poll timeout value of 30 milliseconds. When the delay is issued a timestamp is kept. Then the client will not issue new commands until the duration declared in the delay command is elapsed.

Tests

I wrote some integration and unit tests. You can run them with `make run_all_tests`. They were very helpful as I refactored large portions of code.

Known Problems

There are a few minor memory management issues in both the client and the server. There is nothing critical but they are visible when using `valgrind`. I have a lot to do in my life right now and cannot be bothered to iron them all out. Otherwise, all required commands work as expected.

Setup

- Open up the app. If you need a new version, it is available via git: `git clone https://github.com/lukekivi/GopherChat.git`
- Enter the `GopherChat` directory `cd GopherChat`
- Build the client executable and server executable with by navigating to the base directory and running: `make`

Running The Server

I did all of my testing using make targets. The project description asks for executeables. Both are possible. I will detail both options.

Your Way

Generated executeables are directed to a `build` directory, so the syntax is a little different than the project description execution.

In `GopherChat` directory

```
./build/server/server_main <server-port-num>
```

Resetting the Registration Database

```
./build/server/server_main reset
```

My Way

Modify the top two values in `Makefile` to match your machine and the port number you will use. That way we have a single source of truth for servers and clients alike.

```
port_num = 9013
server_ip = 54.172.118.127
```

Run command:

```
make run_server
```

Running the Clients

Again there is your way and my way. Be sure you ran `make` in the base directory prior to attempting to execute a client.

Your Way

In `GopherChat` directory

```
./build/client/client_main <server-ip-address> <server-port-num> <command-file>
```

My Way

Adjust the top 3 entries in the `Makefile` to point to the server, the port the server is using, and the command file you are using. Here is an example setup

```
port_num = 9013
server_ip = 54.172.118.127
command_file = "commands.txt"
```

Then you can just run this command:

```
make run_client
```

Output

All output you will be interested in is directed to the terminal as is requested. If you want to look a little deeper at what is happening behind the scenes there are log files.

- Server logs can be found in `log/server_log.txt`.
- Client logs are given a random id. For that reason the first terminal output will always be the designated log file for the client instance. Output will look like this:

  ```
  $ ./build/client/client_main 54.172.118.127 9020 commands.txt
  Log id: 6297
  ```

  So after running the client I could find the log for that instance at `log/client_log_6297.txt`.