Report

Running the client and server

Both the client and server use **Python 3.5** or higher.

To run the client, use: `python3 client.py <host> <port>`

To run the server, use: `python3 server.py <port> <block duration> <timeout period>`

Overview and design

Both the client and server use the Python socket library to handle connections. The select system call is used to handle input from multiple sources. Both the client and server have a data limit of 2048 bytes per message.

Client

The client was mainly designed to only function as a tool to send and receive text to and from the server. It uses the select system call via Python to receive input from the user (via standard input) and the server. When the client receives a message from the server, it will be printed to the user without any modification, other than converting the message from a series of bytes to a string data type.

Server

The server was designed to do the majority of the “heavy lifting” in the assignment. It uses the select system call via Python to receive input from the client sockets whenever a socket has data to pass to the server.

Whenever a new client connects to the server via the welcome socket, the connection is immediately accepted. After accepting the connection, the server checks if the IP address is currently blocked, and immediately closes the connection if this is the case. Otherwise, a new User object is instantiated and assigned to the new client’s socket using a user map dictionary.

When any existing client sends a message to the server, the server finds the User object associated with the client’s socket, and calls a function inside the User object to process the message that the client sent.

The following data are stored globally:

- User map dictionary. Maps client sockets to User objects
- List of sockets connected to the server
- A blockedFromServer dictionary, with keys being IP addresses and usernames, and the values being the time the block was imposed.
- A dictionary storing the users’ last login times.
- A dictionary storing the users’ block lists.
- A dictionary storing each users’ offline messages to be sent to them on login

Checking for user timeout and blocking expiry (i.e. when the user is blocked long enough and should be unblocked from the server) is done by the checktimeout() function. This is a custom function that is run every second.

The User object

Each user object has a name, IP address, port, socket and other various metadata. User specific functions, such as blocking another user, and processing messages sent by the user’s socket are inside this object. If the user hasn’t logged in yet, all messages sent by the client is passed to an authentication function that prompts the user to send their username and password. Once this is done, the user will be allowed to access the full functionality of the chat server.

Design Tradeoffs

Tradeoff	Justification
Server handles the vast majority of the features. Client does not do anything other than sending and receiving messages to the server.	Most of the bug fixing and updating will only need to be done on the server. This minimises the need for the user to update their client. By not allowing the client to do anything other than sending and receiving messages, we don't need to worry about spoofing and many other security-related issues as all data verification is handled on the server.
Using the socket to identify a user instead of their name	While this forces searching of users by their name to be done in linear time, it is more important that we pass messages received from each client socket to their associated User object as fast as possible, since this is done more often than the searching of users that's done when the block/unblock command is used, or when a user logs in etc.
Using a custom function to check for timeout and user unblocking	I couldn't get socket.timeout() to work with select. This is how I got around the issue. A linear search is done through each current online user and blocked name/IP. By limiting this to once every second, impact to performance is minimised.
If user A blocks user B, user A can still send messages to B, but B cannot send messages to A.	While I considered having the block implemented to both users if one of them blocks the other, a user may still find it useful to send a message to a user that they've blocked. The assignment spec also does not state that the block needs to be implemented both ways if one blocks the other. The only requirement is that if user A blocks user B, then B cannot send messages to A in any way (nothing was stated about A sending messages to B after the block), and A does not receive presence notifications from B.

Possible extensions and improvements

Extension/Improvement	Details
Making offline messaging and blocking persistent across server restarts/shutdowns	This could be done efficiently by storing the messages and blocked users in a database. A simple implementation can be done using the Python sqlite3 library.
Minimising access to credentials.txt	Right now, the authentication and block user functions use credentials to validate the existence of specific users. This would not scale very well as opening files is relatively slower. This can be fixed by reading the file into memory once and accessing that to check users. A better way would to put user details into a database and maintaining a constant connection to that database.
Not echoing the user's password	Currently, the user's password is echoed in the client during the login process. This poses a security issue. A way to fix this would be use Python's getpass module.
Storing a history of messages	Users may want to look at messages that have been sent and received in the past. This can implemented by having the client store each message it receives into a log file.

Caveats

The select function call works differently in Unix and Windows. Since this assignment was written and designed to run on the UNSW CSE Linux servers/computers, it is quite likely that the server and client may not work correctly on a Windows computer, even with the Linux Subsystem for Windows.

Application layer message format

Messages broadcasted and sent by users are displayed in the following format:

- Broadcasts: `[yyyy-mm-dd HH:MM:SS] <user> has logged in`
    - Example: `[2017-04-28 21:47:09] hans has logged in.`
- Direct messages: `[yyyy-mm-dd HH:MM:SS] [<from> -> <to>]: <message>`
    - Example: `[2017-04-28 21:55:11] [me -> yoda]: hello; [2017-04-28 21:55:11] [hans -> me]: hello`
- Administrative messages from the server (e.g. block confirmation) are displayed as is

Acknowledgements

Some of the code in the client and server was written with assistance from:

- Chat server & client using select.select - http://code.activestate.com/recipes/531824-chat-server-client-using-selectselect/
    - Code borrowed included the use of select to receive input from multiple client sockets, and sending messages to client sockets.
- Finding local IP addresses using Python's stdlib - http://stackoverflow.com/a/28950776
    - This Stack Overflow answer detailed a procedure to get the server's current IP address, which was necessary to support clients from outside the local network.
- The Python 3 documentation - https://docs.python.org/3/
    - Provided the procedures to use the various standard libraries of Python