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.

Algorithm:
In a loop, accept incoming client connections using accept() method.
Spawn a new thread or use asynchronous methods to handle each client connection separately.

Time Retrieval:
Get the current date and time using language-specific libraries or system calls.
Format the date and time into a human-readable string.

Sending Data:
Send the formatted date and time string to the connected client using the client's socket.

Client Disconnection:
Close the socket when the client is done receiving the data or when an error occurs.

Client Side:
Connection Setup:
Create a socket and connect it to the server's IP address and port.

Data Reception:
Receive data from the server using the socket's recv() method.
Print or process the received date and time information.

Closing Connection:
Close the socket when done receiving data or when an error occurs.

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.

Lab 3: Configure following options on server socket and tests them: SO_KEEPALIVE, SO_LINGER, SO_SNDBUF, SO_RCVBUF, TCP_NODELAY.

Explanation:

SO_KEEPALIVE:

SO_KEEPALIVE is a socket option that allows the operating system to automatically send keep-alive packets on an idle TCP connection to check if the other end is still responsive.

In the code, we set SO_KEEPALIVE to 1 (enabled) using setsockopt.

This option helps detect and close inactive connections more reliably.

```
import java.io.IOException;
import java.net.ServerSocket;
```

```
import java.net.Socket;

public class ServerSocketExample {
    public static void main(String[] args) {
        try {
            ServerSocket serverSocket = new ServerSocket(8080);

            // Enable SO_KEEPALIVE
            serverSocket.setKeepAlive(true);

            System.out.println("Server is running. Waiting for a client to connect...");

            Socket clientSocket = serverSocket.accept();

            // Test SO_KEEPALIVE
            System.out.println("SO_KEEPALIVE enabled: " + clientSocket.getKeepAlive());

            // Close the sockets
            clientSocket.close();
            serverSocket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

SO_LINGER:

SO_LINGER is a socket option that controls what happens when you try to close a socket that still has data to be sent or received.

In the code, we set SO_LINGER to (1, 30), which means that when we close the socket, it will linger for 30 seconds to send any remaining data.

After the timeout, the socket will be forcefully closed.

```
import java.io.IOException;

import java.net.ServerSocket;

import java.net.Socket;


public class ServerSocketExample {

    public static void main(String[] args) {

        try {

            ServerSocket serverSocket = new ServerSocket(8080);


            // Enable SO_LINGER with a timeout of 30 seconds

            serverSocket.setSoLinger(true, 30);


            System.out.println("Server is running. Waiting for a client to connect...");


            Socket clientSocket = serverSocket.accept();


            // Test SO_LINGER

            System.out.println("SO_LINGER enabled: " + clientSocket.getSoLinger());


            // Close the sockets

            clientSocket.close();

            serverSocket.close();
```

```
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

SO_SNDBUF and SO_RCVBUF:

SO_SNDBUF and SO_RCVBUF control the send and receive buffer sizes, respectively.

In the code, we set both to a buffer size of 8192 bytes using setsockopt.

These options allow you to tune the socket's buffer sizes, which can impact the performance of data transmission.

```
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;

public class ServerSocketExample {
    public static void main(String[] args) {
        try {
            ServerSocket serverSocket = new ServerSocket(8080);

            // Set SO_SNDBUF and SO_RCVBUF to custom values (e.g., 8192 bytes)
            serverSocket.setSendBufferSize(8192);
            serverSocket.setReceiveBufferSize(8192);

            System.out.println("Server is running. Waiting for a client to connect...");

            Socket clientSocket = serverSocket.accept();
```

```
            // Test SO_SNDBUF and SO_RCVBUF
            System.out.println("SO_SNDBUF: " + clientSocket.getSendBufferSize());
            System.out.println("SO_RCVBUF: " + clientSocket.getReceiveBufferSize());

            // Close the sockets
            clientSocket.close();
            serverSocket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

TCP_NODELAY:

TCP_NODELAY is a socket option that controls the Nagle's algorithm, which can introduce delays in small data packet transmission.

In the code, we set TCP_NODELAY to 1 (enabled) using setsockopt, disabling the algorithm.

This option can be useful when you want to reduce latency in applications that send small packets frequently.

```
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.SocketException;

public class ServerSocketExample {
    public static void main(String[] args) {
        try {
            ServerSocket serverSocket = new ServerSocket(8080);
```

```
            // Disable TCP_NODELAY
            serverSocket.setTcpNoDelay(false);

            System.out.println("Server is running. Waiting for a client to connect...");

            Socket clientSocket = serverSocket.accept();

            // Test TCP_NODELAY
            System.out.println("TCP_NODELAY enabled: " + !clientSocket.getTcpNoDelay());

            // Close the sockets
            clientSocket.close();
            serverSocket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

In the main() function, we create a server socket, bind it to an address and port, and start listening for incoming connections. When a client connects, it demonstrates the use of these options. For example, SO_KEEPALIVE will help ensure the connection is still alive, SO_LINGER controls how the socket behaves when closing, and SO_SNDBUF and SO_RCVBUF set the buffer sizes for data transmission. Additionally, TCP_NODELAY affects the transmission of small packets.

Testing these options would involve connecting to the server and observing their effects in real-world scenarios. You may need a client program to fully test these options as it requires interaction with the server. You can monitor network traffic and behavior to ensure that these socket options are functioning as expected in your application. Configuring and testing the socket options (SO_KEEPALIVE, SO_LINGER, SO_SNDBUF, SO_RCVBUF, TCP_NODELAY) on a server socket involves several steps.

Algorithm:

Initialize the Server Socket:

Create a server socket using the appropriate socket library in your programming language (e.g., Python's socket library).

Configure SO_KEEPALIVE:

Use the setsockopt method to enable SO_KEEPALIVE on the server socket. Set it to 1 (enabled).

Configure SO_LINGER:

Use the setsockopt method to configure SO_LINGER on the server socket. Set it with a tuple containing (1, timeout), where timeout is the desired linger timeout in seconds.

Configure SO_SNDBUF and SO_RCVBUF:

Use setsockopt to configure SO_SNDBUF and SO_RCVBUF on the server socket. Set the desired buffer sizes using appropriate values (e.g., 8192 bytes).

Configure TCP_NODELAY:

Use setsockopt to enable TCP_NODELAY on the server socket. Set it to 1 (enabled).

Bind and Listen:

Bind the server socket to a specific IP address and port.

Start listening for incoming connections using the listen method.

Accept Client Connections:

In a loop, accept incoming client connections using the accept method.

For each accepted connection, you can print a message to indicate that a connection has been established.

Test SO_KEEPALIVE:

Leave a client connection idle for a while (longer than the SO_KEEPALIVE interval).

Observe if the server socket sends keep-alive packets to the client to maintain the connection.

Test SO_LINGER:

Establish a client connection.

Simulate a scenario where the server closes the socket while data is still in the send buffer.

Observe if the SO_LINGER option delays the socket closure and ensures data transmission.

Test SO_SNDBUF and SO_RCVBUF:

Measure the data transmission performance by sending/receiving data between the server and client.

Adjust the buffer sizes and observe the impact on data transmission speed and efficiency.

Test TCP_NODELAY:

Measure the latency and performance of sending small packets over the connection with TCP_NODELAY enabled.

Compare this to the performance with TCP_NODELAY disabled to see the difference in packet transmission behavior.

Cleanup and Close:

Properly close the client sockets after testing.

Close the server socket when the server is finished testing or interrupted (e.g., by a keyboard interrupt).

Analyzing Results:

Examine the results of your tests and monitor network traffic, if necessary, to ensure that the socket options are behaving as expected.

Java Code:

```
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.SocketException;

public class ServerSocketOptionsTest {
    public static void main(String[] args) {
        int port = 8080;

        try {
            // Create a server socket
            ServerSocket serverSocket = new ServerSocket(port);

            // Set SO_KEEPALIVE option (1 enables, 0 disables)
            serverSocket.setKeepAlive(true);

            // Set SO_LINGER option (Linger on close with a timeout of 30 seconds)
            serverSocket.setSoLinger(true, 30);

            // Set SO_SNDBUF option (send buffer size)
            serverSocket.setSendBufferSize(8192);

            // Set SO_RCVBUF option (receive buffer size)
            serverSocket.setReceiveBufferSize(8192);
```

```
            // Set TCP_NODELAY option (disable Nagle's algorithm, true enables, false disables)
            serverSocket.setTcpNoDelay(true);

            System.out.println("Server is listening on port " + port);

            while (true) {
                // Accept incoming connections
                Socket clientSocket = serverSocket.accept();
                System.out.println("Accepted connection from " + clientSocket.getInetAddress());

                // You can perform any necessary server-side operations here

                // Close the client socket
                clientSocket.close();
            }
        } catch (SocketException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Output:

Server socket created on port 8080

Client connected: /127.0.0.1

Send Buffer Size: 8192

Receive Buffer Size: 8192

Client connected: /192.168.1.100

Send Buffer Size: 8192

Receive Buffer Size: 8192

Lab 4: Incrementing a counter in shared memory.

Explanation:

Incrementing a counter in shared memory refers to the process of increasing the value of a counter variable that is stored in a region of memory that is accessible to multiple threads or processes concurrently. Shared memory is a form of inter-process communication (IPC) that allows different processes or threads to access and manipulate the same memory region, making it a useful mechanism for synchronization and coordination in concurrent programming.

Here's a general overview of how incrementing a counter in shared memory might work:

Initialization: Initially, you would create or allocate a block of shared memory that can hold the counter variable. This shared memory block is typically created by one process or thread and then shared among multiple processes or threads.

Access Control: To ensure that multiple processes or threads don't try to access and modify the shared counter simultaneously (which could lead to race conditions and data corruption), synchronization mechanisms like mutexes, semaphores, or atomic operations are often used. These mechanisms provide a way to control access to the shared memory region.

Increment Operation: When a process or thread wants to increment the counter, it first acquires the necessary synchronization mechanism to gain exclusive access to the shared memory. Once it has exclusive access, it can safely read the current value of the counter, increment it, and write the updated value back to the shared memory location.

Release Lock: After incrementing the counter and updating the shared memory, the process or thread releases the synchronization mechanism, allowing other processes or threads to access the counter if they need to.

Program Code:

In Java, you can use the java.util.concurrent package to implement a counter increment operation in shared memory. The package provides thread-safe constructs for synchronization, including AtomicInteger for atomic operations on integers and Lock for more fine-grained control.

```
import java.util.concurrent.atomic.AtomicInteger;

public class SharedMemoryCounter {
    public static void main(String[] args) {
        // Create an AtomicInteger with an initial value of 0
        AtomicInteger counter = new AtomicInteger(0);

        // Create multiple threads to increment the counter
        int numThreads = 4;
        Thread[] threads = new Thread[numThreads];

        for (int i = 0; i < numThreads; i++) {
            threads[i] = new Thread(() -> {
                for (int j = 0; j < 10000; j++) {
                    // Increment the counter atomically
                    counter.incrementAndGet();
                }
            });
            threads[i].start();
        }

        // Wait for all threads to finish
        try {
            for (Thread thread : threads) {
                thread.join();
```

```
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // Print the final value of the counter
        System.out.println("Final Counter Value: " + counter.get());
    }
}
```

In the above program we create an AtomicInteger called counter and use the incrementAndGet() method to increment it atomically from multiple threads. The example creates four threads, each incrementing the counter 10,000 times. Finally, we wait for all threads to finish and print the final value of the counter. The use of AtomicInteger ensures that the counter is incremented safely in a multi-threaded environment without the need for explicit synchronization mechanisms like locks.

Lab 5: Create CORBA based server-client application

What is CORBA?

CORBA stands for Common Object Request Broker Architecture, is a middleware technology and a set of specifications that enable communication between distributed objects in a networked system. It allows objects written in different programming languages to interoperate seamlessly, making it a powerful solution for building distributed, language-agnostic applications. CORBA provides a standard way for objects to make requests and share services across a network, regardless of the platforms and languages used to implement those objects.

Procedure and Java Code:

To create a simple CORBA-based server-client application using Java, you will need to follow these steps:

1. Define the IDL (Interface Definition Language): In CORBA, the IDL is used to define the interfaces of objects that will be accessible remotely. You'll need to create an IDL file that describes the methods and attributes of the remote object.

Here's a simple example of an IDL file:

```
module HelloWorld {
  interface HelloWorld {
    string sayHello();
  };
};
```

2. Generate Java Stubs and Skeletons: Use the IDL compiler to generate Java stubs and skeletons from your IDL file. The stubs are used by the client to make remote calls, and the skeletons are used on the server side to receive and process those calls.

You can use the idlj tool to generate the Java code:

```
idlj -fall HelloWorld.idl
```

This will generate a package named HelloWorld with the necessary Java files.

3. Implement the Server: Write a server implementation that provides the functionality specified in your IDL. This class should extend the generated skeleton and implement the methods defined in the IDL interface.

```
import HelloWorld._HelloWorldImplBase;

public class HelloWorldImpl extends _HelloWorldImplBase {
  public String sayHello() {
    return "Hello, World!";
  }
}
```

4. Create and Initialize the ORB: The Object Request Broker (ORB) is responsible for handling remote method invocations. Initialize the ORB in your server and client applications.

Server:

```
import org.omg.CORBA.*;
import HelloWorld.HelloWorldHelper;

public class Server {
  public static void main(String[] args) {
    try {
      ORB orb = ORB.init(args, null);
      HelloWorldImpl helloWorld = new HelloWorldImpl();

      // Register the HelloWorld object with the ORB
      orb.connect(helloWorld);

      NamingContextExt ncRef = NamingContextExtHelper.narrow(orb.resolve_initial_references("NameService"));
      NameComponent path[] = ncRef.to_name("HelloWorld");
      ncRef.rebind(path, helloWorld);

      System.out.println("Server is ready and waiting for client requests...");

      orb.run();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

5. Create the Client: Write a client application to connect to the server and make remote method calls.

```
import org.omg.CORBA.*;
import HelloWorld.HelloWorld;
```

```
import HelloWorld.HelloWorldHelper;

public class Client {
  public static void main(String[] args) {
    try {
      ORB orb = ORB.init(args, null);
      org.omg.CORBA.Object objRef = orb.resolve_initial_references("NameService");
      NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);

      NameComponent path[] = ncRef.to_name("HelloWorld");
      HelloWorld helloWorld = HelloWorldHelper.narrow(ncRef.resolve(path));

      String message = helloWorld.sayHello();
      System.out.println("Message from server: " + message);
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Compile and Run: Compile both the server and client applications and then run them.

You'll need to run the server first and keep it running. Then, you can run the client, which will connect to the server and invoke the sayHello method.

This is a basic example of a CORBA-based server-client application in Java. CORBA can be more complex in real-world applications, but this should give you a starting point to understand how to create and use CORBA components.

Output:

Server Output:

Server is ready and waiting for client requests...

This message indicates that the server is up and running, waiting for client requests.

Client Output:

Message from server: Hello, World!

The client will connect to the server and invoke the sayHello method, and the server will respond with the "Hello, World!" message, which the client will print to the console. Please note that you should have both the server and client applications running concurrently. The server should be running first to listen for incoming requests, and the client should be run afterward to make a request to the server.