COMP3331 – Assignment Report

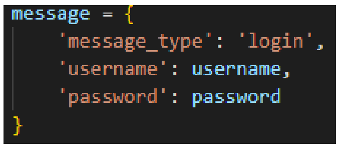
z5207984

Note: Python3 is used

Program Design

Client.py

The client establishes a TCP connection with the entered server IP and port. The user is then asked to input a username and password, which is sent to the server in a json format.



```
message = {
    'message_type': 'login',
    'username': username,
    'password': password
}
```

All commands contain a message_type field to distinguish the command

The next action is taken depending on the server response as detailed below

Response	Error message	Action taken
OK	Welcome to…	
INVALID	Invalid credentials. Please try again	The while loop to take user input starts from the beginning
BLOCK	Your account is blocked due to multiple login failures. Please try again	The program exits

After the user is verified, the client starts a threaded function using the entered UDP port to receive data. Finally the program starts an infinite loop which takes user input to execute any of the required commands, with incorrect input yielding either a custom error message depending on the command or a default error message.

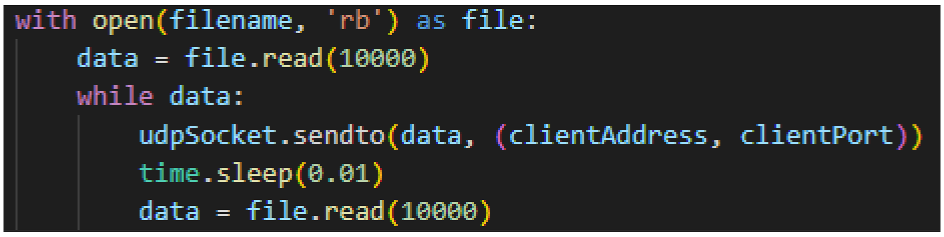
Command	Input	Error message
SRB	Entering own username as argument	Do not include your username in the usernames to add to the room!
SRM	Entering the wrong type for the room ID argument (only int is accepted)	Please enter a valid room ID
RDM	Entering the timestamp in the wrong format	Please enter timestamp in the format dd/mm/yyyy hh:mm:ss
UDP	Entering own username as argument	Can't upload file to yourself!

When the user input is OUT, the client sends a logout request to the server which terminates the client's thread and exits the program.

Client.py – UDP

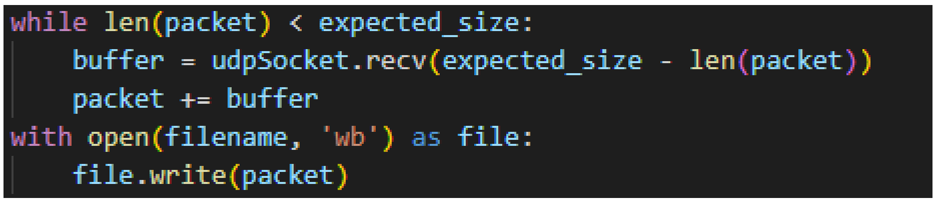
When designing the UDP section of client.py, I initially started with the same Thread class from server.py. However this led to many terminal hanging issues as I could not find any clean ways to terminate a thread without scrapping the idea of the class completely. Finally I decided to just call simply call the functions using Thread, hoping that no error would be raised when the client logged out. Sadly there is still an error which I could've debugged if there was more time.

For sending data, the filename size and filename is read and sent to the audience. Then, the file size is sent to allow the receiver to verify that all file data was sent before writing. The file to send is broken up in chunks in a while loop, which exits when all data is read. A small delay between sending chunks is needed, otherwise the receiver is unable to process all the received data quickly enough.



```python
with open(filename, 'rb') as file:
    data = file.read(10000)
    while data:
        udpSocket.sendto(data, (clientAddress, clientPort))
        time.sleep(0.01)
        data = file.read(10000)
```

For receiving data, the data is processed in the same order as sending data, the filename size, filename, file size and file. The data is read in a while loop, checking to see if the data received matches the file size. After it exits the while loop, the program writes the data to a file.



```python
while len(packet) < expected_size:
    buffer = udpSocket.recv(expected_size - len(packet))
    packet += buffer
with open(filename, 'wb') as file:
    file.write(packet)
```

Server.py

Uses a heavily modified version of provided multi-threaded server.py code written by Wei Song

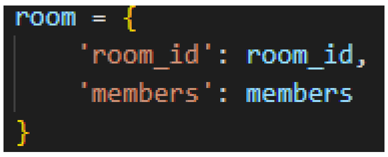
The server takes in 2 command line arguments, server port and the number of allowed failed attempts. The number of allowed failed attempts is saved as a global variable as all clients check against this value.

Two text files userlog.txt and messagelog.txt are created or wiped if they already exist. Finally an infinite while loop is started to listen for any requests for connection, and the program starts a separate thread instance for each connection.

The thread class waits to receive json messages in the run while loop, and decodes them. It then executes the specific command contained in the message_type field. Each command is separated into its own individual function.

To help with server functionality, some global variables are used to store data.

- The existing_users' list contains all usernames that have logged in, and does not remove them when they log out.
- The active_users list contains all users that are actively logged in.
- The rooms dictionary contains data on all the initialized rooms in the format

- ◦ 

```
room = {
    'room_id': room_id,
    'members': members
}
```

- The blocked users dictionary contains data on all the blocked users, where the key is the username of the blocked user and the value is the timestamp of when the user was blocked as an integer. This allows for easy comparison to check when 10 seconds have elapsed

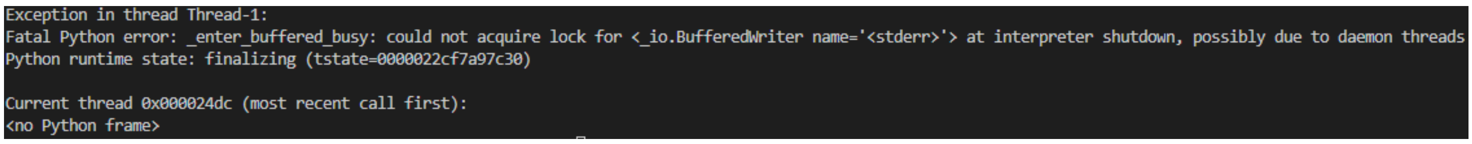
There are also two helper functions: get_room and get_message_after_timestamp. The function get_room returns a room's data based on the room_id argument and is used to check if a user is in the room before sending back messages when using the SRM command. The function get_messages_after_timestamp gets all the messages after a given timestamp from a list of lines. The list of lines is generated from a file, allowing it to be used for both broadcast messages and room messages, cutting down on repeated code.

Improvements

- Each file contained many lines of code. Readability could be increased if the code was split up across multiple files, such as a helper.py to store more helper functions and data.py to store global variables
- Several instances of repeated code could be removed by creating more functions.

Known Bugs

- When logging out, an error is raised due to the daemon thread not exiting properly



```
Exception in thread Thread-1:
Fatal Python error: _enter_buffered_busy: could not acquire lock for <_io.BufferedWriter name='<stderr>'> at interpreter shutdown, possibly due to daemon threads
Python runtime state: finalizing (tstate=0000022cf7a97c30)

Current thread 0x000024dc (most recent call first):
<no Python frame>
```