Assignment 2: Peer-to-Peer Chat Application

CS 216: Introduction to Blockchain

06.02.2025

Assignment Deadline: 22nd February 2025 (Saturday) by 11:30 PM.

Submission Instruction

Each team should follow this submission rule. **Only one team member needs to create a github account, create a repository and share the link of the same.** If you fail to follow this rule then points will be deducted.

1. Create a GitHub repository with a meaningful name (mandatory: your team name in the repo name).
2. Add all relevant source files, including:
   - The main program source code.
   - Any additional configuration or dependency files.
   - A `README.md` file with instructions on how to run the program. **Mention in this file the name and roll numbers of all the team members.** Mention if your code also handles the bonus question.
3. Ensure your repository is public.
4. Submit the repository URL in the designated submission portal mentioned in Moodle.

Example Submission Format:

```
GitHub Repository: https://github.com/yourusername/your\_repository\_name
```

Problem Statement

Implement a peer-to-peer chat program in a language of your choice that enables simultaneous sending and receiving messages simultaneously, supports multiple peers, and allows users to query and retrieve the list of peers from which it had received messages. Multiple instances of the code can be run in seperate terminal environments to form a peer to peer chat network. We assume that

- The program requires the user to know the IP address and port numbers of other users beforehand.
- **You should (mandatorily) send message to these 2 set of IP address and port:**

- IP: 10.206.4.122, PORT: 1255
- IP: 10.206.5.228, PORT: 6555

Assume that these 2 IP addresses has the capacity to accept just 1000 connection requests.

- **Standardization of format of message send to receiver: To prevent inconsistency when you connect to other peers in the network**

  Your message send to the server must look like this:

  ``<IP ADDRESS:PORT> <team name> <your message>"

  For example: If you want to send the message "hello", IP address and PORT is 10.206.4.201 & 8080 respectively, and your team name is "blockchain", server should receive:

  ``10.206.4.201:8080 blockchain hello"

  As a server, it is your task to ensure that the server code parses this string, extracts out "10.206.4.201:8080", and stores it in the table of peers. I presume that you store the peer by their IP address and PORT number. The PORT you should send to the server should be a fixed PORT number. For example, peer1 runs in port 8080 and IP address is 10.103.45.601, and peer2 runs in port 9090 and IP address is 10.103.44.109. Peer1 has name P1 and peer2 has name P2. Peer1 sends message "hello, you there" to peer2, message displayed at peer2's terminal is

  ```
  10.103.45.601:8080 P1 hello, you there
  ```

  If your client by default depends on ephemeral port, then peer1 port will be some random number but ensure that the server on receipt of this message ignores that dynamically allocated port numbers and instead reads it from the message received.

  To check IP address check `https://www.avg.com/en/signal/find-ip-address`, for Linux systems `https://www.ionos.com/digitalguide/hosting/technical-matters/get-linux-ip-address/`.

  In C code, this is what is done when you receive the message from a client:

  ```
  client_socket = accept(server_fd, (struct sockaddr *)&address,
  (socklen_t *)&addrlen)
  ```

  By default you put store *address.sin_addr* and *address.sin_port* as the peer IP and PORT. However, if you are using an ephemeral port, then the PORT number changes every time you send a message to a server; I would suggest extracting IP and PORT from the message as explained above.

- **Double-entry of same (IP:PORT) in peer table to be strictly avoided.** This is because the peer table will help you to query your peer and send connection request to such peers. Double-entry will be a trouble.

- Client-server port dilemma:

  - **If you are not comfortable using ephemeral ports, and found it confusing then use a fixed port number for client (different what you had chosen for your server) and connect it to your server. Format will remain unchanged**.

  - If you are using ephemeral ports and the client of one peer had previously connected to the server of another peer, and now this client sends another message to the server, the server must treat it as a single connection. Repeated entries based on changing ephemeral ports of client should be avoided. For this, peers must explicitly communicate their listening ports (i.e. port number of their server) to allow reconnections.

- Clarification regarding query() function: Only retain the list of peers from whom you have directly received message or who has established a connection (upon adding an additional connect() function) with your peer.

I would suggest try developing this assignment using TCP
Bonus Question: Once your query the list of peers it will output the list of IP:PORT from whom you received messages. Let this list be denoted by X. Write a connect() function so that it connects to active peers and send a connection message to the peer with whom you have established connection. Now in your terminal, if you query the peer, this IP address and PORT will also show up.

Your output interface should then looks like this:

```
Enter your name: <write your team name as mentioned in google form>
Enter your port number: <Choose a port number of yours>
Server listening on port <the output is a port number>
***** Menu *****
1. Send message
2. Query active peers
3. Connect to active peers
0. Quit
```

For example, if peer1 has IP:PORT as 10.206.5.228:6555 and peer2 has IP:PORT as 10.206.4.201:1255, and peer1 sends two messages to peer2, then peer2 should treat this as one peer that is sending message and must not make double-entry in the list due to peer1's client picking up an ephemeral port (this gets handled if you are using fixed port). After connect() is send from peer1 to peer2, peer2 is reflected as neighbor of peer1 when querying for the peer's of peer1.

Querying a Peer

Querying a peer refers to the process of discovering available peers in a P2P chat system. This is typically done using a discovery protocol to request information about active peers. The main purpose of querying is to gather a list of online peers before establishing direct communication.

For example, a peer may send a "Who is online?" message using 'sendto()' and receive responses from other peers using 'recvfrom()'. The response includes details of active peers, which can be used to establish a connection later.

Connecting to a Peer

Connecting to a peer involves establishing a direct communication channel between two peers. This is usually achieved using a TCP connection with the 'connect()' function, or through direct UDP messaging once the peer's address is known.

Once a peer is identified through querying, a connection request is sent to initiate communication. If successful, the peers can exchange messages in real-time.

On Ephemeral Ports: A peer can act as both a client and server on the same port by utilizing the underlying network programming functionalities to simultaneously listen for incoming connections (server behavior) while also initiating outbound connections to other peers (client behavior), effectively allowing it to both send and receive data on the same port number; this is a fundamental principle of peer-to-peer (P2P) networking where each node functions as both a client and a server. In terms of C code: When a peer is acting as a server (bind() and listen()) in TCP, the OS allows it to accept multiple connections on the same port from different source IPs/ports. When the same peer acts as a client, the OS assigns a new ephemeral port for the outgoing connection, ensuring it does not conflict with the listening port. Servers don't need to know the ephemeral port; they just listen on a fixed port and automatically learns the client's ephemeral port during connection establishment. The TCP handshake automatically exchanges port information, so replies are routed correctly. Peers can use the same port for listening while using ephemeral ports for outgoing connections.
For UDP, a peer can use the same port for both sending and receiving because UDP is connectionless and does not establish dedicated connections like TCP. Instead, UDP sockets simply send and receive packets (datagrams) independently. For TCP vs UDP: check `https://www.geeksforgeeks.org/differences-between-tcp-and-udp/`

Simultaneous send and receive: The program achieves simultaneous send and receive by running the receive method on seperate thread (use pthread, if you are doing C programming). The program involves the use of select() system call (if using C program) to identify the ready file descriptors and loop over them to receive the messages in queue.

Key Features:

- Simultaneous Send and Receive: The receive function runs in a separate thread.
- Use of *select()* for managing file descriptors: Allows detecting ready sockets for I/O operations.
- Run-time IP address and Port Input: Each peer can specify the target IP and port at run-time, making the program suitable for multi-host environments.
- Maintain a List of Active Connections: Use a data structure, like an array or a linked list, to store details of connected peers (e.g., IP address, port, and socket file descriptor).
- Update the List on New Connections: When a new client connects to the server by sending a message, add its details to the list.
- Remove Disconnected Peers: When a client disconnects by sending an "exit" message, remove its details from the list.
- Add a Query Functionality: Implement a feature (e.g., a menu option) to display the list of connected peers. If a peer had send an "exit" message, it should not be displayed in the list.

Implementation Plan

- Networking Model: You can use sockets (TCP) for communication. Implement a peer discovery mechanism, possibly by maintaining a local list of connected peers.
- Peer Node: Acts as both a sender and receiver. Manages communication with multiple peers. Stores a list of peers it has interacted with.

- Message Handling: Messages should be sent and received without blocking other operations. Use a separate thread for receiving.
- Language Choice: Preferred one's

```
Python (Using socket, threading, or asyncio)
Java (Using java.net and multithreading)
C or C++
Node.js (Using net module with event-driven programming)
```

Running instructions example for C program: The program was executed on a Linux system using the gcc compiler. Suppose you save the file as peer.c then do

```
Open 4 terminals and run this
gcc peer.c -o peer1 (in terminal 1)
./peer1
gcc peer.c -o peer2 (in terminal 2)
./peer2
gcc peer.c -o peer3 (in terminal 3)
./peer3
gcc peer.c -o peer4 (in terminal 4)
./peer4
```

Sample Workplan: peer1 set port number 8080, peer2 set port number 9090, peer3 set port number 7070, and peer4 set port number say 6060. peer1 sends message to peer2, for that peer1 sets IP address of peer2 (i.e. recipient) to 127.0.0.1 and port number to 9090, then it sends a message say "hello". When you query peers of peer2, it will show IP address and port number of peer 1. Since you are running on localhost, IP address of all peer remains the same i.e. 127.0.0.1 but port number will be different. Repeat the same where peer3 sends message to peer4, peer2 sends message to peer4, peer4 sends message to peer1, peer1 sends message to peer3.
If you query peers for each following will be the output:

- peer1: peer4
- peer2: peer1
- peer3: peer1
- peer4: peer2, peer3

Now if peer1 sends "exit" message to peer3 then terminal 3 will show that peer1 got disconnected and the output will change. If you query peers for each following will be the output:

- peer1: peer4
- peer2: peer1
- peer3: No connected peers
- peer4: peer2, peer3

Your output interface should looks like this:

```
Enter your name: <write your team name as mentioned in google form>
Enter your port number: <Choose a port number of yours>
Server listening on port <the output is a port number>

***** Menu *****
1. Send message
2. Query active peers
0. Quit

Enter choice: 1

Enter the recipient's IP address: <IP address of the peer node>
Enter the recipient's port number: <port number of the peer node>
Enter your message: <send any message>
```

On the server side, the message displayed is

```
<client IP:client PORT> <client message>
```

N.B: If you send "exit" as message, it will just disconnect the peer at the other end.

```
***** Menu *****
1. Send message
2. Query connected peers
0. Quit

Enter choice: 2
<If there are any connected peer, display below>
Connected Peers:
<Peer number and IP:PORT>
<If there are no connected peer, display below>
No connected Peers

Enter choice: 0
Exiting
```

Reference Material:

1. Dick Buttlar, Jacqueline Farrell, Bradford Nichols - PThreads Programming A POSIX Standard for Better Multiprocessing-O'Reilly
2. Sukanta Das - Socket Programming through C
3. `https://www.geeksforgeeks.org/socket-programming-cc/`