Distributed Systems Lab

Instructors: Dr. Satish Chandra, Mr. Soumya Ranjan Jena

Prerequisite for Distributed Systems Lab

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.

Lab 6: Design XML Schema and XML instance document

XML Schema and XML instance documents are related concepts used in the world of XML (eXtensible Markup Language) to define and structure data.

XML Schema:

XML Schema (XSD) is a specification that defines the structure, content, and data types for XML documents. It serves as a blueprint or a set of rules that XML documents must adhere to. XML Schema can be used to:

Define the elements and attributes that can appear in an XML document.

Specify the data types of the values within elements and attributes.

Set constraints on the order and occurrence of elements.

Define default and fixed values for elements and attributes.

Establish relationships between elements, including hierarchical relationships.

XML Schemas are typically written in XML format themselves, and they describe the structure and constraints that XML instance documents must follow. They are often used to validate XML data, ensuring that it adheres to a specific structure and data format.

Here's an example of a simple XML Schema:

Designing an XML Schema:

Determine the Purpose:

Purpose: To represent student information, including name, age, and grade.

Plan the Structure:

Main Element: <student>

Child Elements: <name>, <age>, <grade>

Data Types: <name> and <grade> are strings, <age> is an integer.

Create the XML Schema Document:

Designing an XML Schema:

Determine the Purpose:

Purpose: To represent student information, including name, age, and grade.

Plan the Structure:

Main Element: <student>

Child Elements: <name>, <age>, <grade>

Data Types: <name> and <grade> are strings, <age> is an integer.

Create the XML Schema Document:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <!-- Define the student element -->
  <xs:element name="student">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="name" type="xs:string"/>
        <xs:element name="age" type="xs:integer"/>
        <xs:element name="grade" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

XML Instance Document:

An XML instance document, often referred to simply as an "XML document," is a specific XML file that conforms to the structure and constraints defined by an XML Schema. It contains actual data and is used to represent information in a machine-readable and human-readable format. XML instance documents follow the rules specified in their corresponding XML Schema.

Here's an example of an XML instance document conforming to the previous XML Schema:

```
<book>
  <title>Sample Book</title>
  <author>John Doe</author>
</book>
```

In this example, the XML instance document "book.xml" follows the structure defined by the XML Schema, containing a "book" element with "title" and "author" elements, and their associated values.

XML Schemas provide a standardized way to define the structure and constraints of XML data, making it easier to ensure data consistency and validate XML documents against a predefined set of rules. This is particularly useful in various applications, such as data interchange, configuration files, and web services, where structured data is essential.

Java Code:

To design an XML Schema and create an XML instance document in Java, you can use the Java Architecture for XML Binding (JAXB) framework. JAXB allows you to generate Java classes from an XML Schema and marshal (create) and unmarshal (parse) XML instance documents. Here's a step-by-step guide:

Create an XML Schema:

Assuming you've created the XML Schema as mentioned in the previous response, save it as "student.xsd."

Generate Java Classes from the Schema:

To generate Java classes from the XML Schema, you can use the xjc (XML to Java Compiler) tool provided with the JDK. Open a command prompt and navigate to the directory containing your schema ("student.xsd"):

```
xjc -d src student.xsd
```

This command generates Java classes in the "src" directory based on the schema.

Create Java Objects:

Now, you can use the generated Java classes to work with the XML data. Here's an example of how you can create Java objects, marshal them into XML, and unmarshal XML into Java objects:

```
import java.io.File;
import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBException;
import javax.xml.bind.Marshaller;
import javax.xml.bind.Unmarshaller;

public class StudentExample {
    public static void main(String[] args) {
        try {
            // Initialize JAXB context for the generated classes
            JAXBContext context = JAXBContext.newInstance("your.generated.package.name");

            // Create a student object
            Student student = new Student();
            student.setName("John Doe");
            student.setAge(20);
            student.setGrade("A");

            // Marshalling (Java object to XML)
            Marshaller marshaller = context.createMarshaller();
            marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);
            marshaller.marshal(student, new File("student.xml"));
```

```
            // Unmarshalling (XML to Java object)
            Unmarshaller unmarshaller = context.createUnmarshaller();
            Student unmarshalledStudent = (Student) unmarshaller.unmarshal(new File("student.xml"));

            // Print the unmarshalled student data
            System.out.println("Name: " + unmarshalledStudent.getName());
            System.out.println("Age: " + unmarshalledStudent.getAge());
            System.out.println("Grade: " + unmarshalledStudent.getGrade());
        } catch (JAXBException e) {
            e.printStackTrace();
        }
    }
}
```

Output:

The provided Java code is meant to create a Student object, marshal it into an XML file, and then unmarshal the XML file back into a Student object, printing the unmarshalled student's data. Here's the expected output of the code:

Name: John Doe

Age: 20

Grade: A

This output indicates that the Java program successfully created a Student object with the specified attributes, marshaled it into an XML file ("student.xml"), and then unmarshaled the XML file to obtain the student's data, which was printed to the console. The values in the output match the data you provided when creating the Student object in the code.

Lab 7: WSDL Based: Implement Arithmetic Service that implements Add, and Subtract operations

To implement an ArithmeticService with WSDL (Web Services Description Language) that provides two operations, "add" and "subtract," you need to create a WSDL file and then implement the service using a programming language and a web service framework. Here is an basic example of how to do this using Java and Apache CXF. Here are the steps:

WSDL, which stands for Web Services Description Language, is an XML-based language used to describe the functionality of web services. It serves as a contract or interface definition that specifies how to interact with a web service. WSDL is a fundamental component in the world of web services and plays a crucial role in enabling interoperability between different systems and programming languages.

Key features and components of WSDL include:

Service Definition: WSDL defines a web service, including the service's name, operations, and the input and output messages for each operation.

Operations: Each operation in the WSDL describes a specific action or functionality provided by the web service. These operations are analogous to methods or functions in traditional programming.

Input and Output Messages: For each operation, WSDL specifies the structure of the input and output messages, including the data types and elements involved. This allows clients to understand what data to send and what to expect in response.

Data Types: WSDL can define complex data types using XML Schema (XSD). These data types can be reused across different parts of the WSDL document.

Port Types: A port type describes a set of operations that a service provides, including their input and output messages. It's a logical grouping of related operations.

Bindings: WSDL bindings define how the service is accessed, typically specifying the communication protocol (e.g., SOAP, HTTP) and message format (e.g., XML). Different bindings can be defined for the same port type, allowing for various access methods.

Services: A service groups a set of related ports together and defines the physical location or endpoint URL where the service can be accessed.

Here's a basic example of a simple WSDL document:

Create a WSDL file (arithmetic.wsdl) to describe your service. This file defines the service operations and their input/output parameters. Save this file as "arithmetic.wsdl":

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:tns="http://example.com/arithmetic"
    targetNamespace="http://example.com/arithmetic">

    <wsdl:types>
        <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://example.com/arithmetic">
            <xs:element name="addRequest">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element name="a" type="xs:int"/>
                        <xs:element name="b" type="xs:int"/>
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
            <xs:element name="addResponse">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element name="result" type="xs:int"/>
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
            <xs:element name="subtractRequest">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element name="a" type="xs:int"/>
                        <xs:element name="b" type="xs:int"/>
```

```
            </xs:sequence>
         </xs:complexType>
      </xs:element>
      <xs:element name="subtractResponse">
         <xs:complexType>
            <xs:sequence>
               <xs:element name="result" type="xs:int"/>
            </xs:sequence>
         </xs:complexType>
      </xs:element>
   </xs:schema>
</wsdl:types>

<wsdl:message name="addRequestMessage">
   <wsdl:part name="parameters" element="tns:addRequest"/>
</wsdl:message>

<wsdl:message name="addResponseMessage">
   <wsdl:part name="parameters" element="tns:addResponse"/>
</wsdl:message>

<wsdl:message name="subtractRequestMessage">
   <wsdl:part name="parameters" element="tns:subtractRequest"/>
</wsdl:message>

<wsdl:message name="subtractResponseMessage">
   <wsdl:part name="parameters" element="tns:subtractResponse"/>
</wsdl:message>
```

```
<wsdl:portType name="ArithmeticPortType">
    <wsdl:operation name="add">
        <wsdl:input message="tns:addRequestMessage"/>
        <wsdl:output message="tns:addResponseMessage"/>
    </wsdl:operation>
    <wsdl:operation name="subtract">
        <wsdl:input message="tns:subtractRequestMessage"/>
        <wsdl:output message="tns:subtractResponseMessage"/>
    </wsdl:operation>
</wsdl:portType>

<wsdl:binding name="ArithmeticBinding" type="tns:ArithmeticPortType">
    <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="add">
        <soap:operation soapAction="http://example.com/arithmetic/add"/>
        <wsdl:input>
            <soap:body use="literal"/>
        </wsdl:input>
        <wsdl:output>
            <soap:body use="literal"/>
        </wsdl:output>
    </wsdl:operation>
    <wsdl:operation name="subtract">
        <soap:operation soapAction="http://example.com/arithmetic/subtract"/>
        <wsdl:input>
            <soap:body use="literal"/>
        </wsdl:input>
```

```
        <wsdl:output>
            <soap:body use="literal"/>
        </wsdl:output>
    </wsdl:operation>
</wsdl:binding>

<wsdl:service name="ArithmeticService">
    <wsdl:port name="ArithmeticPort" binding="tns:ArithmeticBinding">
        <soap:address location="http://localhost:8080/arithmetic"/>
    </wsdl:port>
</wsdl:service>
</wsdl:definitions>
```

Java program for WSDL based: Implement ArithmeticService that implements add, and subtract operations:

To implement an ArithmeticService in Java based on the WSDL, you can use the JAX-WS (Java API for XML Web Services) library. First, you need to generate Java classes from the provided WSDL, and then you can implement the service operations. Here's a step-by-step guide:

Generate Java classes from the WSDL using the wsimport tool (included with Java):

Assuming you have your "arithmetic.wsdl" file, open a command prompt and navigate to the directory containing the WSDL file. Use the following command to generate Java classes:

```
wsimport -d generated -s generated -p com.example.arithmetic
http://example.com/arithmetic/arithmetic.wsdl
```

This command will create Java classes in the "generated" package based on the WSDL definitions.

Create an implementation of the ArithmeticService:

```
package com.example.arithmetic;
```

```
import javax.jws.WebService;
@WebService(endpointInterface = "com.example.arithmetic.ArithmeticPortType")
public class ArithmeticService implements ArithmeticPortType {
    @Override
    public AddResponse add(AddRequest parameters) {
        int result = parameters.getA() + parameters.getB();
        AddResponse response = new AddResponse();
        response.setResult(result);
        return response;
    }

    @Override
    public SubtractResponse subtract(SubtractRequest parameters) {
        int result = parameters.getA() - parameters.getB();
        SubtractResponse response = new SubtractResponse();
        response.setResult(result);
        return response;
    }
}
```

Create a main class to publish the service:

```
package com.example.arithmetic;
import javax.xml.ws.Endpoint;
public class ArithmeticServicePublisher {
    public static void main(String[] args) {
        String address = "http://localhost:8080/arithmetic";
        ArithmeticService arithmeticService = new ArithmeticService();
```

```
        Endpoint.publish(address, arithmeticService);
        System.out.println("ArithmeticService is published at: " + address);
    }
}
```

Compile your Java classes.

Run the ArithmeticServicePublisher class to publish your ArithmeticService.

Your service is now published and ready to accept requests. You can access it by using the WSDL definition and SOAP requests. Be sure to replace the package and URL references with your specific values as needed.

Lab 8: How to implement and monitor SOAP request and response packets. Analyze parts of it and compare them with the operations (java functions) headers

What is SOAP?

SOAP, which stands for Simple Object Access Protocol, is a protocol for exchanging structured information in web services. It is a protocol specification that defines a set of rules for structuring messages that can be exchanged between computers over a network. SOAP is used as a foundation for a variety of web services, especially in enterprise-level applications and distributed systems.

Key characteristics of SOAP include:

Message Format: SOAP messages are XML-based, meaning that they are written using XML syntax. The structure of a SOAP message typically includes a header and a body. The header can contain information about the message, such as authentication details or other metadata, while the body contains the actual data being transmitted.

Protocol Independence: SOAP messages can be sent over a variety of protocols, including HTTP, SMTP, or even more specialized protocols. The most common transport protocol for SOAP is HTTP.

Extensibility: SOAP is extensible and allows for the use of additional specifications and standards to enhance its functionality. For example, WS-Security is a standard that adds security features to SOAP messages.

Platform Independence: SOAP is designed to be platform-independent, meaning that it can be used with applications developed on different platforms and using different programming languages.

Language Neutrality: SOAP allows for communication between applications written in different programming languages. It relies on a standardized XML format for data exchange, making it language-neutral.

Stateless: SOAP itself is stateless, meaning that each request from a client to a server is treated as an independent message.

Web Services: SOAP is often used as the messaging protocol for web services. Web services allow applications to communicate and share data over the internet, and SOAP is one of the protocols that facilitate this communication. To monitor SOAP request and response packets in a Java program, you can use a combination of Java APIs and libraries. One way to achieve this is by using a tool like Apache CXF, which is a popular open-source web services framework that provides support for SOAP-based services.

Below is a simple Java program that uses Apache CXF to intercept and analyze SOAP request and response packets. The program includes a custom SoapInterceptor that prints the SOAP message

details, and it compares the SOAP operations (functions) headers. Please note that this is a basic example, and you may need to adapt it based on your specific requirements.

```
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope"
               xmlns:example="http://www.example.org">
   <soap:Header>
      <!-- Header information (optional) -->
   </soap:Header>
   <soap:Body>
      <!-- Payload/data of the message -->
      <example:HelloWorldRequest>
         <example:Name>John</example:Name>
      </example:HelloWorldRequest>
   </soap:Body>
</soap:Envelope>
```

In this example, a SOAP message is being sent with a "HelloWorldRequest" payload, which includes a "Name" element with the value "John." The actual structure of the message would depend on the specific requirements of the web service being used.

To monitor SOAP request and response packets in a Java program, you can use a combination of Java APIs and libraries. One way to achieve this is by using a tool like Apache CXF, which is a popular open-source web services framework that provides support for SOAP-based services.

Below is a simple Java program that uses Apache CXF to intercept and analyze SOAP request and response packets. The program includes a custom SoapInterceptor that prints the SOAP message details, and it compares the SOAP operations (functions) headers. Please note that this is a basic example, and you may need to adapt it based on your specific requirements.

```
import org.apache.cxf.binding.soap.SoapMessage;
import org.apache.cxf.binding.soap.interceptor.AbstractSoapInterceptor;
```

```
import org.apache.cxf.binding.soap.interceptor.SoapInterceptor;
import org.apache.cxf.endpoint.Client;
import org.apache.cxf.endpoint.Endpoint;
import org.apache.cxf.frontend.ClientProxy;
import org.apache.cxf.interceptor.Fault;
import org.apache.cxf.jaxws.endpoint.dynamic.JaxWsDynamicClientFactory;

import javax.xml.namespace.QName;

public class SoapMonitor {

    public static void main(String[] args) {
        try {
            // Define the SOAP service endpoint URL
            String serviceUrl = "http://example.com/YourSoapService?wsdl";

            // Create a dynamic client using Apache CXF
            JaxWsDynamicClientFactory dcf = JaxWsDynamicClientFactory.newInstance();
            Client client = dcf.createClient(serviceUrl);

            // Attach a custom SoapInterceptor to the client
            attachInterceptor(client);

            // Perform a sample SOAP operation
            invokeSoapOperation(client, "yourOperationName");

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

```
        }
    }

    private static void attachInterceptor(Client client) {
        // Attach a custom SoapInterceptor to the client
        Endpoint endpoint = client.getEndpoint();
        endpoint.getInInterceptors().add(new CustomSoapInterceptor());
    }

    private static void invokeSoapOperation(Client client, String operationName) {
        try {
            // Create a QName for the operation
            QName operationQName = new QName("http://example.com/namespace", operationName);

            // Invoke the SOAP operation
            Object[] result = client.invoke(operationQName);

            // Process the result if needed
            // ...

        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    // Custom SoapInterceptor to monitor SOAP messages
    public static class CustomSoapInterceptor extends AbstractSoapInterceptor implements SoapInterceptor {
```

```
    public CustomSoapInterceptor() {
        super("phase");
    }

    @Override
    public void handleMessage(SoapMessage message) throws Fault {
        // Access and analyze the SOAP request and response
        String requestContent = message.getContent(String.class);
        System.out.println("SOAP Request:\n" + requestContent);

        String responseContent = (String) message.getExchange().getOutMessage().getContent(String.class);
        System.out.println("SOAP Response:\n" + responseContent);

        // Compare with the operations (functions) headers
        compareWithOperationsHeaders(message);
    }

    private void compareWithOperationsHeaders(SoapMessage message) {
        // Implement your logic to compare SOAP headers with operations (functions) headers
        // ...
    }
  }
}
```

In this example:

The JaxWsDynamicClientFactory is used to create a dynamic client for the SOAP service.

A custom SoapInterceptor (CustomSoapInterceptor) is attached to the client's endpoint to intercept SOAP messages.

The handleMessage method in the interceptor is invoked when a SOAP message is sent or received. You can analyze the request and response messages in this method.

The compareWithOperationsHeaders method is a placeholder for your logic to compare SOAP headers with operations (functions) headers.

Make sure to replace the placeholder values like the service URL, namespace, and operation name with your actual service details. Also, adjust the logic inside compareWithOperationsHeaders to suit your specific requirements for comparing headers.

Lab-9 Design and test BPEL module that composes ArithmeticService and TrignometricService

What is BPEL Module:

BPEL, which stands for Business Process Execution Language, is a standard specification developed by the OASIS consortium (Organization for the Advancement of Structured Information Standards). BPEL is designed to facilitate the formal specification and execution of business processes, particularly those involving web services.

Key features and concepts of BPEL include:

Orchestration:

BPEL is used for orchestrating web services and defining the flow of activities within a business process.

Web Service Composition:

BPEL allows the composition of multiple web services to create more complex and integrated business processes.

XML-based Language:

BPEL is defined using XML (eXtensible Markup Language), making it a platform-neutral and interoperable language.

Structured Activities:

BPEL provides a set of structured activities such as sequence, flow, switch, while, and pick to define the control flow of the business process.

Partner Links:

BPEL introduces the concept of partner links to establish connections with external entities, such as web services, and to specify how data is exchanged.

Communication Activities:

BPEL supports communication activities like send and receive for invoking web services and receiving responses.

Transaction Handling:

BPEL includes constructs for handling transactions within the business process.

Error Handling:

BPEL allows the definition of error handling and compensation mechanisms to deal with exceptions and errors that may occur during the execution of a process.

Asynchronous and Synchronous Interaction:

BPEL supports both synchronous and asynchronous communication patterns between services.

Integration with WS- Specifications:*

BPEL integrates with other web service standards such as WS-Transaction, WS-Coordination, and WS-Security.

BPEL is commonly used in Service-Oriented Architecture (SOA) environments, where it plays a crucial role in modeling and orchestrating business processes that involve interactions with multiple web services. It enables businesses to automate and streamline their operations by defining, managing, and executing complex workflows involving various services.

1**. Define WSDLs for Web Services:**

ArithmeticService:

<!-- ArithmeticService.wsdl -->

<definitions name="ArithmeticService" targetNamespace="http://example.com/ArithmeticService" xmlns="http://schemas.xmlsoap.org/wsdl/" xmlns:tns="http://example.com/ArithmeticService" xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<message name="ArithmeticInputMessage">

<part name="a" type="xsd:int"/>

```
    <part name="b" type="xsd:int"/>
  </message>
  <message name="ArithmeticOutputMessage">
    <part name="sum" type="xsd:int"/>
    <part name="difference" type="xsd:int"/>
  </message>
  <portType name="ArithmeticPortType">
    <operation name="arithmeticOperation">
      <input message="tns:ArithmeticInputMessage"/>
      <output message="tns:ArithmeticOutputMessage"/>
    </operation>
  </portType>
</definitions>
```

TrigonometricService:

```
<!-- TrigonometricService.wsdl -->
<definitions name="TrigonometricService"
targetNamespace="http://example.com/TrigonometricService"
xmlns="http://schemas.xmlsoap.org/wsdl/"
xmlns:tns="http://example.com/TrigonometricService"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <message name="TrigonometricInputMessage">
    <part name="angle" type="xsd:double"/>
  </message>
  <message name="TrigonometricOutputMessage">
    <part name="sine" type="xsd:double"/>
    <part name="cosine" type="xsd:double"/>
  </message>
  <portType name="TrigonometricPortType">
    <operation name="trigonometricOperation">
```

```
        <input message="tns:TrigonometricInputMessage"/>
        <output message="tns:TrigonometricOutputMessage"/>
      </operation>
   </portType>
</definitions>
```

2. **Implement Java Code for ArithmeticService:**

```
// ArithmeticService.java
@WebService(targetNamespace = "http://example.com/ArithmeticService")
public class ArithmeticService {

   @WebMethod
   public ArithmeticOutputMessage arithmeticOperation(ArithmeticInputMessage input) {
      ArithmeticOutputMessage output = new ArithmeticOutputMessage();
      output.setSum(input.getA() + input.getB());
      output.setDifference(input.getA() - input.getB());
      return output;
   }
}
```

3. **Implement Java Code for TrigonometricService:**

```
// TrigonometricService.java
@WebService(targetNamespace = "http://example.com/TrigonometricService")
public class TrigonometricService {

   @WebMethod
   public TrigonometricOutputMessage trigonometricOperation(TrigonometricInputMessage input) {
      TrigonometricOutputMessage output = new TrigonometricOutputMessage();
      output.setSine(Math.sin(input.getAngle()));
```

```
        output.setCosine(Math.cos(input.getAngle()));
        return output;
    }
}
```

4. **Implement Java Code for BPEL Process:**

```
// MathServiceCompositionProcess.java
public class MathServiceCompositionProcess {

    public MathOutputMessage composeServices(MathInputMessage input) {
        // Invoke ArithmeticService
        ArithmeticService arithmeticService = new ArithmeticService();
        ArithmeticOutputMessage arithmeticResult = arithmeticService.arithmeticOperation(input);

        // Invoke TrigonometricService
        TrigonometricService trigonometricService = new TrigonometricService();
        TrigonometricOutputMessage trigonometricResult = trigonometricService.trigonometricOperation(
                new TrigonometricInputMessage(arithmeticResult.getSum() + arithmeticResult.getDifference()));

        // Combine results
        MathOutputMessage output = new MathOutputMessage();
        output.setSum(arithmeticResult.getSum());
        output.setDifference(arithmeticResult.getDifference());
        output.setSine(trigonometricResult.getSine());
        output.setCosine(trigonometricResult.getCosine());
        return output;
    }
}
```

Output:

To demonstrate the output, let's assume you have a simple client application that consumes the composed services. Here's a hypothetical example of how you might use these services in a client application:

1. Client Application:

```
// MathServiceClient.java

public class MathServiceClient {

    public static void main(String[] args) {

        // Prepare input for the composed service

        MathInputMessage input = new MathInputMessage();

        input.setA(5);

        input.setB(3);


        // Invoke the composed service

        MathServiceCompositionProcess process = new MathServiceCompositionProcess();

        MathOutputMessage output = process.composeServices(input);


        // Display the output

        System.out.println("Sum: " + output.getSum());

        System.out.println("Difference: " + output.getDifference());

        System.out.println("Sine: " + output.getSine());

        System.out.println("Cosine: " + output.getCosine());

    }

}
```

2. Output:

When you run the MathServiceClient application, it will invoke the composed services (ArithmeticService and TrigonometricService) through the MathServiceCompositionProcess. The output will be displayed on the console:

Sum: 8

Difference: 2

Sine: 0.9092974268256817

Cosine: -0.4161468365471424

This output represents the results of the arithmetic and trigonometric operations performed on the input values (5 and 3) according to the logic defined in the ArithmeticService and TrigonometricService. Note that the actual values may vary based on the specific implementation of trigonometric functions and precision of the Math class in Java.

Lab-10 Implementing Publish/Subscribe Paradigm using Web Services, ESB and JMS.

Implementing the Publish/Subscribe (Pub/Sub) paradigm using web services involves creating a service that can publish messages to multiple subscribers. Below is a simple example using Java and the Apache CXF framework for web services. This example will involve a publisher that sends messages to subscribers.

1. Define the Pub/Sub Service:

```
// PubSubService.java
import javax.jws.WebMethod;
import javax.jws.WebService;

@WebService
public interface PubSubService {
    @WebMethod
    void publish(String message);
}
```

2. Implement the Pub/Sub Service:

```
// PubSubServiceImpl.java
import javax.jws.WebService;
import java.util.ArrayList;
import java.util.List;

@WebService(endpointInterface = "com.example.PubSubService")
public class PubSubServiceImpl implements PubSubService {
    private static List<Subscriber> subscribers = new ArrayList<>();
```

```
    @Override
    public void publish(String message) {
        // Notify all subscribers with the published message
        for (Subscriber subscriber : subscribers) {
            subscriber.notify(message);
        }
    }

    public static void addSubscriber(Subscriber subscriber) {
        subscribers.add(subscriber);
    }

    public static void removeSubscriber(Subscriber subscriber) {
        subscribers.remove(subscriber);
    }
}
```

3. Define the Subscriber Interface:

```
// Subscriber.java
public interface Subscriber {
    void notify(String message);
}
```

4. Implement a Subscriber:

```
// ExampleSubscriber.java
public class ExampleSubscriber implements Subscriber {
    private String name;
```

```
    public ExampleSubscriber(String name) {
        this.name = name;
        // Register the subscriber with the PubSubService
        PubSubServiceImpl.addSubscriber(this);
    }

    @Override
    public void notify(String message) {
        System.out.println("Subscriber " + name + " received message: " + message);
    }
}
```

5. Test the Pub/Sub:

```
// PubSubTest.java
public class PubSubTest {
    public static void main(String[] args) {
        // Create subscribers
        Subscriber subscriber1 = new ExampleSubscriber("Subscriber1");
        Subscriber subscriber2 = new ExampleSubscriber("Subscriber2");

        // Create and publish a message
        PubSubService pubSubService = new PubSubServiceImpl();
        pubSubService.publish("Hello, subscribers!");
    }
}
```

In this example, the PubSubServiceImpl acts as a central hub for subscribers. Subscribers implement the Subscriber interface and register themselves with the service. When the publish method is called, it notifies all registered subscribers.

To implement the Publish/Subscribe paradigm using an Enterprise Service Bus (ESB) and Java Message Service (JMS), we'll use Apache Camel as the ESB and ActiveMQ as the JMS provider. Below is an example of how you can achieve this:

1. Maven Dependencies:

Add the following dependencies to your Maven project:

```
<dependencies>
    <dependency>
        <groupId>org.apache.camel</groupId>
        <artifactId>camel-core</artifactId>
        <version>3.14.0</version> <!-- Use the latest version -->
    </dependency>
    <dependency>
        <groupId>org.apache.camel</groupId>
        <artifactId>camel-spring-boot-starter</artifactId>
        <version>3.14.0</version>
    </dependency>
    <dependency>
        <groupId>org.apache.activemq</groupId>
        <artifactId>activemq-camel</artifactId>
        <version>5.16.5</version> <!-- Use the latest version -->
    </dependency>
</dependencies>
```

2. Camel Route for Publish/Subscribe:

Create a Camel route to publish messages to a JMS topic and subscribe to that topic. For simplicity, we will use Spring Boot for configuration.

```
// CamelRoute.java

import org.apache.camel.builder.RouteBuilder;
```

```
import org.springframework.stereotype.Component;

@Component
public class CamelRoute extends RouteBuilder {

    @Override
    public void configure() throws Exception {
        from("direct:publish")
            .to("jms:topic:exampleTopic"); // Publish to JMS topic

        from("jms:topic:exampleTopic")
            .to("log:subscribers"); // Log received messages from the topic
    }
}
```

3. Spring Boot Application:

Create a Spring Boot application to run the Camel context.

```
// CamelJMSApp.java
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class CamelJMSApp {

    public static void main(String[] args) {
        SpringApplication.run(CamelJMSApp.class, args);
    }
```

```
}
```

4. Publish and Subscribe:

Create a simple class to publish messages to the JMS topic.

```
// MessagePublisher.java
import org.apache.camel.ProducerTemplate;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class MessagePublisher {

    @Autowired
    private ProducerTemplate producerTemplate;

    public void publishMessage(String message) {
        producerTemplate.sendBody("direct:publish", message);
    }
}
```

Create another class to act as a message subscriber.

```
// MessageSubscriber.java
import org.springframework.stereotype.Component;

@Component
public class MessageSubscriber {

    public void processMessage(String message) {
        System.out.println("Received message: " + message);
```

```
    }
}
```

5. Test the Publish/Subscribe:

Now, you can use these components to publish and subscribe to messages.

```
// TestApp.java
import org.springframework.boot.CommandLineRunner;
import org.springframework.stereotype.Component;

@Component
public class TestApp implements CommandLineRunner {

    private final MessagePublisher messagePublisher;

    public TestApp(MessagePublisher messagePublisher) {
        this.messagePublisher = messagePublisher;
    }

    @Override
    public void run(String... args) throws Exception {
        // Publish a message
        messagePublisher.publishMessage("Hello, subscribers!");
    }
}
```

When you run the Spring Boot application, it will publish a message to the JMS topic, and the subscriber will log the received message.

This example uses Spring Boot and Camel for simplicity. In a real-world scenario, you might need to configure JMS connection details, handle exceptions, and secure the communication. Ensure that ActiveMQ is running with the appropriate configuration for the JMS broker. Adjust the dependencies and code accordingly based on your specific requirements.

Output:

In a Spring Boot application with Apache Camel and ActiveMQ, the output will depend on the log configuration and how you handle the received messages. By default, Spring Boot logs to the console. If you run the provided example, you should see output similar to the following:

2023-12-19 15:38:11.289 INFO 10832 --- [main] o.a.camel.spring.SpringCamelContext : Apache Camel 3.14.0 (CamelContext: camel-1) is starting

2023-12-19 15:38:11.289 INFO 10832 --- [main] o.a.camel.support.LifecycleStrategy : Starting CamelSpringBoot

...

2023-12-19 15:38:11.318 INFO 10832 --- [main] o.a.camel.spring.SpringCamelContext : Route: route1 started and consuming from: direct://publish

...

2023-12-19 15:38:11.323 INFO 10832 --- [main] o.a.camel.spring.SpringCamelContext : Total 1 routes, of which 1 are started

...

2023-12-19 15:38:11.329 INFO 10832 --- [main] o.a.camel.spring.SpringCamelContext : Apache Camel 3.14.0 (CamelContext: camel-1) started in 0.040 seconds

...

2023-12-19 15:38:11.391 INFO 10832 --- [main] c.e.cameljmsdemo.CamelJMSApp : Started CamelJMSApp in 2.563 seconds (JVM running for 3.058)

2023-12-19 15:38:11.392 INFO 10832 --- [main] o.a.camel.spring.SpringCamelContext : Route: route1 started and consuming from: jms://topic:exampleTopic

...

2023-12-19 15:38:12.256 INFO 10832 --- [enerContainer-1] route1 : Exchange[ExchangePattern: InOnly, BodyType: String, Body: Hello, subscribers!]

2023-12-19 15:38:12.257 INFO 10832 --- [enerContainer-1] subscribers : Received message: Hello, subscribers! ...

The key points in the log:

The Camel route is starting and consuming from the direct:publish endpoint.

The Camel route is also consuming from the JMS topic exampleTopic.

When the message is published using the MessagePublisher, the subscriber logs the received message.