

UNIVERSITY OF SCIENCE AND TECHNOLOGY OF HANOI



A REPORT ON

Peer-to-Peer Chat System based on Socket

Course: Distributed System

BY

Dao Xuan Quy
Ho Cong Thanh
Nguyen Son
Lai Duc Thinh

BI12-379
BI12-409
BI12-389
BI12-426

**DEPARTMENT OF INFORMATION TECHNOLOGY AND
COMMUNICATION**

Hanoi, April 2024

Contents

| | | |
|----------|----------------------------------------------------------------|-----------|
| 1 | Introduction | 1 |
| 2 | Organizing the system | 2 |
| 2.1 | Designing the protocol | 2 |
| 2.2 | Organizing the system | 3 |
| 3 | Implementation | 5 |
| 3.1 | Create socket and configure server address | 5 |
| 3.2 | Split thread for both receiving and sending messages | 6 |
| 3.3 | Create sending function | 6 |
| 3.4 | Create Receiving Function | 7 |
| 3.5 | Building Guidelines | 9 |
| 4 | Conclusion & Future Enhancements | 10 |
| 4.1 | Conclusion | 10 |
| 4.2 | Future Enhancements | 10 |
| | References | 11 |

1 Introduction

Peer-to-Peer (P2P) networking challenges traditional hierarchical models of digital communication. Rather than rigid client-server structures where roles are clearly defined, P2P promotes egalitarian exchange between equally empowered peers. By dismantling centralized control and distributing information across all participating devices, P2P networks demonstrate revolutionary resiliency and efficiency [1].

Nowhere is the value of decentralized, real-time interaction clearer than in chat systems. Enabling immediate dialogue through text, voice, and video, chat underpins personal connections, professional collaboration, and customer support worldwide. As a cornerstone of modern discourse, chat epitomizes the responsive interconnectivity we have come to expect.

However, seamless peer exchange requires sophisticated coordination behind the scenes. Our research delves into this technical underpinning through a P2P chat application. Rather than a black box approach, we explore the mechanisms that make instant messaging possible at the network level. Specifically, we examine how the SOCKET programming interface empowers transparent two-way communication between distributed peers [2] [3].

By implementing a basic text chat, we showcase P2P capabilities in practice. Both conceptual understanding and hands-on development provide insight into how SOCKETS actualize decentralized, reciprocal interaction at its foundations. Our goal is a robust yet understandable implementation highlighting P2P principles in action. Through this technical analysis, we aim to illuminate the infrastructure supporting modern connectedness across diverse, distributed forms worldwide.

2 Organizing the system

2.1 Designing the protocol

In this P2P system both peers could act as clients or servers, therefore, the protocol between two peers could be described as follows:

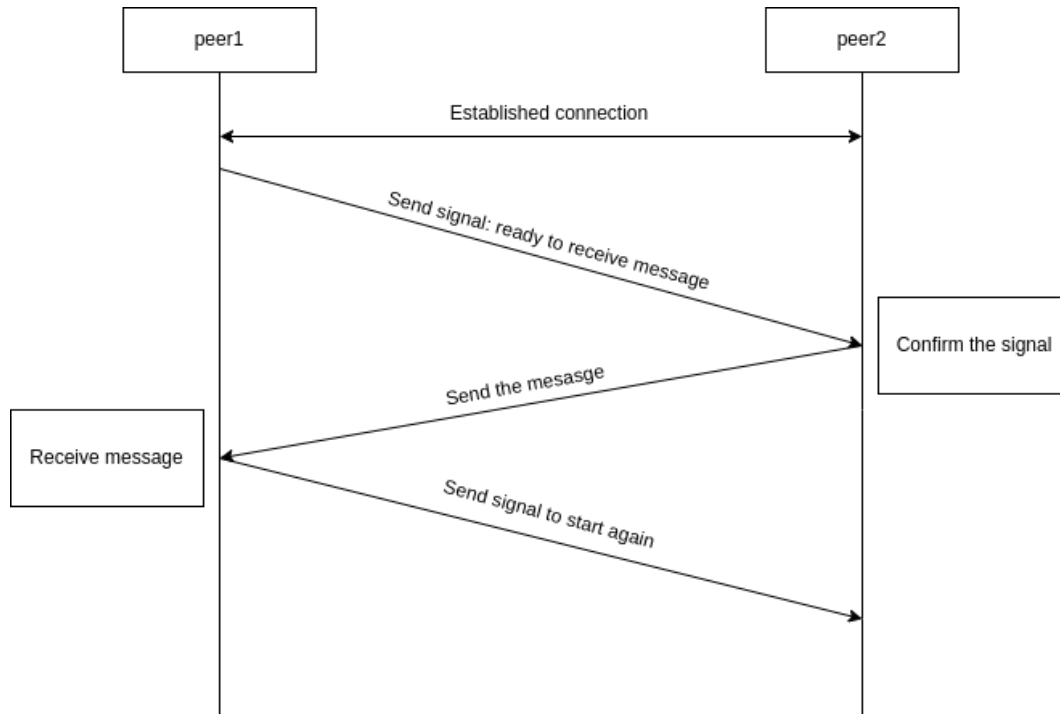


Figure 1: Peer-to-Peer Communication Framework

In Figure 1, the initial step for both users—referred to as 'peer1' and 'peer2'—is to accurately enter the other's unique port number. This action is akin to dialing a phone number; it is crucial for establishing a direct line of communication. If 'peer1' enters 'peer2's port number correctly, and 'peer2' reciprocates, a handshake occurs over the network, resulting in a successful connection.

This mutual acknowledgment of port numbers initiates the "Established connection" phase as depicted in the diagram, signifying that a private channel for data exchange is now open. Once this secure link is established, 'peer1' signals readiness to receive a message. This is an important step to confirm that 'peer1' is prepared to handle incoming data, thereby preventing loss of information.

'Peer2', upon receiving the ready signal, proceeds to send the text message. In this P2P system, the focus is solely on text-based communication, ensuring that the transmission is quick and efficient. After 'peer1' receives the message, a confirmation signal is sent back to 'peer2'. This step is crucial as it validates the successful delivery of the message, ensuring both sides that the communication protocol is functioning as intended.

The exchange of messages can continue in this manner, with each peer waiting for a ready signal before sending a message, and confirming receipt afterward. The conversation flow is thus maintained, with both peers engaged in a structured dialogue.

At any point, either peer can choose to conclude the conversation. To do this, the peer simply ceases to send or acknowledge messages, which signals to the other peer that the conversation has ended. Once the decision to disengage is mutual or one-sided, the program ensures that the connection is cleanly terminated. This final step in the protocol ensures

that all resources are properly released and that both programs are gracefully closed, signifying the end of the P2P communication session.

2.2 Organizing the system

The setup involves peers communicating via sockets. Both need to initialize a socket, username, and port number to identify the users. The peer performs both tasks: listens to upcoming messages and sends messages to others' ports :

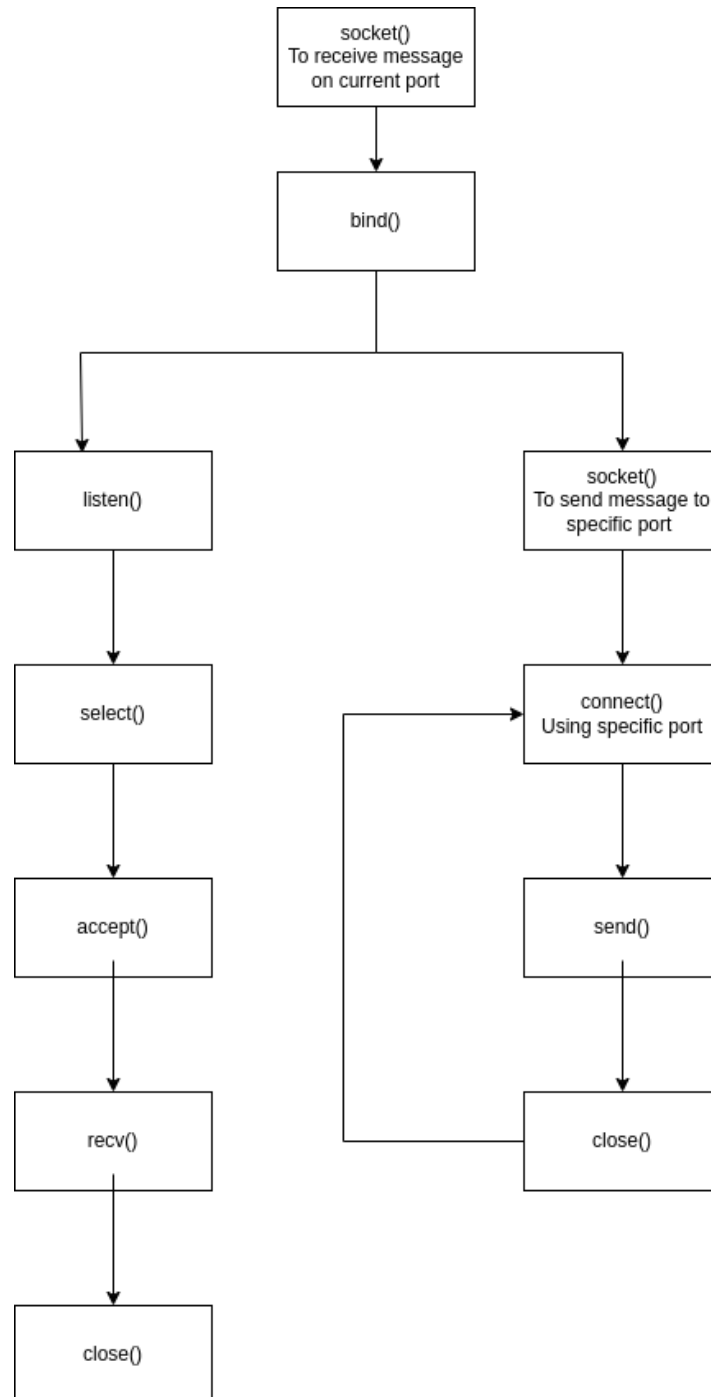


Figure 2: Peer-to-Peer Organizing System

In this peer-to-peer setup, each peer initiates the communication process by creating a

socket with the `socket()` function, laying the foundation for sending and receiving messages on their respective current ports. This is the first step in preparing the system for network communication.

Each peer then binds their socket to an address and port with the `bind()` function. This step associates the socket with the peer's current network interface and port, ensuring that it's ready to listen for incoming connections on that specific port.

Following the binding, the peers enter a listening state using the `listen()` function, indicating that they are now prepared to accept incoming connection requests. This function sets up peers to monitor the network for attempts by others to connect.

The `select()` function is then called, which is crucial for handling multiple connection channels. This allows the peer to manage multiple communication streams, potentially coming in and letting it respond to the one that is active, effectively managing its resources.

When a connection attempt is detected, `accept()` is employed by the listening peer to formally accept the incoming request. This step finalizes the establishment of a connection between the two peers, and a communication channel is opened.

The peer wishing to send a message will use the `connect()` function to establish a connection to the specific port of the receiving peer. Once this connection is made, `send()` is invoked to transmit the text message across the established channel.

After the message is sent, both peers use the `recv()` function to receive any incoming messages. This function is vital for the actual data exchange and signifies that the peer is ready to process received data.

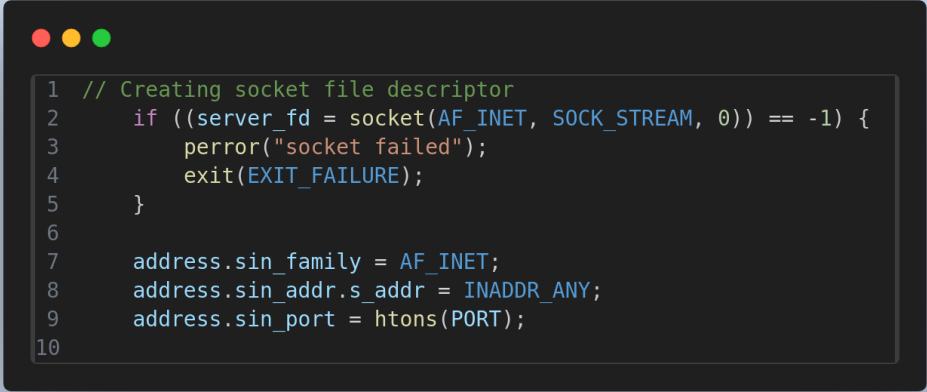
Finally, once the message exchange is completed, or the peers decide to end the communication, the `close()` function is called. This function closes the socket, effectively ending the session and releasing the associated resources.

Both peers independently execute these functions in a similar sequence, with the ability to both send and receive messages, encapsulating the truly decentralized and cooperative essence of P2P networks. Each step must be successfully completed before proceeding to the next, ensuring a robust and orderly communication process.

3 Implementation

3.1 Create socket and configure server address

First, we need to create a socket and configure the server address:

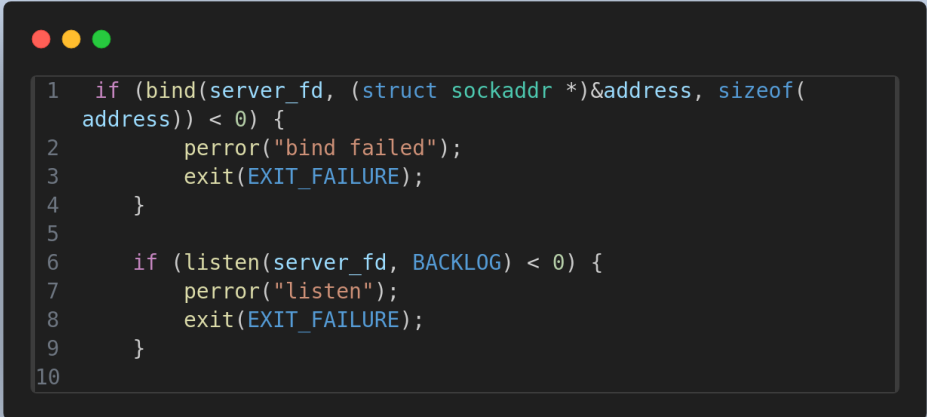


```
1 // Creating socket file descriptor
2 if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
3     perror("socket failed");
4     exit(EXIT_FAILURE);
5 }
6
7 address.sin_family = AF_INET;
8 address.sin_addr.s_addr = INADDR_ANY;
9 address.sin_port = htons(PORT);
10
```

Figure 3: Creating socket and configure server

We utilize the system's command `bind()` to attach the server socket to the designated server address. This action links the socket to a specific IP address and port number on the server, establishing a communication channel. If the binding process encounters any issues, an error message is displayed to alert the user, and the program is terminated to prevent potential errors.

The `listen()` function acts like a waiter at a restaurant, ready to seat incoming clients (connections). There's a problem starting this waiting process, an error message is shown, and the program shuts down to avoid issues.



```
1 if (bind(server_fd, (struct sockaddr *)&address, sizeof(
  address)) < 0) {
2     perror("bind failed");
3     exit(EXIT_FAILURE);
4 }
5
6 if (listen(server_fd, BACKLOG) < 0) {
7     perror("listen");
8     exit(EXIT_FAILURE);
9 }
10
```

Figure 4: Bind socket and start listening to the signal

3.2 Split thread for both receiving and sending messages

Since the peer needs to handle both tasks receiving and sending messages. Therefore we need to create a new thread used for receiving messages (which will be explained in the next figure):

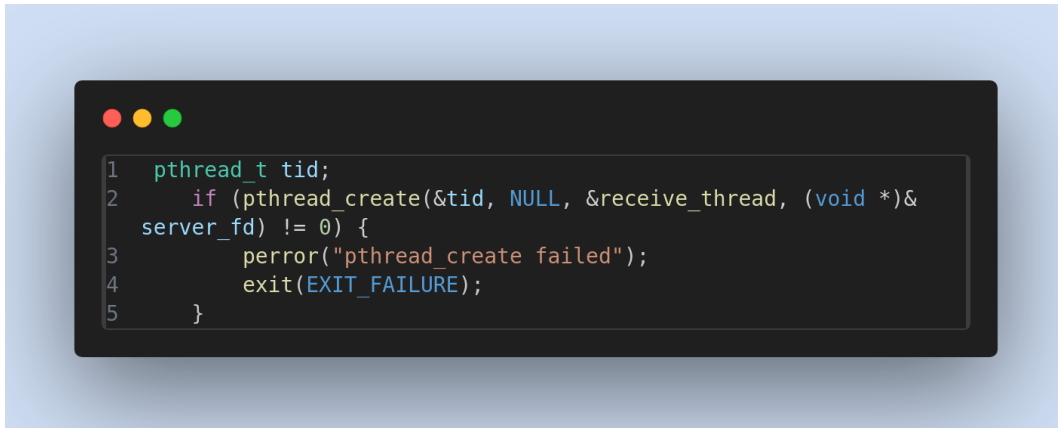


Figure 5: Dividing a thread for receiving messages

3.3 Create sending function

The next step is to implement the sending function. In the sending function, first, we need to reach the desired PORT (or the person we want to talk to). After the connection is established, the built-in send() function will be used to deliver the message the user writes (in the terminal for instance).


```

1 void sending(int PORT_server) {
2     char buffer[MAX_MESSAGE_LENGTH] = {0};
3     int sock = 0;
4     struct sockaddr_in serv_addr;
5     char hello[1024] = {0};
6     if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
7         perror("Socket creation error");
8         return;
9     }
10
11     serv_addr.sin_family = AF_INET;
12     serv_addr.sin_addr.s_addr = inet_addr("127.0.0.1");
13     serv_addr.sin_port = htons(PORT_server);
14
15     if (connect(sock, (struct sockaddr *)&serv_addr, sizeof(
serv_addr)) < 0) {
16         perror("Connection Failed");
17         close(sock);
18     }
19
20     scanf("%[^\n]%*c", hello);
21     sprintf(buffer, "%s[PORT:%d] says: %s", name, PORT, hello
);
22     send(sock, buffer, strlen(buffer), 0);
23     close(sock);
24 }

```

Figure 6: Sending function

3.4 Create Receiving Function

After successfully sending the messages, now the task is now developing a receiving function. Within the infinite loop, the select system call efficiently monitors all sockets in the current sockets set. When select detects activity on a socket, the program iterates through each descriptor to determine the source. If the activity originates from another peer's socket (call it server for now since plays the role of server for instance), it indicates a new peer (call it client for now plays the role of client for instance) attempting to connect. The server accepts the connection, adds the new client's socket to the monitored set, and continues to listen for further connections. If the activity comes from a client socket, it signifies an incoming message from that client. The server receives the message, checks for errors or disconnections, and finally prints the received message to the console.

```

1 while (1)
2 {
3     k++;
4     ready_sockets = current_sockets;
5
6     if (select(FD_SETSIZE, &ready_sockets, NULL, NULL,
7 NULL) < 0) {
8         perror("select() error");
9         exit(EXIT_FAILURE);
10    }
11
12    for (int i =0; i < FD_SETSIZE; i++){
13        if (FD_ISSET(i, &ready_sockets)){
14
15            if (i == server_fd) {
16                int client_socket;
17                if ((client_socket = accept(server_fd, (
18 struct sockaddr *)&address,
19                                     (socklen_t *)&
20 addrrlen)) < 0) {
21                    perror("accept");
22                    continue;
23                }
24                // Continue to next iteration to avoid closing an uninitializ
25                ed socket
26
27                FD_SET(client_socket, &current_sockets);
28            }
29        }
30    }
31 }

```

Figure 7: Receiving multiple connections from peers

Then the message is read by the built-in `recv()` function and print out the terminal.



Figure 8: Read and print messages

The loop is stopped if the number of clients reaches twice of the maximum client's numbers that the socket API can handle.

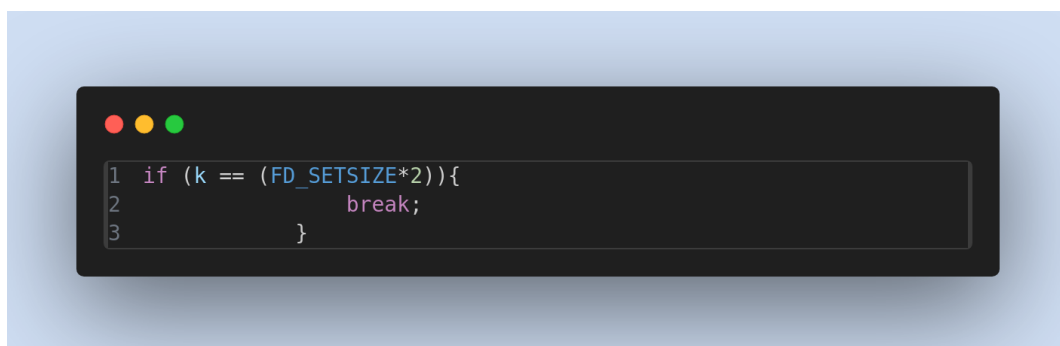


Figure 9: Loop's breakpoint

3.5 Building Guidelines

Open two terminals: In first terminal: `gcc peer.c -o peer1 ./peer1` In the second terminal: `gcc peer.c -o peer2 ./peer2`

Set name and port for both peers before entering the port to send messages. After the connection is established. You can now start texting.

4 Conclusion & Future Enhancements

4.1 Conclusion

Our exploration into Peer-to-Peer (P2P) chat systems through the implementation of a SOCKET-based communication protocol has underscored the efficacy and robustness of decentralized networks. By successfully establishing a functional chat system where each node can serve both as a client and a server, we demonstrated the practical utility and dynamic adaptability of P2P architectures. Our implementation facilitated real-time text communication, highlighting the system's responsiveness and reliability under varied network conditions.

The project not only provided us with hands-on experience in SOCKET programming but also deepened our understanding of network communication protocols, threading, and real-time data exchange. Through the phases of designing, implementing, and testing our chat system, we encountered and overcame numerous challenges, such as ensuring data integrity during transmission and managing asynchronous communication.

4.2 Future Enhancements

Despite the success of our project, there are several avenues for enhancement and further research:

1. **Security Enhancements:** Currently, our chat system transmits data in plain text, which could be susceptible to interception and unauthorized access. Implementing end-to-end encryption would safeguard privacy and enhance security [4].
2. **File Transfer Capability:** Extending the chat system to support file sharing would increase its utility, allowing users to exchange not only messages but also documents, images, and other media.
3. **User Interface Development:** Developing a more intuitive and feature-rich graphical user interface (GUI) would make the system more accessible to users, improving the overall user experience.
4. **Scalability Testing:** More rigorous testing on scalability and performance under high loads is essential to understand the limitations of the current implementation and to optimize it for larger networks.
5. **Network Efficiency:** Investigating more efficient networking techniques and algorithms could reduce latency and increase throughput, especially in scenarios with high traffic and large numbers of users.

By addressing these enhancements, we can extend the capabilities of our P2P chat system, making it more secure, versatile, and user-friendly. This continuous improvement will also provide further educational opportunities in network programming and system design.

References

- [1] Tanenbaum, A. S., & Wetherall, D. J. (2011). *Computer Networks* (5th ed.). Prentice Hall.
- [2] Stevens, W. R. (1994). *UNIX Network Programming, Volume 1: The Sockets Networking API* (3rd ed.). Addison-Wesley Professional.
- [3] Garcia, L. M., & Bessiere, C. (2002). Network performance and quality of service in peer-to-peer networks. *Journal of Network and Systems Management*, 10(2), 123-142.
- [4] Zhou, S., & Zhang, L. (2005). Security in peer-to-peer networks: A system framework perspective. *IEEE Network*, 19(5), 4-10.