COMP3331 Assignment Report

- z5280803
- Note: Salil OK'd this report being many pages but low in word count

Program Design

Note: Socket refers to a TCP socket in this report.

Files

- Two directories
  - `client/`
    - contains all `client.py` related files
    - `ClientMethods.py`
      - handles all commands issued by the user
      - handles all messages from the server
      - maintains all the state for a single user
    - `P2P.py`
      - File for handling direct peer to peer connections
      - Contains two classes
        - `P2P_server` for allowing one peer to act as a server to listen for another client to join
        - `P2P_client` for allowing one peer to act as a client to join another peer
        - both classes contain a `socket` attribute and `send_message()` method that allows a client to access the peer-to-peer socket objects and send messages over that socket with the same interface.
  - `server/`
    - contains all `server.py` related files
    - `ClientThread.py`
      - Contains a threaded class that is instantiated for each new client joining the server
      - Handles state of user's activity
    - `ServerHandlers.py`
      - Contains `ClientThread` class that handles all commands issued by the client
      - Maintains state about a client
- Root directory
  - `client.py`
    - A user will run this to start their client
    - Creates a socket that joins the server
    - Creates a thread that listens to the server
      - passes server instructions to `ClientMethods.py`
  - `server.py`

        - Creates and listens over a socket
        - Creates an instance of a `ClientThread` class for each new user that joins
    - `helper.py`
        - useful helper functions used by both `client.py` and `server.py` files

Data Structure Design

Client

- `client.py` instantiates `ClientMethod` object
- `ClientMethod` has two dictionaries
    - `handle` has key=command, value=function that parses command
        - most are just passed to the server
        - client-side-only functions are parsed individually
    - `response` has key=server response, value=function that parses response

Peer to Peer Communication

- `P2P.py` has two simple classes
    - One for creating a server
    - One for creating a client
    - Both behave exactly the same (albeit extremely simplified) compared to the `client.py` and `server.py` but for one client only

Server

- `clients` dictionary contains key=username, value=client info
- client info:
    - ```
      {
          "block_time": 0,  # what time they were blocked
          "client_socket": None,
          "client_obj": None, # ServerHandler object
          "client_thread": None, # ClientThread object
          "password": "",
          "log_on_time": -1, # what time the user last logged on
          "blacklist": set(),
          "offline_messages": []  # list of tuples of (msg, from user)
      }
      ```
    - Stores state about users that are required for functionality between users
    - Use locks to prevent race conditions
- `credentials.txt` stores username and password of each user in plain text
- `ClientThread` class stores time information about a client
- `ServerHandler` parses all commands issued by user
      ```

Application Layer Message Format

- Client and server sends JSON string as bytes
  - Converts dictionary -> JSON string -> `\r\n` appended to end of string -> Bytes sent to other end of socket
  - Receives and decodes Bytes -> separate by `\r\n` -> each JSON string decoded -> dictionary
  - `\r\n` is necessary since messages could be sent concatenated and something is needed to figure out the start/end of each message
- Sends information relevant to message only
- usually has format of
  - ```
    {
        "command": "string for which command",
        "message": "string for information to tell client"
        # any additional information that needs to be passed for this
    command specifically
    }
    ```
  - followed by `\r\n` when being sent as bytes

Description of How System Works

Overall Summary of behaviour

server.py

1. Server is started using `python3 server <port> <block duration> <timeout duration>`
2. `clients` dictionary is created which contains information about every user that has been created
2. Opens the given port and listens for any clients requesting to join
3. Client joins and a new instance of `ClientThread` is created
4. `ClientThread` keeps track of what time a user was active and instantiates `ServerHandler`
5. `ServerHandler` parses any commands a user issues to the server

client.py

1. Client started using `python3 client.py <port>`
2. `ClientMethod` class instantiated
3. thread created for `handle_server_messages()` which listens to the server on the given port
4. messages received from the server is passed into the `ClientMethod` object which parses it
5. On successful username and password input, infinite loop that gets the user's commands
6. Commands are passed into the `ClientMethod` object and parsed by it
7. `ClientMethod` sends messages to server

Trade-offs and Considerations
    ```

Timeout

- For timeout, I didn't want to create another thread for just timing out to save on resources on the server.
    - Threading would have allowed the user to be notified the moment they have been timed out
    - Currently, the user is notified if they are timed out after they have tried to issue a command.

`client` dictionary in `server.py`

- Find online history is O(n) since we have check each user against the time frame
    - Could create a second array of tuple with (`user's last online time`, `username`) but that would introduce redundant data which adds the possibility to introduce inconsistencies and also increases space requirements. Would decrease time complexity to O(log n) though.
- Use set instead of list for blocked users to reduce complexity from $O(n^2)$ to O(n) for searching this set.
    - also allows using set difference to find who is logged in and not blocked for simplicity

Client Socket Timeout window is very long

- Set socket timeout to an arbitrarily long time (1hr)
    - Hope that the server's timeout window > socket timeout window
    - The client starts a `recv()` that we do not want to have timeout since the server could wait up to `timeout window` amount of time before sending another message
- If the socket times out, the user is notified and logged off

Borrow Code

- Starter code for the server/client provided

Assumptions

- Password cannot be empty
- Username cannot be empty
- "private <user> <message> command sending an error message" is overwritten by a connection not existing
    - the command still fails when a connection doesn't exist or is offline, but the reason will end up being "You need to start a private session with a user first" since a connection needs to be set up first
    - technically fulfils requirements so I left it
- Spec doesn't specify what should happen on the other peer's side when one user uses stopprivate, so I made it the same behaviour as what happens when a user logs out