CMPT 373 Web Server Documentation.

John Wu

A list of documentation to help understand the social-gaming chatroom code.

The following text will be an attempt at documenting the provided networking code in a simple and readable way, in order to make development of the social gaming platform easier.

Included in the base networking code are the following files:

- Client.<h/cpp>
- **Server.<h/cpp>**
- **chatServer.cpp**
- chatClient.cpp
- chatWindow.<cpp/h>

For the most part, you'll only ever need to understand the interaction between of Server.cpp, and chatServer.cpp. The rest of the files are mostly ununderstandable black magic voodoo and probably won't help in the development of the social gaming platform.

Server.cpp is responsible for receiving the messages that clients send and sending out new messages to clients.

chatServer.cpp is responsible for taking in *`Message`* objects, processing them appropriately, and returning them along with the userID of all clients that should receive the message.

One thing that you should make sure you understand is the *`Message`* object (defined in Server.h)

- *`Messages`* contain two variables,
  - a connection (just think of it as a client ID)
  - the message's text

When a client sends a *`Message`* to the server, the ClientID will be the ID of said sender. chatServer.cpp will use this ID to process a message (refer to chatServer.cpp::processMessage).

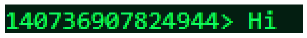
When chatServer.cpp is done processing all messages, it'll return a dequeue of *`Messages`*, but this time, the client ID will be used to identify the client that the message needs to be sent to.

chatServer.cpp

- Processes messages from the clients and generates a list of messages to send to all other users.
- Important Functions / Variables
    - `std::vector <Connection> clients`
        - A vector filled with ‘connection objects’ (defined in Server.h)
            - All you need to know is that they’re integer client IDs.

    - *`onConnect`*
        - Adds a new client ID to the *`clients`* vector.
        - Ran (by *`Server.h`*) when a new client connects.

    - *`onDisconnect`*
        - Removes a client ID from the *`clients`* vector.
        - Ran (by *`Server.h`*) when a client disconnects.

    - *`processMessages`*
        - Receives a list of *`Message`*s (defined in Server.h), sent from the server.
        - If the *`message`*‘s text was “quit”, disconnect the user who sent it.
        - If the *`message`*‘s text was “shutdown”, shutdown the server.
        - else:
            - Extracts the UserId of the message’s sender.
            - Creates a new ‘processed’ string of text in the following form:
                - “USER_ID> MESSAGE_TEXT”
                - ex. 

```
140736907824944> Hi
```

            - Appends the processed string to a previous one, if there is one.
        - Returns the final ‘processed’ string.

    - *`buildOutgoing`*
        - Receives a single ‘processed’ string
        - For all clients in the `clients` vector.
            - Create a new *`Message`*
                - The UserId, will be the recipient client’s Id
                - The Text will be the processed string.
        - Return a deque of all generated *`Messages`*

Server.cpp

- Responsible for receiving input from clients, and sending output back.
- Important Functions / Variables
  - *`Server::Receive`*
    - Returns a dequeue of *`Messages`* sent to the server by the clients.

  - *`Server::Send`*
    - Takes in a dequeue of *`Messages`* that (should) have been processed by chatServer.cpp
      - Extracts the userId portion of the *`Messages`*.
      - Sends out the *`Text`* part of the *`Messages`* to the user from the step above

  - *`Server::Disconnect`*
    - Given a userId, attempt to disconnect that user from the server.

Data Flow of a string of text.

For example's sake, lets take a room with two users with ID, 123 and 456.

- User 123 sends a message to the server "Hello User 456"
- User 456 sends a message to the server "I like shorts. They are comfy and easy to wear"
- chatServer.cpp::main() calls `Server::Receive.`
  - It receives the following dequeue of *`Message`* objects

    -
      ```
      {Connection: 123
       Text: Hello User 456}
      ```

    -
      ```
      {Connection: 456
       Text: I like shorts. They are comfy and easy to wear}
      ```
- chatServer.cpp::main() calls `processMessages()` on the dequeue.
  - It converts the first *`Message`* into the string:
    - `"123 > Hello User 456\n"`
  - It converts the second *`Message`* into a string and appends it to the previous string:

    -
      ```
      "123 > Hello User 456" \n
      "456 > I like shorts. They are comfy and easy to
      wear\n"
      ```
  - `It returns the final string`
- chatServer.cpp::main() calls `buildOutgoing()` on that string
  - It creates a new dequeue of *`Messages`*
  - It creates a new *`Message`* to be sent to user 123 and adds it to the dequeue

    -
      ```
      {Connection:     123

       Text:          "123 > Hello User 456" \n
                      "456 > I like shorts. They are comfy
                      and easy to wear \n"
      }
      ```
  - It creates a new *`Message`* to be sent to user 456 and adds it to the dequeue

    -
      ```
      {Connection:     456

       Text:          "123 > Hello User 456" \n
                      "456 > I like shorts. They are comfy
                      and easy to wear \n"
      }
      ```
  - It returns the dequeue of *`Messages`*
- Server::Send() receives the dequeue of *`Messages`*, and sends them out according the userIDs located in `Connection.`