Multi-Threaded Ping Pong Chat Application with Relay Server

PARALLEL & DISTRIBUTED COMPUTING

Muhammad Owais Zameer | Project Report | 04-Jan-2024

Group Members

1. Muhammad Nomir | 11330 (Leader)

2. Parkash | 11089

Introduction

The purpose of the MPI (Message Passing Interface) server, client, and relay server application is to facilitate distributed communication and coordination between multiple computing nodes in a parallel computing environment. This system is designed to allow different processes to exchange messages, share information, and collaboratively solve complex problems. The significance of this application lies in its ability to harness the power of parallel processing, enabling improved performance and efficiency in solving computationally intensive tasks.

PURPOSE & GOALS:

The purpose of the application is to create a reliable and secure communication infrastructure where a client and a server can exchange messages seamlessly. The inclusion of a relay/vpn server adds an extra layer of versatility, allowing for potential use cases such as secure data relay or virtual private network (VPN) functionality, and all of this happens through MPI framework.

1. **MPI Framework:** Develop the whole application through MPI framework and incorporate it into the project to ensure seamless communication.

2. **Client-Server Communication:** Develop a client application that can establish a connection with a server, enabling bidirectional communication.

3. **Relay/VPN Server:** Implement a relay/vpn server that acts as an intermediary between the client and server, facilitating the exchange of messages and potentially enhancing security through VPN-like functionality.

4. **Thread Management:** Efficiently manage threads to handle concurrent communication between the client, relay/vpn server, and server components.

5. **Logging:** Incorporate logging functionality to record communication messages, aiding in debugging and analysis.

Architecture Overview

CLIENT-SERVER ARCHITECTURE:

The MPI (Message Passing Interface) server, client, and relay server application follows a classic client-server architecture, where distinct components play specialized roles in the communication model.

1. **Server Component:** The server component, executing on the node with rank 0, serves as the central hub for communication. It is responsible for establishing a listening socket, accepting client connections, and facilitating the exchange of messages. The server handles incoming messages from clients, processes them, and relays information as needed. It also initiates communication with the relay server for message forwarding.

2. **Client Component:** Clients, executing on nodes with rank 0, connect to the server and engage in a bidirectional communication flow. Clients can send messages to the server, receive responses, and interact with the distributed system. Each client has an independent thread dedicated to receiving server messages asynchronously, ensuring prompt communication.

RELAY/VPN SERVER FUNCTIONALITY:

The relay server acts as an intermediary node that facilitates communication between multiple clients. This server has a critical role in ensuring seamless and secure message relay in a distributed environment.

1. **Connection Management:** The relay server manages connections from both the server and clients. It listens for incoming connections, accepts them, and establishes communication channels with the connected entities.

2. **Message Relay:** Upon receiving a message from a client, the relay server forwards it to the server. Similarly, messages from the server are relayed to the appropriate clients, enabling bidirectional communication. This relay mechanism allows for efficient information exchange between distributed components.

3. **Logging and Tracing:** The relay server incorporates logging functionality to record important events and messages. This logging mechanism aids in monitoring system activity and analyzing communication patterns.

4. **VPN-like Functionality:** In addition to basic message relay, the relay server can be likened to a VPN (Virtual Private Network) server, securely connecting clients,

and enabling them to communicate over potentially insecure networks. It encapsulates and forwards messages, maintaining the confidentiality and integrity of the transmitted data.

5. **Thread Management:** The relay server manages threads efficiently to handle concurrent communication between the client and server components. Separate threads handle communication in both directions, ensuring responsiveness.

Code Organization

The code is organized into three separate C# console applications, each fulfilling a specific role in the communication system: the client, the server, and the relay/vpn server. The modular structure of the code enables clear separation of concerns, making the application more maintainable and extensible.

CLIENT APPLICATION:

The client code is organized to establish and manage a connection with the MPI server. It includes functionality for sending user-input messages to the server and asynchronously receiving responses.

SERVER APPLICATION:

The server code initializes a listening socket, accepts incoming client connections, and facilitates bidirectional communication. It continuously receives messages from clients, relays them to other clients, and allows server-side user input for communication.

RELAY/VPN SERVER APPLICATION:

The relay server acts as an intermediary, managing connections from both clients and the server. It relays messages bidirectionally, enhancing communication security with VPN-like functionality. The code includes logging and error-handling mechanisms for system stability.

MPI Server Implementation

The server code begins by initializing the MPI environment and obtaining the world communicator. It then checks the rank of the current process, with rank 0 designated as the server. The server sets up a listening socket, accepts a client connection, and initiates bidirectional communication. Inside the server's main loop, user input is read from the console, converted to bytes, and sent to the connected client. Simultaneously, a dedicated thread continuously receives messages from the client, printing them to the console. The server operates within an infinite loop, ensuring continuous communication until a termination condition is met.

ROLE IN THE MPI COMMUNICATION MODEL:

In the MPI communication model, the server plays a crucial role as the central coordinator. It serves as the primary point of contact for clients, managing the flow of messages and facilitating communication between connected nodes. The server's responsibilities include accepting client connections, relaying messages, and responding to user input. By acting as a communication hub, the server enables collaborative parallel processing among the connected clients.

USE OF SOCKETS, IP ADDRESSES, AND PORT NUMBERS:

The server employs the Socket class in the System.Net.Sockets namespace to establish a TCP socket for communication. It binds the socket to a specified IP address and port number. This binding ensures that the server is reachable at the designated network address and port, allowing clients to establish connections. The use of IP addresses and port numbers is fundamental for network communication, enabling the server to listen for incoming connections and clients to connect to the server at the specified address and port.

COMMUNICATION PROTOCOL USED FOR MESSAGE EXCHANGE:

The server and client communicate using a simple ASCII-based protocol. Messages are encoded into bytes using the ASCII encoding, allowing for human-readable text transmission. The server sends a welcome message upon client connection, and subsequent messages are relayed bidirectionally between the server and connected clients. The server continually receives messages from the client, processes them, and prints them to the console. The communication protocol relies on the consistency of message formats to ensure proper interpretation by both the server and clients. An exit condition is defined to gracefully terminate the communication loop when a specific exit command is received.

MPI Client Implementation

The client code starts by initializing the MPI environment and obtaining the world communicator. It checks the rank of the current process, with processes other than rank 0 designated as clients. The client then sets up a socket, connects to the MPI server at a specified IP address and port number, and engages in communication. The main loop of the client continuously reads user input from the console, converts it to bytes, and sends it to the server for processing. Simultaneously, a dedicated thread receives messages from the server, prints them to the console, and ensures asynchronous communication.

ROLE IN THE MPI COMMUNICATION MODEL:

In the MPI communication model, the client serves as a distributed node that establishes a connection with the server to participate in parallel processing. Clients send user-input messages to the server and receive responses, allowing for collaborative communication in a distributed environment. Each client operates independently but contributes to the overall parallel processing capabilities of the system.

USE OF SOCKETS, IP ADDRESSES, AND PORT NUMBERS:

The client uses the Socket class from the System.Net.Sockets namespace to establish a TCP socket for communication. It connects to the server using the specified IP address and port number. This connection ensures that the client can communicate with the server at the designated network address and port. The use of sockets, IP addresses, and port numbers is fundamental for network communication, enabling the client to establish a reliable connection with the server and participate in the distributed system.

CLIENT CONNECTION AND COMMUNICATION WITH THE SERVER:

The client initiates a connection with the MPI server by creating a socket and connecting to the server's IP address and port number. Upon successful connection, the client receives a welcome message from the server, indicating the establishment of the communication session. The main loop of the client continually reads user input from the console, allowing users to input messages. These messages are then converted to bytes using ASCII encoding and sent to the server for processing. Simultaneously, the client's dedicated receiving thread continuously listens for incoming messages from the server, decoding and printing them to the console for user interaction.

Relay Server Implementation

The relay server code initializes the MPI environment, obtains the world communicator, and checks the rank of the current process. If the rank is 0, indicating the relay server, the code proceeds to set up two sockets. One socket connects to the MPI server, and the other binds to a specified IP address and port number to manage incoming client connections. Inside the main loop, the relay server continuously receives messages from the server, relays them to connected clients, and vice versa. It also manages the bidirectional communication flow by forwarding messages between clients and the server.

ROLE IN FACILITATING COMMUNICATION BETWEEN CLIENTS:

The relay server acts as a communication hub, serving as a central point for message exchange between clients and the MPI server. Its primary role is to manage connections from both the server and clients, ensuring that messages are properly relayed bidirectionally.

USE OF SOCKETS, IP ADDRESSES, AND PORT NUMBERS:

The relay server utilizes sockets for communication, creating one socket to connect to the MPI server and another to bind to a specific IP address and port number for accepting client connections. This dual-socket setup allows the relay server to act as an intermediary node, facilitating communication between different processes in the MPI system. The use of IP addresses and port numbers is essential for networking, enabling the relay server to establish connections with the MPI server and accept incoming connections from clients.

RELAY SERVER MESSAGING BETWEEN CLIENTS:

Upon receiving a message from the MPI server, the relay server processes the message, prints it to the console, and forwards it to the connected clients. Simultaneously, when a client sends a message, the relay server receives the message, prints it to the console, and relays it to the MPI server. This bidirectional relay mechanism ensures that messages are efficiently exchanged among all connected nodes in the MPI system. In addition to message relay, the relay server incorporates logging functionality to record important events and messages. It also includes error-handling mechanisms to ensure the stability and reliability of the communication system, providing a robust framework for facilitating communication in a distributed environment.

Use of Threads

The application employs multithreading to manage concurrent communication between different components, ensuring responsiveness and efficiency. Here is an overview of how threads are utilized in the client, server, and relay/vpn server applications:

CLIENT APPLICATION & SERVER APPLICATION:

1. **Receive Thread:** Responsible for continuously receiving messages from the server. It uses the Receive method, which is a blocking operation, ensuring responsiveness.

2. **User Input Thread:** Dedicated to reading user input from the console and sending messages to the server.

RELAY/VPN SERVER APPLICATION:

1. **Client Receive Thread:** Continuously listens for messages from the client, processes them, and forwards them to the server.

2. **Server Receive Thread:** Listens for messages from the server, processes them, and relays them to the client.

Networking Protocol (TCP)

The application relies on the Transmission Control Protocol (TCP) as the fundamental networking protocol for facilitating communication between the client, server, and relay/VPN server components. TCP offers a connection-oriented approach, ensuring reliable, ordered, and error-checked data delivery.

CONNECTION ESTABLISHMENT:

TCP provides a reliable connection establishment mechanism, requiring acknowledgment from both the client and server before initiating data transfer. The Socket class in the .NET framework is configured with SocketType.Stream and ProtocolType.Tcp to create TCP sockets for communication.

RELIABLE DATA TRANSFER:

TCP guarantees the reliable and ordered delivery of data between the communicating parties. The Send and Receive methods of the Socket class are employed to send and receive data over the established TCP connection.

FLOW CONTROL:

TCP incorporates flow control mechanisms to prevent overwhelming the receiver with data, dynamically adjusting the rate of data transmission based on the receiver's capacity.

Future Improvements

These are the future improvements that can be done in the project:

1. **Encryption and Security:** Enhance the relay/vpn server to support encryption for secure communication between the client and server. Implementing encryption algorithms, such as SSL/TLS, would bolster the confidentiality of data.

2. **Dynamic IP and Port Configuration:** Allow dynamic configuration of IP addresses and port numbers, providing flexibility for deployment in different network environments without requiring code changes.

3. **User Interface Improvements:** Enhance the client application with a more user-friendly interface, potentially incorporating a graphical user interface (GUI) for a better user experience.

4. **Error Handling Enhancements:** Augment error handling mechanisms to provide more detailed error messages and facilitate easier troubleshooting.

5. **Threading Model Optimization:** Evaluate and potentially optimize the threading model, considering alternatives like asynchronous programming or thread pooling to improve scalability and responsiveness, especially in scenarios with many concurrent connections.

Conclusion

The development of the client-server communication system with a relay/vpn server has been an exploration into creating a versatile and secure communication infrastructure. Key aspects discussed in this report include the architecture overview, code structure, client-server communication, relay/vpn server functionality, thread management, networking protocols, and areas for future improvements.

In conclusion, the implemented MPI (Message Passing Interface) server, client, and relay server application provides a structured framework for parallel communication in a distributed computing environment. By leveraging sockets and a straightforward ASCII-based communication protocol, the system facilitates bidirectional message exchange between clients and the server. The client-server architecture, coupled with the relay server's intermediary role, enables collaborative parallel processing. The organized code structure, use of threads, and consideration of error handling contribute to the application's robustness. This MPI communication model serves as a foundational component for scalable and efficient parallel computing solutions, addressing the increasing demands of modern computational tasks across diverse domains.

Code

SERVER:

```
using System;

using System.Net.Sockets;

using System.Net;

using System.Text;

using MPI;


namespace MPI_Server

{

    class Program

    {

        static void Main(string[] args)

        {

            using (new MPI.Environment(ref args))

            {

                Intracommunicator world = Communicator.world;


                Console.WriteLine("=============================================");

                Console.WriteLine("|\t\tMPI SERVER\t\t|");

                Console.WriteLine("=============================================");


                if (world.Rank == 0)

                {

                    IPEndPoint ipe = new IPEndPoint(IPAddress.Parse("192.168.0.137"), 12000);
                    Socket S = new Socket(ipe.AddressFamily, SocketType.Stream,
ProtocolType.Tcp);

                    S.Bind(ipe);

                    S.Listen(1);

                    Socket C = S.Accept();


                    string welcomeText = "Welcome To MPI Server";

                    byte[] buffer = Encoding.ASCII.GetBytes(welcomeText);

                    C.Send(buffer);


                    Thread receiveThread = new Thread(() =>

                    {

                        while (true)

                        {

                            byte[] buffer1 = new byte[200];

                            C.Receive(buffer1);

                            string msg1 = Encoding.ASCII.GetString(buffer1);

                            Console.WriteLine(msg1);

                        }

                    });

                    receiveThread.Start();


                    int a = 1;

                    while (a != 0)

                    {

                        string msg2 = "MPI Server: " + Console.ReadLine();
```

```
                        byte[] buffer2 = Encoding.ASCII.GetBytes(msg2);

                        C.Send(buffer2);

                    }

                }

            }

        }

    }

}
```

CLIENT:

```
using System;

using System.Net;

using System.Net.Sockets;

using System.Text;

using System.Threading;

using MPI;


namespace MPI_Client

{

    internal class Program

    {

        static void Main(string[] args)

        {

            using (new MPI.Environment(ref args))

            {

                Intracommunicator world = Communicator.world;


                Console.WriteLine("========================================");

                Console.WriteLine("|\t\tMPI CLIENT\t\t|");

                Console.WriteLine("========================================");


                if (world.Rank == 0)

                {

                    IPEndPoint ipe = new IPEndPoint(IPAddress.Parse("192.168.0.137"), 12000);
                    Socket s = new Socket(ipe.AddressFamily, SocketType.Stream,
ProtocolType.Tcp);

                    s.Connect(ipe);


                    byte[] buffer = new byte[200];

                    s.Receive(buffer);

                    string welcomeText = Encoding.ASCII.GetString(buffer);

                    Console.WriteLine(welcomeText);


                    string connectionText = "Connection Established\n";

                    Console.WriteLine(connectionText);

                    byte[] bufferC = Encoding.ASCII.GetBytes(connectionText);

                    s.Send(bufferC);


                    Thread receiveThread = new Thread(() =>

                    {

                        while (true)

                        {

                            byte[] buffer1 = new byte[200];

                            s.Receive(buffer1);

                            string msg1 = Encoding.ASCII.GetString(buffer1);

                            Console.WriteLine(msg1);

                        }

                    });

                    receiveThread.Start();
```

```
                int a = 1;
                while (a != 0)
                {
                    string msg2 = "MPI Client: " + Console.ReadLine();
                    byte[] buffer2 = Encoding.ASCII.GetBytes(msg2);
                    s.Send(buffer2);
                }
            }
        }
    }
}
```

RELAY/VPN SERVER:

```
using System;

using System.IO;

using System.Net;

using System.Net.Sockets;

using System.Text;

using MPI;


namespace RelayServerMPI

{

    internal class Program

    {

        static void Main(string[] args)

        {

            Console.WriteLine("============================================");

            Console.WriteLine("|\t\tRELAY SERVER\t\t|");

            Console.WriteLine("============================================");


            using (new MPI.Environment(ref args))

            {

                Intracommunicator world = Communicator.world;


                if (world.Rank == 0)

                {

                    IPEndPoint ipe = new IPEndPoint(IPAddress.Parse("172.25.240.1"), 12000);
                    Socket s = new Socket(ipe.AddressFamily, SocketType.Stream,
ProtocolType.Tcp);

                    s.Connect(ipe);


                    IPEndPoint ipe2 = new IPEndPoint(IPAddress.Parse("172.25.240.1"), 12000);

                    Socket s2 = new Socket(ipe2.AddressFamily, SocketType.Stream,
ProtocolType.Tcp);

                    s2.Bind(ipe2);

                    s2.Listen(1);

                    Socket c = s2.Accept();


                    string logFileName = "message_log.txt";

                    int a = 1;

                    while (a != 0)

                    {

                        byte[] buffer = new byte[200];

                        s.Receive(buffer);

                        string msg = Encoding.ASCII.GetString(buffer);

                        Console.WriteLine(msg);

                        LogMessage(logFileName, msg);


                        string msg2 = msg;

                        byte[] buffer2 = Encoding.ASCII.GetBytes(msg2);

                        c.Send(buffer2);


                        byte[] buffer3 = new byte[200];
                        c.Receive(buffer3);
                        string msg3 = Encoding.ASCII.GetString(buffer3);
                        Console.WriteLine(msg3);
                        LogMessage(logFileName, msg3);

                        string msg4 = msg3;
                        byte[] buffer4 = Encoding.ASCII.GetBytes(msg4);
                        s.Send(buffer4);
                    }
                }
            }
        }

        static void LogMessage(string fileName, string message)
        {
            string logEntry = $"[{DateTime.Now}] - {message}";

            try
            {
                using (StreamWriter sw = File.AppendText(fileName))
                {
                    sw.WriteLine(logEntry);
                }
            }
            catch (Exception ex)
            {
```

```
                Console.WriteLine($"Error writing to log file: {ex.Message}");
            }
        }
    }
}
```