**FINAL EXAM**

In this project, you will implement a simple **chat room *server*** and ***GUI client*** in the **Python3** programming language. In doing so, you will gain:

- Hands-on experience with parallel, asynchronous programming
- Hands-on experience with GUI development using Tkinter
- Hands-on experience developing a server where parallel tasks must communicate with each other.

## Requirements

The high-level goal of this project is as follows:

**Implement a *server* that provides a common chat room, and implement a *GUI* client that can communicate with the server. With this system, a client can send text messages to all parties in the chat room, as well as receive notifications when other clients connect or disconnect. Clients do not communicate directly with other clients. Instead, all communication is routed through the central server.**

You must use the Python 3 programming language, specifically version 3.2.x or newer, which should be widely available.
(Note: On most systems, the binary will be called python3. To be safe, run python --version and python3 --version at the command line to see what you have.)

Your server program should be runnable from a file called **server.py**. Your client program should be runnable from a file called **client.py**. You can import additional Python files (so that your entire project is not in a single file), but the user should not invoke these helper files directly.

The following command-line arguments should be supported by the **client**:

- --help: This argument will print out a helpful message describing what arguments the program takes
- --version: This argument will print out the version number of the program (e.g. client.py 1.0)
- --target: This argument specifies the IP or hostname of the server to connect to. If not set, the default should be localhost.
- --port: This argument specifies the port number of the server to connect to. If not set, the default should be 8765.
- --username: This argument specifies the username the client will use when communicating with the server. This argument is required, and has no default.

The following command-line arguments should be supported by the **server**:

- --help: This argument will print out a helpful message describing what arguments the program takes
- --version: This argument will print out the version number of the program (e.g. server.py 1.0)
- --port: This argument specifies the port number of the server to listen on. If not set, the default should be 8765.

You should use the argparse Python library instead of parsing the arguments yourself. Argparse will provide the --help and --version arguments for "free".

## Client/Server Protocol

Clients do not communicate directly with other clients. Instead, all communication is routed through the central server. Each client communicates with the server over a single TCP socket. Thus, a client will only have one active socket, but a sever will have many active sockets, one per client.

The protocol used by the chat client and server is very "HTTP-like", with headers and payload data. The following methods must be supported by clients and servers. The same methods are used for outgoing data (e.g. the client sending a message to the server), and for incoming data (e.g. the server sending messages to the clients). Each method must be prefixed by CHAT/1.0, where 1.0 is the version number of the chat protocol.

| Method Type | Description |
|---|---|
| **JOIN** | The client sends a JOIN message to the server when entering the chat room. The server forwards this JOIN message to all other connected clients to notify them of the new user's arrival. This message must include the username header. |
| **LEAVE** | The client sends a LEAVE message to the server when leaving the chat room. The server forwards this LEAVE message to all other connected clients to notify them of this user's departure. This message must include the username header. |
| **TEXT** | The client sends a TEXT message to the server with user-provided text. The server forwards this TEXT message to all other connected clients. A valid TEXT message must include the username header, the Msg-len header, and a non-zero payload length. Clients and servers should ignore TEXT messages not meeting this requirement. |

The following headers can be specified in a message. Each header must be formatted as "Header Name" : "Value", with each header separated by \r\n symbols.

| Header Name | Description |
|---|---|
| **Username** | The name the client wishes to use in the chat room. This header is required on all messages. |
| **Msg-len** | Message Length: The number of bytes of the payload portion of the message. (The payload immediately follows the \r\n\r\n separator from the header). If this header is not set, the length of the payload is assumed to be 0 bytes. |

**Example -** client joining a chat room:

```
CHAT/1.0 JOIN\r\n
Username: shafer\r\n
\r\n
```

**Example -** client leaving a chat room:

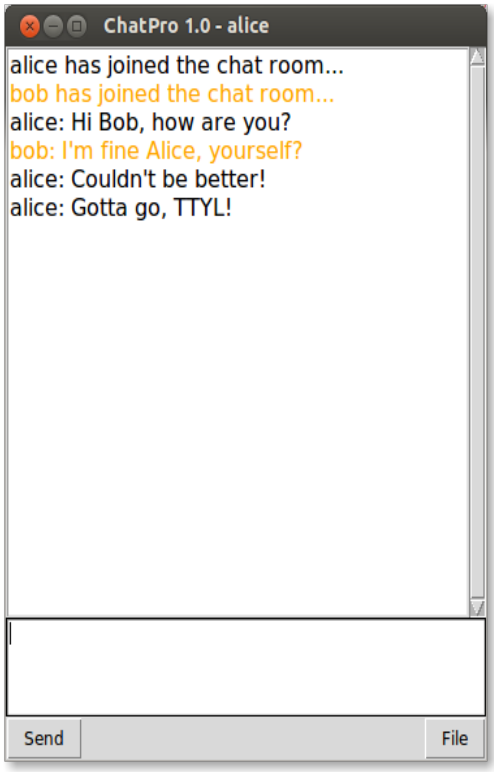
```
CHAT/1.0 LEAVE\r\n
Username: shafer\r\n
\r\n
```

**Example** - client sending a message to chat room:

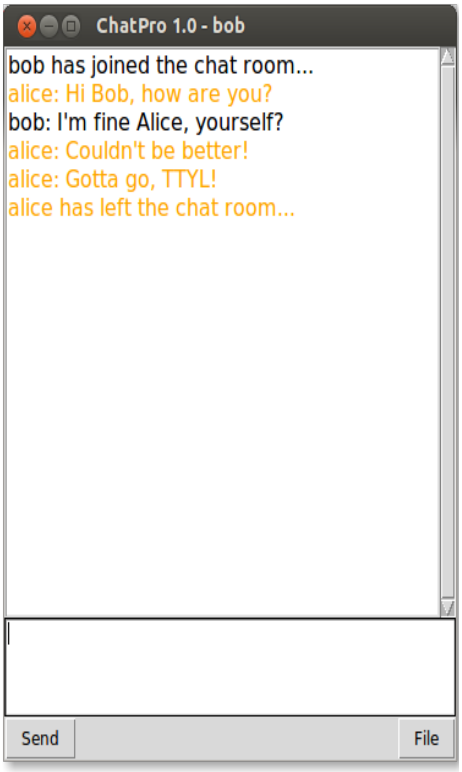
```
CHAT/1.0 TEXT\r\n
Username: shafer\r\n
Msg-len: 22\r\n
\r\n
My first chat message!
```

## Chat Client

You will write a chat client to allow the user to communicate with other users. This client must include a graphical user interface (GUI) built using the **Python Tkinter** environment. (Note: You are welcome to print debugging or error messages on the console, but the user interaction with the program should only be through the GUI). An example GUI interface is shown below. Here, Alice joined the chat room first, then Bob. They communicated back and forth, and then Alice logged off first.




**Alice Chat Client**




**Bob Chat Client**

Your user-friendly chat client must be **asynchronous**. For example:

- If the user is typing a message to send to the chat room, new messages should still be received and displayed on screen. (New messages should **never** be blocked until the user finishes typing their outgoing message.)
- After the user finishes typing and sending one message, the user should be able to immediately begin composing and sending another message.
- The GUI should be completely decoupled from the network. The GUI should not be impacted if the network is slow, and the GUI should never block waiting on a network call to send() or recv().

**If you find yourself calling network functions such as send() and recv() from inside the same thread as the GUI, you are almost certainly building the chat client *incorrectly*.** As a **minimum**, you will need at least two threads in your chat client, and it might be easier to add additional threads:

- **User interface thread** that displays the GUI and handles GUI events like the user typing data, clicking on buttons, etc...
    - **Warning!** Tkinter is only designed to work in the **main thread**. Thus, put your user interface code in the main thread, and put non-UI tasks in other threads.

- **Data thread** that transmits data on the network (as received from the GUI), and receives data from the network (and forwards it to the GUI for display).

**Tip:** You will need to communicate between threads in a safe manner to prevent data corruption. The **Python Queue** class has been specifically designed to be thread-safe in a multi-producer, multi-consumer environment. You can put multiple items on the Queue as a group - see the **Python tuple** primitive.

**Tip:** You may want to synchronize events between your threads. For example, it would be nice if the user interface thread did not display the GUI until the data thread successfully connects to the server. (Otherwise, without a successful connection, your chat client can't do any useful work). The **Python Barrier** class is a simple way to accomplish this.

**Tip:** Be sure to review the given Python3 GUI Example Program **"gui.py"**, which should save you some time.

After the user composes a message and clicks the send button, that message should immediately be displayed by the client in the chat window. When the server receives the TEXT message, the server will echo the message to all connected clients except for the client that originally sent the message. This ensures that the user will never see their own messages duplicated in the chat window.

## Chat Server

You will write a chat server to act as the central communication hub for the chat room. The server should listen on the specified TCP port (8765 by default) and accept incoming connections from clients. Once a connection is established, a server should receive incoming messages (JOIN, LEAVE, or TEXT) from clients, and forward those messages to all other connected clients. **Note that a server should not forward a message back to the same client that sent it!** Pipelined communication is specifically permitted in this protocol. The server can send multiple messages back-to-back to the same client, and could also expect to receive multiple back-to-back messages from a client.

You server will support multiple sockets, one per connected client. To manage these sockets, you should adopt one of the following programming models (your choice):

- One thread per socket, where blocking on send() and recv() is allowed
- All sockets in one thread, managed by select() to choose only sockets that have active work to do so that blocking is minimized.

A usable chat server must be asynchronous. Specifically, receiving a large message from slow client X should not also prevent the server from receiving a message from fast client Y. Similarly, if client X is silent and not sending any messages, that should not prevent the server from receiving a message from client Y.

## Resources

See the attached resource file for links that will help you when developing your solution.