Emily Deneen, Craig R Harrell

edene190@live.kutztown.edu, charr534@live.kutztown.edu

Client/Server Chat Application

Table of Contents

[Project Members 2](#_Toc151230447)

[Communication Plan 3](#_Toc151230448)

[Tasks 4](#_Toc151230449)

[Programming Languages 7](#_Toc151230450)

[Project Requirements 8](#_Toc151230451)

[Application Inputs/Outputs 10](#_Toc151230452)

[Common Functionalities 12](#_Toc151230453)

[Protocol 13](#_Toc151230454)

[Sequence Diagram 16](#_Toc151230455)

[Test Plan 17](#_Toc151230456)

# Project Members

The team consists of two team members, Emily Deneen, and Craig R. Harrell. Craig will be acting as the team lead.

# Communication Plan

The team has decided to use the following means of communication for this project:

In-person, Discord, Zoom

In-person communication will happen whenever we have in-class work time allotted or when we designate a time to meet outside of class for the purposes of this assignment.

Discord will be the primary means of communication between team members, it will be used for rapid communication of minor issues, ideas, and files.

Zoom will be used to communicate at times when a meeting is necessary if travel is a inconvenience/burden.

# Tasks

Tasks have been broken down into 4 major categories: Server, Client, Libraries, Documentation. Within each of these categories are smaller tasks, the team has decided on the following distribution of the major categories.

Craig will be handling the Server.

Emily will be handling the Client.

Shared responsibilities will be the Libraries and Documentation.

Server Tasks:

|  |  |
| --- | --- |
| Task | Estimated time |
| Build a basic server that able handle a client connection with correct error handling and interrupt listening. Sends only its own messages to client. | 30 minutes |
| Build on the basic server to accept N number of connections. | 30 minutes |
| Build on the N number of connections by storing usernames and IP addresses associated with the username. | 45 minutes |
| Filter usernames to only allow unique usernames | 20 minutes |
| Filter usernames to omit usernames that would misleading | 20 minutes |
| Add broadcasting ability for sending messages to all clients. | 15-20 minutes |
| Add message received capability. | 10-15 minutes |
| Add message logging capability | 10-15 minutes |
| Add message broadcasting ability to all but the original sender. | 30-45 minutes |
| Add removal of client from active list on client exit and broadcast user leaving to all still connected clients. | 30 minutes |
| Add functionality for broadcasting to all currently connected clients that the server is shutting down and deny any new connections from occurring. | 40 minutes |
| Add GUI For better experience | 2-3 hour |
| Document Server | 1 hours |

Client Tasks:

|  |  |
| --- | --- |
| Task | Estimated time |
| build basic client that can connect to the server and receive + output messages. have correct error handling | 30 minutes |
| allow input for nickname and check with server for uniqueness | 20 minutes |
| add functionality for sending messages | 30 minutes |
| Handle client disconnecting (expected and unexpected) | 30 minutes |
| Create GUI for better user experience | 2-3 hours |
| Document Client | 1 hour |
|  |  |

Library Tasks

|  |  |
| --- | --- |
| Task | Estimated time |
| Decide which functions/methods can be universal across server and client | 1 - 1.5 hours |
| Decide on naming conventions and classes | 30 minutes |
| Create classes and methods | 1 - 2 hours |
| Test individual methods with both Server and client | 1-2 hours |
| Document Library | 1-2 hours |

Documentation tasks

|  |  |
| --- | --- |
| Task | Estimated time |
| Document Server, Client, library | 1-2 hours each |
| Create ReadMe | 1 hour |
| Create Network Program Design | 4 days |

# Programming Languages

Our team has decided to use strictly Python for our client, server, and libraries.

# Project Requirements

Server Requirements:

* Must check usernames for uniqueness.
* Must store messages from clients.
* Must transmit messages received from client to other clients w/out sending to originating client.
* When shutting down must warn client it is shutting down in x seconds and then wait x seconds before closing
* Must shut down gracefully.
* Must store timestamp, IP of client and client username first connection in a log and in a array.
* Must keep a log of messages.
* Must follow best programming practices.
* Must be documented.

Client Requirements:

* Must allow for input of a username multiple times until unique one is found.
* Must read new messages sent from server & output them.
* Must take a server name and port address in its command line arguments.
* Must be able to send messages.
* Must send 'Bye' message to server on disconnect.
* Must be able to receive either a push or pull request to read messages.
* Must follow best programming practices.
* Must be documented.

# Application Inputs/Outputs

Client Inputs

* Username - must contain letters and numbers only. The username will be entered in either the command line or through a GUI.
* Server name/host and port - host may contain letters or numbers and periods based on IPv4, IPv6 protocols. (i.e. 10.0.0.1, localhost, 2001:0db8:85a3:0000:0000:8a2e:0370:7334) this will be entered in either the command line as client <hostname> <port> or through a GUI, the port will be resigned to a int and be within the range of 10000-65535.
* Messages from user - may be any length and contain any character, contained in a JSON string with the following format {"type":"<type>", "message":"<message>", "timestamp":"<timestamp>"}
* Keyboard Interrupts - user will be able to use keyboard shortcuts to close the app and the app will gracefully close.
* Username uniqueness from server - Server will respond with {"type": "username", "message": "<good/taken>", "timestamp":"<timestamp>"}
* Server messages - client will accept messages from the server, messages will be formatted in JSON and follow the following conventions: {"type":"<message/shutdown/user exit/username>", "message":"<<client msg>/shutdown/<username>/<good/taken>", "timestamp":"<timestamp>"}
* Other client messages from server - client will accept client messages from the server in the following JSON format: {"type": "message", "message":"<message>", "username": "<username\_of\_sender>", "timestamp":"<timestamp>"}
* Interrupts - Interrupts will be any exception, user input such as ctrl+ c, or any other factors that would hinder the expected parameters of the running of the application. The client will use best programming practices to catch these and close the program gracefully.

Client Outputs:

* User message to server- the client will send messages to the server in the following format {"type”: “message", "message":"<message>"}
* Exiting message - the client will send an exit message to the server; it will be in the following format {"type": "exit"}
* User message to display- the client will display the messages to the user in the following format as a new line: <timestamp>: <username>: <message>
* Username to server - client will send the username selected by the user to the server in the following format: {"type": "username", "username": "<username>"}
* Server messages - the client will output any messages received from the server in either the terminal or GUI.

Server Inputs:

* Port to listen on - server will accept either a command line argument or a GUI input for the port number, it should be within the range of 10000-65535.
* Host name - Optional input on either the command line or GUI to provide a IPv4/IPv6 address, default address will be provided for localhost with no further input from user.
* Client Connections - server will listen on selected port for connections and accept those connections.
* Client Username/IP- client will send server input in the following format: {"type": "username", "username": "<username>"} and if unique to the client list will add to the client list and capture IP address as key.
* Client message - Server will accept client messages in the following format: {"type": "message", "message": "<message>"}
* Client exiting - Server will accept client exiting messages in the following format: {"type": "exit"}
* Interrupts - Interrupts will be any exception, user input such as ctrl+ c, or any other factors that would hinder the expected parameters of the running of the application. The client will use best programming practices to catch these and close the program gracefully.

Server output:

* JSON {"type": "username", "message":"<good/taken>", "timestamp": "<timestamp>"} to client on connect with username sent to server.
* Log file for connections - server will keep a log of all connections and exits with IP address of client, client username, timestamp, and action (connect/exit) in the following format: (timestamp, IP, username, action) on individual lines.
* Log file for chat messages - server will keep a log of chat messages with the following format: timestamp, IP, username, <message>
* Exiting/shutting down- Server will broadcast to all users that it is shutting down in x number of seconds. The format will be in JSON and be: {"type": "shutdown", "message": "shutting down in....(n) seconds."}, it will also display to the admin the countdown in either a terminal or GUI.
* User/client messages - server will accept, process, then broadcast all user messages to the chat. The format will be: {"type": "message", "message":"<message>", "username": "<username\_of\_sender>", "timestamp":"<timestamp>"}

# Common Functionalities

The following methods were found to be in common between the server and the client:

Send, Receive, Close, Connect, Event Listener(s), Exceptions

These functionalities can be achieved through careful consideration of the methods use and proper tweaking or by using the multipledispatch library to overload.

# Protocol

*Figure 1 Server*

Server Machine state


*Figure 2 Client*

Client Machine state


Server: (see figure 1)

Start:

The server is started from the command line. It is given the arguments of a port number to bind to then enters an idle state.

New Connection Broadcast:

A client connects to the server, passing it a desired username (nick). If the nick is unique the server adds the nick to a list and informs the user that it has been accepted. If the nick is not unique it requests the user try again. Once accepted the server broadcasts out a message to all clients currently connected that nick has entered the chat and returns to an idle state.

Client Message Broadcast:

A client sends a new message to the server. The server then broadcast that message to all other users connected to the server, but not to the original sender. Once every connected user has been notified the server returns to an idle state.

Client Exit Broadcast:

A client sends a message that they are leaving the chat. The server then broadcasts that the user has left to all other clients connected and removes the user from the nick list.

Exit Broadcast:

The server has either received a shutdown command or has encountered an unrecoverable error. It broadcasts a message to all connected clients that the server will shut down within x number of seconds, then waits x number of seconds. After which the server cleans up and closes.

Client: (see figure 2)

Start:

The user will start the program from the command line and enter the arguments for host and port host is listening on.

Send Username:

The client will prompt user for a username, send the username to the server and listen for a response. An accepted response will return the client to idle, a rejection response will prompt the user to find another username. This process will continue until a unique username has been established.

Send Message:

A user will enter a message and the client will show the message within the list of messages of the chat, it will then send the message to the server for processing and return to idle state.

Receive Message:

The client will listen for messages broadcast from the server. When it receives one it will add it to the list of chat messages and display it to the user. It will then return to the idle state.

Send Exit:

The client will send to the server that the user is shutting down the client and then close.

Messages sent between the server and client or visa versa will be in JSON format with the following being the accepted format of the JSON per request:

**User/Client Requests:**

New user: {"type": "username", "username": <username>}

Message: {"type": "message", "message": <message>}

Exit: {"type": "exit"}

**Server Requests:**

Unique Username: {"type":"username", "message": <good/used>}

User joined: {"type": "message", "message": "<username> has joined the chat", "timestamp": <timestamp>}

User Message: {"type": "message", "message": <user message>, "username": <username>, "timestamp": <timestamp>}

User Exit Message: {"type": "message", "message": "<username> has left the chat", "timestamp": <timestamp>}

Server exit message: {"type": "exit", "username":"admin", "message": "Server is shutting down in <n> seconds.", "timestamp": <timestamp>}

# Sequence Diagram



Test Plan

Username Uniqueness:

We will conduct tests on whether more than one user will be able to claim the same username. This test will include the following usernames:

sparky

user1

pope

Frank

Bob

Jane

Julia

Reserved Usernames:

We will conduct tests to ensure certain reserved usernames are unavailable to the client users. The test will include the following usernames:

Anything with mod, admin, super-user, owner

Correct port:

We will test to see if enter the wrong port will allow the client to continue to run.

Closing program unexpectedly:

We will test to see if closing both the server and client unexpectedly, through keyboard interrupts and purposely throwing errors will result in the expected graceful close.

Long message:

We will make sure that the server and clients are able to handle a maximum package and process it correctly.

Repeat messages:

We will test a case where there is a DoS type attack by setting a client to flood the server with messages. To ensure that proper procedures are followed.

Simultaneous messages:

We will attempt to have two clients send simultaneous messages to make sure threading is properly working.