Prerequisite for Distributed Systems Lab Course

Dr. Satish Chandra, Mr. Soumya Ranjan Jena

Before you implement this lab course, there are several prerequisites you should be familiar with. Here are some of the key concepts and technologies you should understand:

Socket Programming: Understand the basics of socket programming, which involves creating, binding, listening, accepting, and connecting sockets for communication over a network.

TCP/IP Protocol: Learn about the Transmission Control Protocol (TCP) and Internet Protocol (IP), which form the basis of most network communication. TCP provides reliable, connection-oriented communication between devices.

Concurrency: Understand the concept of concurrency, where multiple tasks or threads can execute simultaneously. In the context of a concurrent server, this allows the server to handle multiple client connections at the same time.

Threads or Asynchronous Programming: Familiarize yourself with threads in programming, as they're often used to handle multiple clients concurrently. Alternatively, you can explore asynchronous programming techniques using frameworks like Java's NIO (New I/O) or asynchronous libraries in other languages.

Basic Network Concepts: Have a basic understanding of IP addresses, ports, and how data is routed over networks.

Client-Server Architecture: Understand the client-server architecture and the roles of the client and server in a network application.

Basic Programming Knowledge: Be comfortable with programming concepts in your chosen language (such as C, Java, Python). This includes variables, loops, conditional statements, functions/methods, and data structures.

Error Handling: Learn about handling errors and exceptions in network programming to ensure your application can handle unexpected situations gracefully.

Multi-Threading or Concurrency Libraries: In languages like Java or Python, explore the threading libraries that provide tools for creating and managing threads. Understand thread safety and synchronization mechanisms.

I/O Operations: Understand how input and output operations work in network programming. This includes reading from and writing to sockets, files, and streams.

Testing and Debugging: Learn techniques for testing and debugging network applications. Use tools like Wireshark or network debugging tools provided by your programming environment.

Security Considerations: Consider basic security practices, especially when it comes to handling user input, protecting against malicious clients, and encrypting sensitive data. A concurrent echo

client-server is a network application architecture that allows multiple clients to connect to a server simultaneously and communicate with it concurrently. In this architecture, the server is designed to handle multiple client connections concurrently, and each client is capable of sending messages to the server, which then echoes back those messages to the respective clients.

Lab 1: Implement Concurrent Echo Client-Server Application.

What is concurrent echo client-server?

Server Setup:

The server is running and waiting for incoming connections from clients.

It creates a new thread or task for each incoming client connection, allowing multiple clients to be handled concurrently.

The server listens for incoming messages from each client and echoes those messages back to the same client.

Client Interaction:

Clients connect to the server and establish a socket connection.

Once connected, clients can send messages to the server.

The server receives the message from a client and immediately sends the same message back to the client (echoes the message).

The client receives the echoed message from the server and can continue sending more messages if desired.

The primary advantage of a concurrent echo client-server architecture is that it enables efficient communication between multiple clients and a server without blocking. Each client operates independently, and the server's ability to handle multiple clients concurrently ensures that one client's activity doesn't delay or block the communication of others.

This architecture is often used as a simple example to demonstrate concepts such as socket programming, concurrency, and network communication. It can also be a foundation for building more complex applications where multiple clients need to interact with a central server concurrently.

Program Code: Using Java

Echo Server (Server.java):

```
import java.io.*;

import java.net.*;
```

```
public class Server {
    public static void main(String[] args) {
        int portNumber = 12345;

        try {
            ServerSocket serverSocket = new ServerSocket(portNumber);
            System.out.println("Server listening on port " + portNumber);

            while (true) {
                Socket clientSocket = serverSocket.accept();
                System.out.println("Accepted connection from " + clientSocket.getInetAddress() + ":"
+ clientSocket.getPort());

                ClientHandler clientHandler = new ClientHandler(clientSocket);
                Thread clientThread = new Thread(clientHandler);
                clientThread.start();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

class ClientHandler implements Runnable {
    private Socket clientSocket;

    public ClientHandler(Socket clientSocket) {
        this.clientSocket = clientSocket;
```

```
    }

    @Override
    public void run() {
        try {
            PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true);
            BufferedReader in = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));

            String inputLine;
            while ((inputLine = in.readLine()) != null) {
                System.out.println("Received from " + clientSocket.getInetAddress() + ":" + clientSocket.getPort() + ": " + inputLine);
                out.println(inputLine);
            }

            in.close();
            out.close();
            clientSocket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

This part of the code sets up the main server logic.

Server class contains the main method where the server starts.

The portNumber is set to 12345 where the server will listen for incoming connections.

Inside the try block, a ServerSocket is created on the specified port. This socket is used to listen for incoming connections from clients.

A message is printed to indicate that the server is listening on the given port.

The server enters an infinite loop (while (true)) where it continuously waits for and accepts client connections.

When a client connects (serverSocket.accept()), the client's socket information is printed.

A ClientHandler instance is created for the connected client, and a new thread is started to handle that client's communication.

This class defines the behavior of the server when interacting with each connected client.

It implements the Runnable interface, which allows instances of this class to be executed in separate threads.

The constructor accepts a Socket parameter representing the client's socket connection.

The run method is overridden from the Runnable interface. It contains the code executed by the thread when started.

Inside the run method, it creates input and output streams (PrintWriter and BufferedReader) for communication with the client using the provided socket.

It enters a loop to continuously read lines from the client using in.readLine().

Each line received is printed along with the client's information, and it's sent back to the client using out.println(), effectively echoing the input.

The loop continues until the client disconnects (returns null line).

After the loop, the input and output streams are closed, and the client socket is closed.

Any IOException occurring during communication is caught, and its stack trace is printed.

Echo Client (Client.java):

```
import java.io.*;

import java.net.*;

import java.util.Scanner;


public class Client {

    public static void main(String[] args) {

        String serverAddress = "127.0.0.1";
```

```
        int portNumber = 12345;

        try {
            Socket socket = new Socket(serverAddress, portNumber);
            PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
            BufferedReader in = new BufferedReader(new
InputStreamReader(socket.getInputStream()));

            Scanner scanner = new Scanner(System.in);
            String userInput;

            while (true) {
                System.out.print("Enter a message to send: ");
                userInput = scanner.nextLine();
                if (userInput.equals("exit")) {
                    break;
                }

                out.println(userInput);

                String response = in.readLine();
                System.out.println("Received: " + response);
            }

            in.close();
            out.close();
            socket.close();
            scanner.close();
        } catch (IOException e) {
```

```
            e.printStackTrace();
        }
    }
}
```

The Client class contains the main method where the client program starts.

serverAddress is set to "127.0.0.1", which represents the IP address of the server. This is the loopback address, indicating that the client and server are on the same machine.

portNumber is set to 12345, which is the same port that the server is listening on.

Inside the try block, a Socket connection is created to the server using the specified IP address and port number.

PrintWriter out is used to send data to the server via the socket's output stream.

BufferedReader in is used to read data from the server via the socket's input stream.

A Scanner named scanner is created to read user input from the console.

The program enters a loop using while (true) to continuously send messages to the server and receive responses.

The user is prompted to enter a message, and the input is read using scanner.nextLine().

If the user types "exit", the loop breaks, and the client program terminates.

Otherwise, the user input is sent to the server using out.println(userInput).

The client then reads a response from the server using in.readLine() and prints the received response.

After the loop, the input stream, output stream, socket, and scanner are all closed to release resources.

If an IOException occurs during any of these operations, the exception is caught, and its stack trace is printed.

Output Steps:

To see the output, follow these steps:

1. Compile the EchoServer.java and EchoClient.java files using the javac command.
2. Open multiple terminal/command prompt windows.
3. In one terminal, run the EchoServer class: java EchoServer

4. In each of the other terminals, run the EchoClient class: java EchoClient

Compile and run the server and client using Java:

Compile the server and client:

javac Server.java

javac Client.java

Run the server in one terminal window:

java Server

Run the client in separate terminal windows for multiple clients:
java Client

The server will handle multiple client connections concurrently and echo back the messages received from each client.

You will see output in the terminal windows where you ran the EchoServer and EchoClient classes. The server will indicate when it's listening, and the clients will prompt you to enter messages. The clients will display the echoed messages they receive from the server.

Lab 2. Implement concurrent day-time client-server application.

Explanation:

Server Code:

We create a ServerSocket instance to listen for incoming client connections on a specific port.

Inside the while loop, we wait for clients to connect using serverSocket.accept(). When a client connects, we create a ClientHandler instance to handle its request.

The ClientHandler implements the Runnable interface and overrides the run method. This method is executed when a new thread is started for a client.

Inside the run method of ClientHandler, we get the output stream of the client socket to send data back to the client.

We format the current date and time using SimpleDateFormat and send it to the client.

After sending the response, we close the client socket.

Client Code:

The client code creates a Socket instance to connect to the server using the server's address and port.

We use a BufferedReader to read the response from the server.

The server's response (current date and time) is printed to the console.

Server:

```
import java.io.*;

import java.net.*;

import java.text.SimpleDateFormat;

import java.util.Date;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;


public class ConcurrentDayTimeServer {

    public static void main(String[] args) {

        final int PORT = 12345;
```

```
        ExecutorService executor = Executors.newCachedThreadPool();

        try (ServerSocket serverSocket = new ServerSocket(PORT)) {
            System.out.println("Server is listening on port " + PORT);

            while (true) {
                Socket clientSocket = serverSocket.accept();
                System.out.println("New client connected: " + clientSocket.getInetAddress());

                Runnable task = new ClientHandler(clientSocket);
                executor.submit(task);
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            executor.shutdown();
        }
    }
}

class ClientHandler implements Runnable {
    private Socket clientSocket;

    public ClientHandler(Socket clientSocket) {
        this.clientSocket = clientSocket;
    }

    @Override
```

```
    public void run() {
        try (OutputStream outputStream = clientSocket.getOutputStream();
             PrintWriter out = new PrintWriter(outputStream, true)) {
            SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
            String dateTime = sdf.format(new Date());

            out.println(dateTime);
            System.out.println("Sent date-time to " + clientSocket.getInetAddress());
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            try {
                clientSocket.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Client:

```
import java.io.*;
import java.net.*;

public class DayTimeClient {
    public static void main(String[] args) {
        final String SERVER_IP = "127.0.0.1";  // Change this to the server's IP address
        final int SERVER_PORT = 12345;
```

```
        try (Socket socket = new Socket(SERVER_IP, SERVER_PORT);
             BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()))) {

            String serverResponse = in.readLine();
            System.out.println("Server date and time: " + serverResponse);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Remember to replace "127.0.0.1" with the actual IP address of the server if you're running the client on a different machine.

Compile and run the server and client code separately. The server will listen for incoming connections and respond with the current date and time to each client. The use of the ExecutorService ensures that multiple clients can be handled concurrently.