CSE 150
Final Project – Let's Chat!

1 Due Dates and Submission

This assignment is divided into multiple parts, each with its own deadline.

Part 0: State diagram of Client protocol and Project Quiz	Submitted during lab session Week 7
Part 1: REGISTER and BRIDGE requests sent from Client to Server. Complete Git Tutorials	Due Sunday 11/24 at 11:59pm
Part 2: Client and Server complete.	Due Wednesday 12/4 at 9pm
Early Birds: Students who finish and submit their entire assignment early by 11/27/2024 can earn 10 points **extra credit**	Due Wednesday 11/27/2024 at 9pm. Demos on Monday 12/2/2024
Project Demos are REQUIRED – no credit will be given for project without demo	Thursday 12/5 and Friday 12/6; possibly Saturday 12/7.

Submissions:

Part 0: **Individual Grade** – Client protocol state diagram and Quiz on Canvas. Completed during the lab.

All code is a Group Grade – Each group submits one copy of their code on Canvas.

Naming Conventions:

Part 1: s1<CruzIDStudent1>s2<CruzIDStudent2>Client1.py – client code for Part1

Part 2:

- s1<CruzIDStudent1>s2<CruzIDStudent2>Client2.py – client code for Part2
- s1<CruzIDStudent1>s2<CruzIDStudent2>Server.py – your Server code
- PDF with the answers to Questions in Section 7. Will update submission instructions.

NO LATE SUBMISSIONS: No late submissions for any portion of the project will be accepted. Please don't ask - we cannot accept late submissions. Submit whatever code you have completed by the above dates for partial credit.

2 Assignment Groups

This project will be completed in a group of 2 students. You can choose a partner or a partner can be randomly assigned. At the end of the project a private partner review in which you will evaluate your work with your partner will be submitted to the teaching team.

3 Requirements

Part 0: Draw a state diagram of the client protocol and complete the online Canvas quiz. **No hand-drawn images accepted. Hint: To determine the states, read the overview in Section 5.**

Part 1: Client sets up communication with the server in which the REGISTER and BRIDGE requests are sent from the Client to the Server (one direction only).

Test thoroughly against provided executable Server and the Gitlab pipeline as well as your own individual unit testing. At the end of this process, you know that you can send 2 properly formatted messages over a TCP connection from the Client to the Server.

Part 2: Client and Server code are complete. All code submitted and Questions (Section 7) are answered and submitted. Part 2 is described in Sections 3, 4, 5 and 6.

Environment:

- Python3 and VM: The program must be written in Python3 and run on the Mininet VM. Code that is not developed on your Mininet VM might not be compatible with our test scripts and Gitlab pipeline. It is very important to follow these guidelines as we cannot run code manually.

- These libraries are allowed and encouraged: socket, argparse and select

- All input must be read from stdin. Terminal output is written to stdout.

Commands and Messages: Message format, commands and message types required by the protocol (provided in Section 5), must be used. No new message types are needed and should not be introduced.

Error Checking: Your program must check for the following error conditions:

Socket Errors: The socket operations should handle any errors or exceptions thrown. An example of an error that you could experience is trying to use a port that is already in use by some other process. Other common errors include creating sockets, binding the socket, or receiving data.

Refer to Exception Handling in Python in Section 9 - Useful Resources

Detecting Closure Signals and Handling Termination: Ensure your program is capable of detecting and appropriately responding to a **QUIT** message from the peer or TCP **FIN** packets indicating the **other side** has closed the connection (gracefully or not). This involves checking for zero-length messages as a signal that the connection has been closed, possibly due to the peer's program abruptly crashing. Get help in the lab with this.

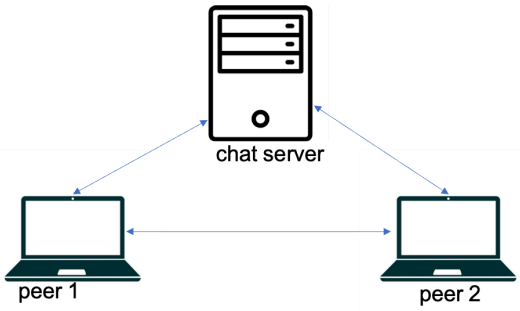
Closing your program: Your program must close gracefully – make sure to close any open sockets before exiting the program.

4 Getting Started

1. Read Section 5 carefully to understand the protocol's message format, types of messages, and the terminal commands.
2. Part 0: Complete the State Diagram of the Client. Focus on the client protocol operation (state, events, action and transitions).
3. Carefully review Example 1
4. Part 1: Follow the Requirements in Section 3 and Message Types in Section 5.5.

5 Overview of Chat Program

The goal is to practice with socket programming to create a TCP-based chat application with Client and Server programs. The server stores peer contact information to facilitate peers connecting and chatting. The simple architecture is shown and described below:



In this chat program the 2 peers operate like WALKIE TALKIES. At any given time while chatting, the peers are either "**reading**" or "**writing**". Writing ends when a newline is entered in the terminal. Following this, the peer switches to reading and waits for an incoming message. This back and forth behavior continues until the chat ends.

5.1 Server Details: The server accepts TCP connections from clients on a designated port (your choice) and listens for incoming messages. See Section 5.5 for a complete description of all Message Types.

The server accepts two types of request messages from clients:

1) REGISTER - clients provide their contact information.

The server stores the client's contact information in a file and returns a REGACK response message.

2) BRIDGE - clients request peer contact information.

When a BRIDGE request is received, the Server returns a BRIDGEACK to the client with previously stored contact information in the headers For example, if peer 2 sends a BRIDGE, the Server will return the contact information for peer 1.

5.2 Client Details: The clients operate in two modes: 1) WAIT - the first client to contact the server (/register and /bridge) waits to be contacted by a peer and 2) CHAT - a client is actively exchanging messages with a peer.

The first client to contact the server transitions into WAIT after /register and /bridge. **The second client** goes directly to CHAT state (bypasses WAIT) after /register and /bridge**.** (Show these transitions in your state diagram - review the commands and messages.)

Terminal input and Starting Chat: Open two terminals – one for each peer. The Client program (see Section 5.3) waits for /register and /bridge to be entered. (Remember to read from **stdin!).** The order matters:

- First, start a client – it will register and bridge.
- Then, start the second client – it will register, bridge and initiate chat with its peer.

Stated another way, there are two ways that clients enter CHAT mode:

1. After /register, /bridge the first client is in the WAIT state: an incoming chat request is received → transition to CHAT
2. After /register, /bridge, and /chat (this is always the second client) → transition to CHAT

TCP Connections: All communication is over a TCP connection. Clients close the TCP connection between the server and client after each request/response - i.e., only one TCP connection is open at a time (See Simplifying Assumptions). Peers chat over a TCP that remains open until the chat conversation is over.

Ending Chat: Chat continues until one peer ends by entering /quit. The TCP connections are then closed, and the chat program terminates. See the entire list of terminal commands in Section 5.4 and Example 1 below for more details.

Ending the Client Program:Chatting peers end the session when one user enters /quit or Control-C. At that point the TCP connection is closed and the chat programs terminate. (see Section 5.8 Example Usage).

5.3 Running the Programs

Starting the Server: Enter a port number on the command line in the server terminal. For example, to start a server listening on port 5555, enter:

```
> ./python3 Server.py –port=5555
```

Starting the Client: Enter the following on the command line in the client terminal:

- client ID (user's name, composed of only letters and numbers)
- Client port number (client will wait for incoming chat requests)
- server's IP address and port number

For example, to start the client program with a client ID "student1" listening on port 3000 and the server running on IP address 200.26.180.43 and listening on port 5555, enter:

```
>python3 client.py --id='student1' --port=3000 --server='200.26.180.43:5555'
```

Note: "IP:port" is standard notation to indicate IP address and port number. You do not need to error check the client IDs - just make sure that you enter unique IDs.

Errors: If you are getting errors using the above line, it could be due to copy/paste. Try to manually type it in. Angled quotes (single or double) cause the error. ‘ vs ' or “ vs "

5.4 Terminal Commands

Client: Your client should accept the following commands entered in the terminal and respond accordingly. If displayed output is required (Section 5.7), write to **stdout**.

All commands (other than Control-C) should be in the format "/<word>", in which <word> is a sequence of lower case letters. To keep it simple - commands appear only at the beginning of a newline, i.e, they do not appear in the middle of messages and any command not defined below can be ignored.

- **/id**: the clientID of the user is displayed in the client terminal
- **/register**: client sends a REGISTER message to server
- **/bridge**: client sends a BRIDGE message to server
- **/chat**: client initiates chat session with peer
- **/quit** -
    - If peer writer enters **/quit** during Chat mode in the terminal, the Client sends a QUIT message its peer and closes socket. Chat ends and programs terminate.
    - If peer writer enters **/quit** outside of Chat mode, Client exits the program.

- **Control-C** should be supported to end programs (server and client)

Server: Your server should accept the following terminal input (use **stdin**!) directly and respond accordingly to stdout:

- **/info:** the server outputs each registered user, the username, IP address and port number (See Section 5.7).

5.5 Message Types

The message types exchanged between client-server and client-client are modeled on the plaintext style of messages used in HTTP following the Client-Server model. Clients issue request messages and Servers respond. The format of all messages exchanged is shown in Figure 1. Think of the message types as similar to HTTP Methods and Headers, which are sent in plain text; the headers are name:value pairs. The following messages are defined:

Client request and Server response messages follow this format:

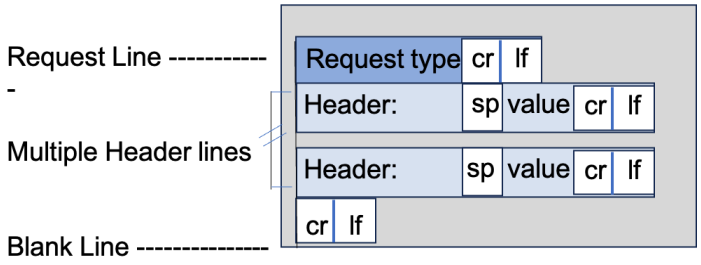


Figure 1: Message format

Client Request Types and Headers:

- **REGISTER** (client->server)
  - The client provides the server with its contact information
    Request Type REGISTER
    Headers clientID: name
    IP: ipaddress
    Port: port number (client contact information)

- **BRIDGE** (client->server)
  - Used to obtain peer contact information
    Request Type BRIDGE
    Headers: clientID: name (clientID of requesting client)

- **CHAT** (client->client) Request chat with peer

  You design this message (and headers if needed) following the message format given above

- **QUIT** (client->client)
  - End communication with peer
    Response Type QUIT
    Headers No headers needed
    Note: QUIT is generated when /quit is entered in the terminal. If entered during Chat mode, the other client is notified to also exit and terminate.

Server Response Types and Headers: Server responses also follow the format in Figure 1, substituting Request Type with Response Type.

- **REGACK** (server -> client)
  - Acknowledges receipt of REGISTER message and echos back contact information received
    Response Type REGACK
    Headers clientID: name
    IP: ipaddress
    Port: port number
    Status: registered

- **BRIDGEACK** (server -> client)
    - Acknowledges receipt of BRIDGE message and returns peer contact information stored (excluding requesting client contact information). **Note: if no client has registered yet, then the header values are empty.**

Response Type	BRIDGEACK
Headers	clientID: name IP: ipaddress Port: port number

5.6 Simplifying Assumptions

To simplify the implementation of the chat program, we will make important several assumptions. Pay close attention to these assumptions when writing your code:

- There are only two clients and one server
- Commands do not appear in the middle of a message
- Clients must REGISTER before they can BRIDGE
- During CHAT mode, peers are **either** "writing" or "reading" at any given time
    - Only one client will initiate chat
    - Only one user writes at a time
    - Remember: read->write->read->write: look at Example 1

- **TCP Connections:**
    - Only oneTCP connection is open at a time, i.e., Client-Server and Client-Client maintain only one TCP connection at a time.
    - Client-Server: TCP connections between the server and client <u>are closed after each request/response</u>
        - Clients close the TCP connection with the server after any operation
    - Peer-Peer: peers in chat mode keep the TCP connection open during chat and close the connection only when chat is finished.
- The chat messages will only include printable characters that do not cause line breaks: letters, digits, spaces and symbols (e.g. !@#$).

5.7 Terminal output messages

<u>Server program</u>:

- When the server is started, it reports to terminal (stdout):

  `>Server listening on <ip address>:<port>`

  Example: `>Server listening on 200.26.180.43:2000`

- According to the incoming message type, Server writes to terminal:

1. REGISTER→ server output:

   `REGISTER: <clientID:name> from <ip address>:<port>`

   Example: `>REGISTER: student1 from 127.0.0.1:2000 received`
2. BRIDGE → server output:

   `BRIDGE: <peer 1 contact info> < peer 2 contact info>`

   Example output if student2 requests peer contact info:

   `>BRIDGE: student2 127.0.0.1:2000 student1 127.0.0.1:2500`

- The server writes to the terminal details of the clients when it receives /info
  /info → server output:
  `<clientID> <clientIP:clientPort>`
  Example:
  `>Bob 128.64.32.1:9000`
  `>Alice 128.200.34.10:8000`

Client program:

- When the client is started, it reports to terminal (stdout):
  `<clientID> running on <ip address:port>`
  Example: `>student1 running on 127.0.0.1:15000`

- Incoming CHAT request
  >Incoming chat request from <peer 1 contact info>
  Example output:
  `>Incoming chat request from student2 127.0.0.1:2000`

5.8 Example Usage and Chat Conversation Examples

While in CHAT mode the two clients exchange messages directly over a TCP connection. Messages should be printed as they arrive. Remember that only one peer will be typing at a time and each peer's mode alternates between reading and writing. The communication should work as indicated in the conversation depicted below in Example 1 and not crash or lose any characters.

Example 1:

Three terminals were opened and the commands were entered sequentially, e.g. first terminal 1, then terminal 2, then terminal 3 (refer to Section 5.3: Running the Programs to review command line arguments).

- Terminal 1: start Server **first**
- Terminal 2: start Peer 1 (e.g., Alice)
- Terminal 3: start Peer 2 (e.g., Bob)

Alice's Terminal **/register and /bridge entered BEFORE Bob's terminal.**	**Bob's Terminal** **Bob initiates a simple chat - invite for coffee.**

`>/id`
`>Alice`
`>/register (1)`
`>/bridge (2)`
	`>/id`
	`>Bob`
	`>/register (1)`
	`>/bridge (2)`

>Incoming chat request from Bob 127.1.1.1:1500 >Hello Alice (3) >How are you doing, Bob? >Doing well, Alice. Time for coffee? >Sorry Bob, I'm too busy. >Okay Alice, another time. >/quit (4) Program terminates	>/chat (3) >Hello Alice >How are you doing, Bob? >Doing well, Alice. Time for coffee? >Sorry Bob, I'm too busy. >Okay Alice, another time. (4) Program terminates
(1) Receives REGACK (2) Receives **empty** BRIDGEACK - goes into WAIT (3) Chat initiated by Bob, Alice has accepted TCP connection (4) Alice hangs up, QUIT message is sent to Bob. Alice closes TCP connection.	(1) Receives REGACK (2) Receives BRIDGEACK **with Alice's information** (3) Chat initiated by Bob (TCP handshake), and Alice accepts. (4) Bob receives a QUIT message from Alice. Bob exits, closing any open sockets.

Example 1: Chat program operation, as seen in the client terminals, between clients Alice (Terminal 2) and Bob (Terminal 3). Note that Bob initiates the chat.

See Example 1 screenshots

Example 2: catching <ctrl C> See Example 2 screenshots

```
>/id
>ctrl C
```

Programs terminate

6 Testing your Chat Program

Testing your Client: You will be able to test your Client program in a few ways:

- We are providing you with a fully functional server that can be downloaded as an executable file and installed on the VM. Before running the server the file must have execute permissions – enter **chmod +x server**

  ./server -p=<enter a registered port number> runs the server.

  First, verify that all of the message types sent to the server are formatted correctly - i.e. when a message is sent to the server, check that the server receives it

correctly. Then, verify that the server's response message is read correctly by your client. Finally, work on getting the chat between peers to work.

- You will also be able to test your client program by running your code on the Git pipeline. Instructions coming.
- Of course you are also expected to do your own unit testing.

Testing your Server: You can test the server you developed in a few ways:

- Use `netcat` to send messages and see the resulting output with print statements. There will be a demo of this process during the lab.
- Testing on the Git pipeline. Instructions will be provided soon.

7 Questions

1. Every protocol needs a name - what do you want to call this chat protocol?
2. What is your complete CHAT message here? What headers did you use (if any) and how were they used?
3. At what layer of the TCP/IP protocol stack is the Chat Protocol? Explain your answer.
4. Referring to the textbook Section 2.1.1:
    a. Looking over the architecture in this project, would you characterize it as Client-Server or something else? Explain your answer.
    b. When in Chat mode, do the peers operate as clients, servers or both? Explain your answer.
5. Is the client-server client communication using a persistent or non-persistent TCP connection? Explain your answer.
6. Is your peer-peer client program using a persistent or non-persistent TCP connection? Explain your answer.
7. Why do you think the specific type of TCP connection (persistent or non-persistent) was made for each of the two connections, client-server versus peer-peer?
8. Which aspect of your program follows half-duplex communication?
9. Include screenshot(s) showing all three terminals, clearly displaying:
    - Start up of your server
    - Chat exchange of at least 3 lines between peers
10. Consider the simplifying assumptions for this assignment. Which simplification in particular needs to be addressed for the server to function as a real server?

8 Grading

Project Demo: Each group will meet with one of the TAs during an assigned time slot. You will be asked to run your programs and discuss your code design. Missing the demo will result in a score of 0. Once you sign up for a demo slot this is your commitment and it cannot be missed without prior rescheduling agreement with the TA.

Rubric:

Grades are allocated using the following guideline:

Item	100 Points Total
State Diagram during lab Week 7 and Quiz – INDIVIDUAL	5
Part 1: Client Requests to Server	10
Part 2: Complete Client and Server code. Functionality (including error handling) and tests performed prior to demo on Gitlab pipeline. Details to come. Each submission will be tested to make sure it works properly and can deal with errors. Equal distribution of points between Client and Server.	40
Questions	15
Live TA Demo	30

Honor Code: All the work must be your own - please remember to cite any sources in your code (comments). MOSS will be run on all submissions and any instances of plagiarism agreed upon by the teaching team will result in a score of 0.

9 Resources

Writing and Reading from Socket:

- Computer Networks: A Top Down Approach- Ch2 Section 7: Sockets
- https://pythontic.com/modules/socket/send
- https://www.geeksforgeeks.org/socket-programming-python/
- Exception Handling in Python
  https://www.programiz.com/python-programming/exception-handlin