Dulimarta:

Menu On this page

TIP

As part of periodic assessment required for ABET accreditation, our course must be assessed for student teamwork. Hence, you are strongly encouraged to complete this assignment in a group of 1-3 students. You will earn extra point by working in group of 2 or 3 students.

Objectives

- Use TCP/UDP sockets for exchanging data among several programs
- Use multi-threading to manage concurrent control flows in a program
- Use a simple text-based UI (TUI) or graphical UI (GUI)

Overview

It is very likely that you are already familiar with many team/group chat apps which have been increasingly popular within the last decade, such as:

- Slack
- Discord
- Google Chat
- Telegram
- WhatsApp
- *... and many more ...*

These apps allow users to have interactive conversations with many participants in the group. In this programming assignment your team will create a *very simple group chat*

application, with significantly fewer features compared to those listed above. For instance your chat app is **NOT** required to implement the following features:

- Workspace/Channel management (as in Slack)
- Direct message
- Communication via voice/video
- File attachments
- User preferences
- Secure messaging
- Chat storage
- Chat bots
- *... and many more ...*

Minimum Requirements

General Requirements

1. You may write your program in any language of your preference. Despite your language of your choice, both the server and client programs must be developed using TCP sockets.

   > **Important**
   >
   > You are NOT allowed to write your program using the SocketiO library, otherwise the chat traffic management becomes too trivial.

2. Since your server does not save conversation history, (new) users who join the group chat late will not see previous conversations; they will only see conversations after they join

3. All users connected to the server, by default, are in the same global space. There is no "channels", "rooms" that break up users to different groups.

4. Your chat application will consist of one instance of a server program and multiple instances of a client program (running on different hosts/IP addresses). To join the group chat, each user will invoke the client program for sending and reading messages
5. Use a git repository to maintain your group development

Server Requirement

6. The server program shall be listening to a port number of your choice
7. The server shall maintain appropriate data structures to manage the active connections.
8. The server shall be designed to handle any number of chat participants which may come and go at any time. Specifically, arrival of new participants in the middle of on-going chat should not disturb existing participants. Likewise for departure of existing participants. In both cases, the server shall continue to operate without errors.
9. The server's role is as a "message reflector", it accepts messages from one chat participant and then send them to the other (active) chat participants
10. After a client connection request is accepted, the server shall spawn a new thread to handle all the communication with that particular client. This implies when there are N active chat participants, the server will be executing N+1 threads (including the main initial thread of the server itself). The following snippet shows the overall structure of using multiple threads.

python

```python
from threading import Thread

def handleClient(sock):
  # Handle communication with one client

  # Remember to close the socket when done
  sock.close()

server_socket.listen(___)
while some_condition_to_check_here:
  connection_socket, _ = server_socket.accept()
  t = Thread(target = handleClient, args=(connection_socket,))

```

```
    t.start()
server_socket.close()
```

Client Requirements

11. At the start of client run, prompt the user to enter his/her name. The name shall be attached to every message that originates from this user. So the other participants know who writes each message.
12. While the server program has no UI, the client program shall be designed with a UI (text-based UI or GUI) which clearly separates the conversation traffic from the input area for entering (and sending) messages.
    - The input box for entering messages shall allow **at least two lines of text**, the inputs shall **automatically be cleared** when the message was sent to the server
    - The conversation/chat traffic shall automatically scroll up as new chats begin to fill up the screen space
    - Incoming messages shall not intermix with the message the user is currently typing
13. Use a separate thread to monitor the incoming conversation traffic from the socket, so messages from other chat participants can update the conversation traffic UI without waiting for the user to complete posting a message
14. You may pick any mechanism of your preference how a client notifies the server when it is terminating its chat connection
15. When the server terminates (abruptly) all the clients shall terminate gracefully

Extra Credit Features

- (1-4 pts) Improved UI presentations, extra points will be assigned based on the amount of logic needed to implement the improved UI
- (3 pts) Interpret URL in text and turn it into a clickable link.

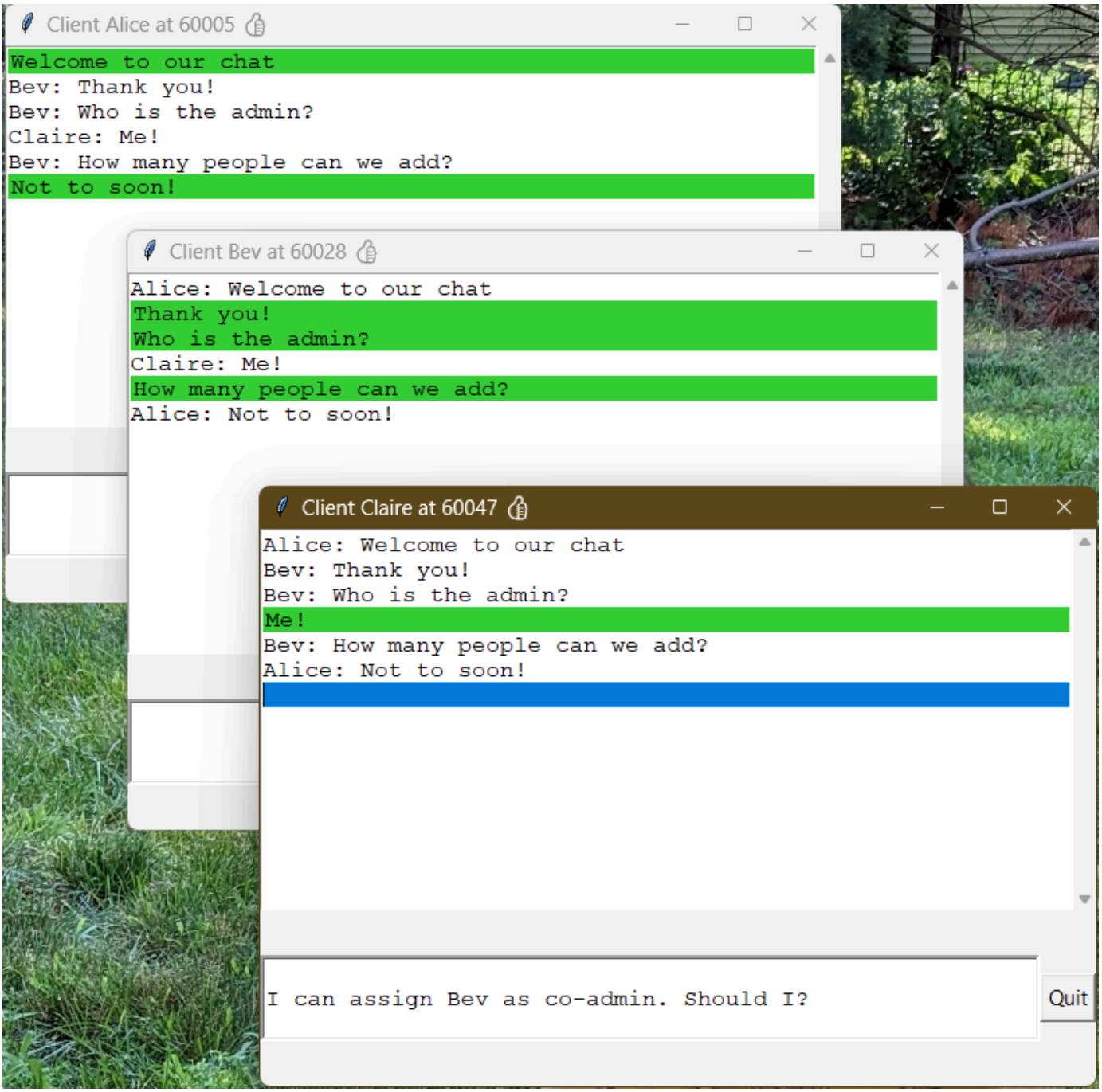
- (3-5 pts) Interpret Emoji shortcuts ala Slack such as `:thumbsup:` for 👍, replace the shortcut with the actual Unicode character when the message is posted. Many modern programming languages support Unicode constants specified with the `\u` prefix (followed by 4 hex digits) or `\U` prefix followed by 8 hex digits). For instance the "thumbsup emoji" is Unicode `\U0001F44D`.
  - Higher credit will be given if the emoji is shown as the user type them. This requires handling of individual keystrokes in the input area.
- (3 pts) Implement a direct message to a particular chat participant when a message is prefixed with `@username` such as `@Ben I stop by at 5p after work today`
- If you think of an extra feature not listed above, consult with your instructor prior to implementation

---

Sample Output

The following screenshots show screenshots of three chat participants Alice (top left), Bev (middle), and Claire (bottom right):

- The instructor's code is written in Python+TkInter
- The window title shows the user name and the port number
- Messages in green background originates from each corresponding sender (color coding is not required)
- Messages by other participants are tagged with the sender's name
- In the snapshot, Claire is currently typing "I can assign Bev as co-admin...."



Choices of UI

- For text-based UI (TUI), you can use the `curses` or other similar libraries
- For graphical UI, you can use `TkInter` or other similar libraries
- Web-based UI
  - Develop a web application that connect to the backend chat-server
  - Develop a desktop application that uses web-technology, such as Electron

WARNING

Among these choices, handling text scrolling in `curses` can be tricky.

Since most TUI/GUI framework implements its own thread to refresh the UI, incorporating another thread to perform background work may require extra setup logic. Most UI frameworks require UI updates be done from within the UI thread, i.e. it disallows UI updates from a non-UI thread.

In the following snippet:

- the function `read_socket`, which runs in a non-UI thread, acquires data (from a socket) which is destined for UI updates. When such data becomes available, it is pushed to a queue
- the function `update_gui` is designed to run once every 100 milliseconds and checks if the queue holds any (new) data provided by the socket reader thread

python

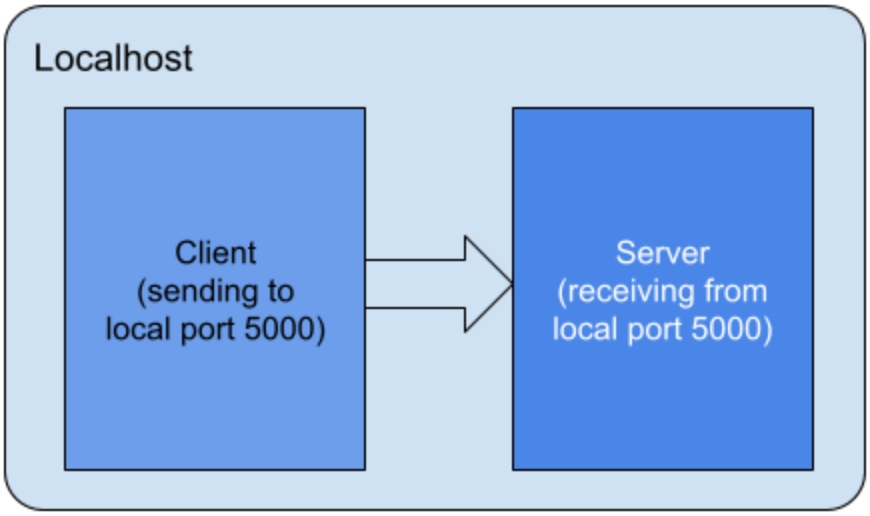
```python
import tkinter as tk
import threading
import socket
import queue

class App:
    def __init__(self, master):
        self.master = master
        master.title("Socket Reader")

        self.label_text = tk.StringVar()
        self.label = tk.Label(master, textvariable = self.label_text)
        self.label.pack()

        self.data_queue = queue.Queue()
        self.running = True

        self.socket_thread = threading.Thread(target=self.read_socket)
        self.socket_thread.daemon = True  # Allow program to exit even if t
        self.socket_thread.start()

        self.update_gui()

```

```
    def read_socket(self):
        host = '127.0.0.1'  # Replace with your host
        port = 12345        # Replace with your port

        try:
            with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
                s.connect((host, port))
                while self.running:
                    data = s.recv(1024)
                    if not data:
                        break
                    self.data_queue.put(data.decode())
        except Exception as e:
             self.data_queue.put(f"Error: {e}")

    def update_gui(self):
        try:
            data = self.data_queue.get_nowait()
            self.label_text.set(data)
        except queue.Empty:
            pass  # No data yet, ignore
        if self.running:
            self.master.after(100, self.update_gui) # Check every 100 ms

    def close(self):
        self.running = False
        self.master.destroy()


root = tk.Tk()
app = App(root)
root.protocol("WM_DELETE_WINDOW", app.close) # Handle window close event
root.mainloop()
```

Testing Your Chat Program

Testing on Localhost

Be sure your code work correctly when you run both the server and client on your own desktop. Assuming your server is listening (locally) on port 5000, you can run your client on two or three different terminals and chat with yourself. Technically the two (three) instances of the client program send their messages to local port 5000.



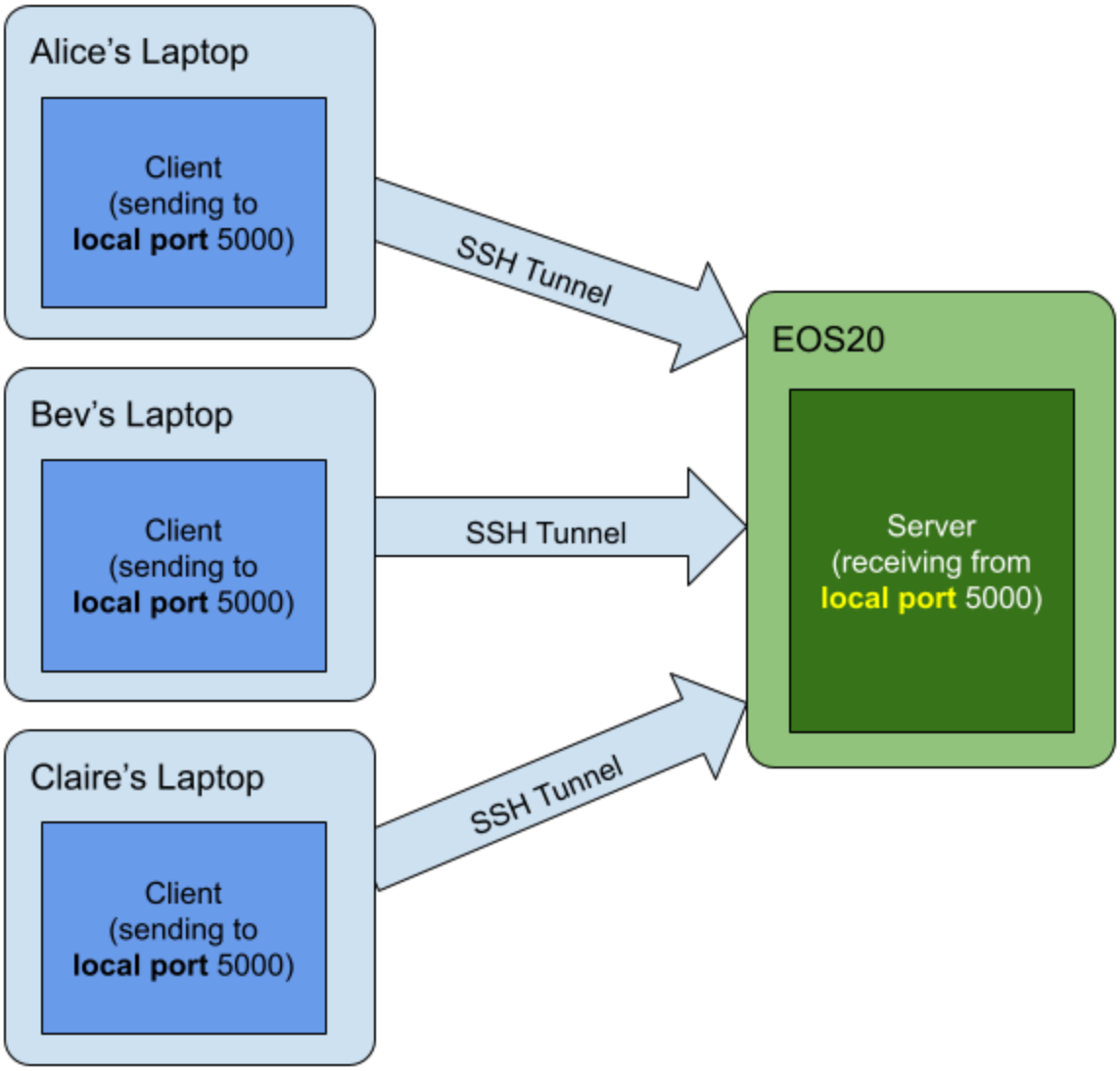
Actual Testing and Demo setup

To allow everyone in your group test your chat application, run the server program on one of our DC/EOS machines and run the client program from your individual laptops/desktops.

The following setup assumes that:

- Your chat server is running on `eos20.cis.gvsu.edu` listening to port 5000
- Your client is running on your laptop(s)

> **Important**
>
> For the actual testing explained below, both the client and server will continue to operate as if all the communication traffic happens on the localhost (or IP 127.0.0.1). Practically, if both the client and server work on the localhost, **without changing the IP address and port number** they should continue to work when they run on two different hosts.



1. Firstly, when you are not in campus be sure you are connected through our VPN

2. One of you in the team shall remote login to `eos20.cis.gvsu.edu` and start the server

```
# Do this on eos20.cis.gvsu.edu
python your-chat-server.py
```

> **WARNING**
>
> I strongly recommend you connect to EOS or Data Communication workstation via the command line (instead of using the graphical remote desktop connection). The RDP connection may sometime unpredictably kick you out for no reason.

3. Everyone in your team then performs the following actions

- From a terminal on your local computer, setup a secure tunnel that forwards port 5000 on `localhost` to port 5000 on `eos20`

```bash
# Keep the following running in background
# Replace zzzzzzzz with your EOS/DC userid
ssh -N -L 5000:localhost:5000 zzzzzzzz@eos20.cis.gvsu.edu &
```

If you are on Windows and ssh is not available, use this guide to enable it or alternatively you can use `plink` (part of Putty installation).

TIP

Use of SSH tunnels solves the issue in case (direct) access to port 5000 is prohibited by firewall rules on our EOS/DC machines. The above tunnel works as follows:

- The SSH client on your local machine will be listening on port 5000 for incoming packets.
- Incoming packets are then delivered to the SSH server (via port 22, the default SSH port) on EOS20
- The remote SSH server (on eos20) then sends the packet to eos20's port 5000
- Your server (which is listening on port 5000 there) will then respond

This technique works because access to port 22 (SSH) is not prohibited.

- Start the client on your local computer

```
python your-chat-client.py
```

Important

A demo of your program is **required** for grading. The Friday lab hours may be too short to check all the demo. In which case, your group should find and schedule a different time to demo your program in my office.

Deliverable

- Source code of the server and the client
- Demo of a live chat session on multiple hosts
- A documentation that includes
  - The overall design of the chat server: data structures and operations
  - The overall design of the chat client
  - Responsibility assigned to each team in your group, developer coordination (git repository)

Grading Rubrics (Tentative)

Grading Item	Point
Server data structure for managing chat connections	4
Adding new chat connections	3
Removing existing chat connections	3
The server uses one thread per chat connection	3
The server correctly relays incoming messages to the other chat participants	3
The client UI clearly separates the input message area from the chat traffic display	4
The chat traffic display scrolls automatically and correctly for messages of any reasonable length	3
Each message is tagged with the name who writes it	3
The client uses a separate thread for receiving messages from its socket	3
Chat participants can join and leave anytime, without disturbing the current chat traffic	3
Graceful client terminate when the chat server terminates abruptly	2
Live demo using multiple hosts	6
Documentation	10
Extra points for working in a group (of 2-3 students) and using git repository	2-4

Grading Item	Point
Extra feature: convert URL to a clickable link	3
Extra feature: improved UI	1-4
Extra feature: Emoji shortcuts	3-5
Extra feature: direct messaging	3

Last update:: 3/31/25, 10:47 AM

Next page
CS457 Home