Thank you for sharing the images. I'll go through each problem statement and provide an explanation of the experiments based on the text provided.

### Sheet 1

1. **Study Basics of OpenMP API (Open Multi-Processor API):**

   - This experiment involves learning the OpenMP API, a tool used for parallel programming in shared-memory architectures. You'll study its syntax and basic functions to manage threads and parallel tasks, which can enhance the performance of applications that run on multi-core processors.

2. **Develop a Distributed Application Using MPI (Message Passing Interface):**

   - Here, you will create an application using MPI, which is used for distributed computing where each process runs independently and communicates with others via message-passing. This experiment will help you understand how MPI facilitates communication between nodes in a distributed system.

3. **Develop a Distributed Application Using Java Sockets and RMI:**

   - In this experiment, you'll implement a client-server model using Java Sockets (for network communication) and RMI (Remote Method Invocation). This involves writing programs that allow a client to communicate with a server over the network, possibly handling requests and responses between them.

4. **Develop a Distributed Application Using a Publish-Subscribe Messaging System:**

   - This task involves creating a messaging-based distributed application using the Publish-Subscribe paradigm. Here, applications (publishers) send messages to a central broker, and other applications (subscribers) receive these messages if they subscribe to the topic. This experiment is useful in real-time messaging and notification systems.

5. **Create a Web Service and Consume it in a Distributed Application:**

- This experiment requires you to build a simple web service (e.g., a REST API) and create another application that consumes this service. This process helps in understanding how to expose services over a network and make them accessible to other applications.

6. **Develop a Distributed Application Using CORBA and Java IDL:**

   - CORBA (Common Object Request Broker Architecture) allows communication between applications written in different languages. This experiment involves creating a CORBA application with Java IDL (Interface Definition Language) to understand how distributed systems can interact across diverse platforms.

7. **Design an RMI Application for Concatenation of Strings:**

   - This experiment requires you to design an RMI-based distributed application where a client sends two strings to a server, and the server returns the concatenation of these strings. This will help you understand basic remote computation and data exchange using RMI.

8. **Design an RMI Application to Check if One String is a Substring of Another:**

   - In this experiment, you'll create an RMI-based application where a client sends two strings, and the server checks if the first string is a substring of the second. This teaches you about basic remote processing and decision-making in a distributed environment.

### Sheet 2

9. **Design an RMI Application to Check if One String is the Reverse of Another:**

   - Here, you will create an RMI-based application where a client sends two strings to a server, and the server verifies if the first string is the reverse of the second. This experiment focuses on text processing and verification over remote communication.

10. **Design an RMI Application to Check for Equal Vowel Count in Two Strings:**

- This task involves an RMI application where the server checks if two input strings contain the same number of vowels. It introduces you to remote string manipulation and analysis tasks.

11. **Design an RMI Application to Compute \(a^b \):**

- In this experiment, a client sends two integers, \(a \) and \(b \), to a server, and the server calculates \(a^b \) (a raised to the power of b). This experiment showcases how RMI can handle mathematical computations remotely.

12. **Design an MPI Application to Check for Prime Number:**

- Here, using MPI, a client submits an integer to the server, which checks if it's a prime number. This task illustrates how MPI can be used for remote computation and decision-making based on mathematical properties.

13. **Design an MPI Application to Convert Feet to Meters:**

- This experiment involves designing an MPI application where a client sends a value in feet, and the server converts it to meters. It shows how MPI can handle simple data transformations.

14. **Design an MPI Application to Convert Degrees to Radians:**

- Similar to the previous experiment, this task involves creating an MPI-based application where a client submits a value in degrees, and the server converts it to radians. It helps in understanding unit conversion in distributed applications.

15. **Design an MPI Application to Calculate Factorial:**

- In this final experiment, the client sends an integer to the server, which then calculates the factorial of that number and returns the result. This task demonstrates MPI's utility in handling recursive or iterative calculations remotely.

Each of these experiments gives you hands-on experience with either RMI or MPI, two essential tools in distributed computing. RMI focuses on remote method invocation in Java, ideal for small-scale distributed systems, while MPI is a widely-used standard for message-passing in large-scale, high-performance distributed computing environments.

Sure, let’s go through each experiment in more detail, covering both the setup and execution steps. This includes creating and running programs for both RMI (Remote Method Invocation) and MPI (Message Passing Interface) applications.

### Experiment 1: **Study Basics of OpenMP API**

1. **Objective**: Understand OpenMP, a library for parallel programming on shared-memory architectures, to write efficient multi-threaded code.

2. **Steps**:
   - Install OpenMP (usually comes with C/C++ compilers like GCC).
   - Write a simple C program that uses OpenMP directives to parallelize a for-loop.
   - Compile and execute with an OpenMP-enabled compiler.

3. **Example Code** (in C):
   ```c
   #include <stdio.h>
   #include <omp.h>

   int main() {
       #pragma omp parallel
   ```

      {
        int id = omp_get_thread_num();
        printf("Hello from thread %d\n", id);
      }
      return 0;
    }
   ~~~
   - **Execution**: Compile with `gcc -fopenmp program.c -o program`, then run with `./program`.

---

### Experiment 2: **Develop a Distributed Application Using MPI**

1. **Objective**: Create a simple MPI application to understand message-passing between processes.

2. **Steps**:
   - Install MPI (e.g., OpenMPI).
   - Write a program that sends and receives messages between processes.
   - Compile and run with an MPI-enabled compiler.

3. **Example Code** (in C):
   ```c
   #include <mpi.h>
   #include <stdio.h>
   ~~~

```
  int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    if (world_rank == 0) {
      int number = 100;
      MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    } else if (world_rank == 1) {
      int received_number;
      MPI_Recv(&received_number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      printf("Received number: %d\n", received_number);
    }
    MPI_Finalize();
    return 0;
  }
```

  - **Execution**: Compile with `mpicc program.c -o program` and run with `mpirun -np 2 ./program`.

### Experiment 3: **Distributed Application with Java Sockets and RMI**

1. **Objective**: Use Java sockets for client-server communication and Java RMI for remote method invocation.

2. **Steps (Java Sockets)**:

  - Write a server program to accept connections and send responses.

  - Write a client program to connect to the server and receive responses.

  Server Code (Sockets):

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

public class Server {
    public static void main(String[] args) throws IOException {
        ServerSocket serverSocket = new ServerSocket(12345);
        Socket clientSocket = serverSocket.accept();
        PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true);
        out.println("Hello from Server!");
        serverSocket.close();
    }
}
```

  Client Code (Sockets):

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

```
public class Client {
    public static void main(String[] args) throws IOException {
        Socket socket = new Socket("localhost", 12345);
        BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
        System.out.println("Message from server: " + in.readLine());
        socket.close();
    }
}
```

- **Execution**: Run the server, then the client. The client should print "Hello from Server!"

3. **Steps (Java RMI)**:
  - Write a remote interface and implementation, a server to host it, and a client to access it.

  Remote Interface:

```java
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Hello extends Remote {
    String sayHello() throws RemoteException;
}
```

  Server Implementation:

```java
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;

public class Server implements Hello {
    public String sayHello() {
        return "Hello, RMI!";
    }
    public static void main(String[] args) {
        try {
            Server obj = new Server();
            Hello stub = (Hello) UnicastRemoteObject.exportObject(obj, 0);
            Registry registry = LocateRegistry.getRegistry();
            registry.bind("Hello", stub);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Client Code:

```java
import java.rmi.registry.LocateRegistry;
```

```
   import java.rmi.registry.Registry;

   public class Client {
       public static void main(String[] args) {
           try {
               Registry registry = LocateRegistry.getRegistry();
               Hello stub = (Hello) registry.lookup("Hello");
               System.out.println("Response: " + stub.sayHello());
           } catch (Exception e) {
               e.printStackTrace();
           }
       }
   }
```
   - **Execution**: Start the RMI registry, run the server, then run the client.

### Experiment 4: **Develop a Publish-Subscribe Messaging Application**

1. **Objective**: Use a publish-subscribe model, e.g., with a message broker like Kafka or MQTT.

2. **Steps**:
   - Set up a Kafka server or MQTT broker.
   - Write producer (publisher) and consumer (subscriber) applications in Java or Python.

Producer Code (Kafka, Java):

```java
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");
Producer<String, String> producer = new KafkaProducer<>(props);
producer.send(new ProducerRecord<>("my-topic", "Hello, World!"));
producer.close();
```

Consumer Code (Kafka, Java):

```java
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("group.id", "test-group");
props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
props.put("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
Consumer<String, String> consumer = new KafkaConsumer<>(props);
consumer.subscribe(Arrays.asList("my-topic"));
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(100));
    for (ConsumerRecord<String, String> record : records)
        System.out.println("Received: " + record.value());
```

  }

  ```

3. **Execution**: Start the Kafka server, run the producer, then run the consumer to receive messages.

---

### Additional RMI/MPI Experiments (Concatenation, Reverse String Check, etc.)

For experiments like concatenation, substring check, reverse check, and other RMI/MPI tasks:

1. **RMI Approach**:

  - Set up an RMI interface for each operation.

  - Implement the logic on the server, and invoke it from the client.

  Example: For concatenation, write an RMI method `String concatenate(String s1, String s2)` and call it from the client.

2. **MPI Approach**:

  - Use message-passing with MPI to send inputs (strings, numbers) to worker processes and receive the results.

  Example: For prime checking, send an integer to a worker process, which checks and returns if it's prime.

---
  ```

Each experiment involves setting up the environment, writing the appropriate code, and running the client and server processes (for RMI/MPI) or producers and consumers (for publish-subscribe) to observe distributed behavior. Let me know if you need further clarification on any specific experiment.

Certainly! Let's go through the remaining experiments in detail.

### Experiment 5: **Create a Simple Web Service and Write a Distributed Application to Consume the Web Service**

1. **Objective**: Set up a simple RESTful web service and write a client application to consume it.

2. **Steps**:

  - Use a framework like Spring Boot (Java) or Flask (Python) to create a REST API.

  - Write a client application to send requests and receive responses from the web service.

3. **Server Code** (using Flask, Python):

  ```python
  from flask import Flask, jsonify, request

  app = Flask(__name__)

  @app.route('/greet', methods=['GET'])
  def greet():
      name = request.args.get('name', 'World')
      return jsonify({'message': f'Hello, {name}!'})
  ```

   if __name__ == '__main__':
       app.run(port=5000)

   - **Execution**: Start the server with `python server.py`. This service listens on `http://localhost:5000/greet`.

4. **Client Code** (Python):
   ```python
   import requests

   response = requests.get("http://localhost:5000/greet", params={"name": "Alice"})
   print("Response:", response.json())
   ```

   - **Execution**: Run the client, which makes a GET request to the server, receiving a greeting message in response.

### Experiment 6: **Develop a Distributed Application with CORBA using Java IDL**

1. **Objective**: Use CORBA (Common Object Request Broker Architecture) for remote method calls in a distributed application.

2. **Steps**:
   - Define an IDL (Interface Definition Language) file with the method signatures.
   - Generate Java stubs and skeletons using `idlj` (IDL to Java compiler).

   - Implement the server and client programs.

3. **Example IDL File** (`Hello.idl`):

   ```idl
   module HelloApp {
       interface Hello {
           string sayHello();
       };
   };
   ```

4. **Steps**:

   - Compile IDL with `idlj -fall Hello.idl`.
   - Implement the generated Java interface.

5. **Server Code**:

   ```java
   import HelloApp.*;
   import org.omg.CORBA.*;
   import org.omg.PortableServer.*;

   public class HelloServer extends HelloPOA {
       public String sayHello() {
           return "Hello from CORBA!";
       }
   ```

    public static void main(String[] args) {

        // Initialize and register CORBA server

    }

  }

6. **Client Code**:

  ```java

  import HelloApp.*;

  import org.omg.CORBA.*;

  public class HelloClient {

    public static void main(String[] args) {

        // Initialize CORBA client and call sayHello()

    }

  }

  ```

  - **Execution**: Start the ORB, run the server, then the client.

### Experiment 7-15: **Distributed Applications Using RMI and MPI**

These experiments focus on distributed computations using RMI and MPI. Here's a breakdown for each specific problem.

### Experiment 7: **RMI for Concatenation of Two Strings**

1. **Objective**: Design an RMI application where the client sends two strings, and the server concatenates them.

2. **Steps**:

   - Define an RMI interface with a `concatenate` method.

   - Implement the server to perform the concatenation.

   - Implement the client to send the strings and receive the result.

3. **Server Code**:

   ```java
   public class ConcatenateServer implements ConcatenateInterface {

       public String concatenate(String s1, String s2) throws RemoteException {

           return s1 + s2;

       }

   }
   ```

4. **Client Code**:

   ```java
   String result = stub.concatenate("Hello", " World");

   System.out.println("Concatenated Result: " + result);
   ```

### Experiment 8: **RMI to Check if One String is a Substring of Another**

1. **Objective**: Design an RMI application where the client submits two strings, and the server verifies if one string is a substring of the other.

2. **Server Code**:

   ```java
   public boolean isSubstring(String s1, String s2) throws RemoteException {
       return s2.contains(s1);
   }
   ```

3. **Client Code**:

   ```java
   boolean result = stub.isSubstring("Hello", "Hello World");
   System.out.println("Is Substring: " + result);
   ```

### Experiment 9: **RMI to Check if One String is the Reverse of Another**

1. **Objective**: The server checks if one string is the reverse of the other.

2. **Server Code**:

   ```java
   public boolean isReverse(String s1, String s2) throws RemoteException {
       return new StringBuilder(s1).reverse().toString().equals(s2);
   }
   ```

3. **Client Code**:

   ```java
   boolean result = stub.isReverse("abc", "cba");
   System.out.println("Is Reverse: " + result);
   ```

### Experiment 10: **RMI to Check Equal Vowel Count in Two Strings**

1. **Objective**: Check if both strings have the same number of vowels.

2. **Server Code**:

   ```java
   public boolean hasEqualVowels(String s1, String s2) throws RemoteException {
       int count1 = s1.replaceAll("[^aeiouAEIOU]", "").length();
       int count2 = s2.replaceAll("[^aeiouAEIOU]", "").length();
   ```

      return count1 == count2;

  }

3. **Client Code**:

  ```java

  boolean result = stub.hasEqualVowels("hello", "world");

  System.out.println("Equal Vowel Count: " + result);

  ```

### Experiment 11: **RMI to Calculate a<sup>b</sup>**

1. **Objective**: The server calculates a raised to the power of b.

2. **Server Code**:

  ```java

  public double power(int a, int b) throws RemoteException {

      return Math.pow(a, b);

  }

  ```

3. **Client Code**:

  ```java
  ```

   double result = stub.power(2, 3);

   System.out.println("Power Result: " + result);

   ```

---

### Experiment 12-15: **MPI for Various Computations**

#### Example Template (For All MPI Problems)

For each of these MPI problems (prime check, unit conversion, factorial, etc.), follow this structure.

1. **Steps**:

   - Initialize MPI with multiple processes.

   - Use rank 0 as the client process to send data to worker processes.

   - Worker processes perform calculations and send results back to rank 0.

#### Experiment 12: **MPI Prime Check**

- The server (worker process) receives an integer, checks if it’s prime, and sends the result.

#### Experiment 13: **Convert Feet to Meters**

- The client process sends a length in feet, which is converted to meters by the server.
   ```

#### Experiment 14: **Convert Degrees to Radians**

- The client sends an angle in degrees, converted to radians by the server.

#### Experiment 15: **Factorial Calculation**

- The client sends an integer, and the server returns its factorial.

### Example MPI Code for Prime Check (Experiment 12)

```c
#include <mpi.h>
#include <stdio.h>
#include <stdbool.h>

bool is_prime(int n) {
    if (n <= 1) return false;
    for (int i = 2; i < n; i++) {
        if (n % i == 0) return false;
    }
    return true;
}
```

```
int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);

    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == 0) {
        int number = 17;
        MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
        bool result;
        MPI_Recv(&result, 1, MPI_C_BOOL, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Is Prime: %d\n", result);
    } else if (rank == 1) {
        int received_number;
        MPI_Recv(&received_number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        bool result = is_prime(received_number);
        MPI_Send(&result, 1, MPI_C_BOOL, 0, 0, MPI_COMM_WORLD);
    }

    MPI_Finalize();
    return 0;
}
```

- **Execution**: Compile with `mpicc program.c -o program` and run with `mpirun -np 2 ./program`.

These steps provide detailed instructions for each experiment. Let me know if you need further help with any particular experiment or code!

Alright, let's go through the next set of distributed application experiments that focus on **Remote Method Invocation (RMI)** and **Message Passing Interface (MPI)** for various types of computations. I'll break down each experiment in terms of objectives, the server and client code structure, and the execution steps.

### Experiment 13: **Distributed Application using MPI for Unit Conversion - Feet to Meters**

1. **Objective**: Create an MPI-based application where the client submits a distance in feet to the server, which converts it to meters and returns the result.

2. **Steps**:

   - Initialize MPI processes with a client-server structure.

   - The client sends a distance in feet to the server.

   - The server receives the distance, performs the conversion (`meters = feet * 0.3048`), and sends the result back to the client.

3. **MPI Code**:

   ```c

   #include <mpi.h>
   ```

```
#include <stdio.h>


int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);


    if (rank == 0) {
        // Client process
        float feet = 10.0;
        MPI_Send(&feet, 1, MPI_FLOAT, 1, 0, MPI_COMM_WORLD);
        float meters;
        MPI_Recv(&meters, 1, MPI_FLOAT, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Feet: %.2f, Meters: %.2f\n", feet, meters);
    } else if (rank == 1) {
        // Server process
        float feet;
        MPI_Recv(&feet, 1, MPI_FLOAT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        float meters = feet * 0.3048;
        MPI_Send(&meters, 1, MPI_FLOAT, 0, 0, MPI_COMM_WORLD);
    }


    MPI_Finalize();
    return 0;
}
```

~~~

4. **Execution**:

  - Compile with `mpicc program.c -o program`.

  - Run with `mpirun -np 2 ./program`.

Experiment 14: **Distributed Application using MPI for Unit Conversion - Degrees to Radians**

1. **Objective**: Create an MPI-based application where the client submits an angle in degrees to the server, which converts it to radians and returns the result.

2. **Steps**:

  - The client sends an angle in degrees.

  - The server performs the conversion (`radians = degrees * (π / 180)`) and returns the result.

3. **MPI Code**:

  ```c
  #include <mpi.h>

  #include <stdio.h>

  #define PI 3.14159265358979323846

  int main(int argc, char** argv) {

      MPI_Init(&argc, &argv);
~~~

```
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == 0) {
      // Client process
      float degrees = 45.0;
      MPI_Send(&degrees, 1, MPI_FLOAT, 1, 0, MPI_COMM_WORLD);
      float radians;
      MPI_Recv(&radians, 1, MPI_FLOAT, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      printf("Degrees: %.2f, Radians: %.2f\n", degrees, radians);
    } else if (rank == 1) {
      // Server process
      float degrees;
      MPI_Recv(&degrees, 1, MPI_FLOAT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      float radians = degrees * (PI / 180.0);
      MPI_Send(&radians, 1, MPI_FLOAT, 0, 0, MPI_COMM_WORLD);
    }

    MPI_Finalize();
    return 0;
  }
```

4. **Execution**:
   - Compile with `mpicc program.c -o program`.

  - Run with `mpirun -np 2 ./program`.

### Experiment 15: **Distributed Application using MPI for Factorial Calculation**

1. **Objective**: The client sends an integer to the server, and the server computes the factorial of that integer.

2. **Steps**:

  - The client sends an integer value.

  - The server calculates the factorial of the received integer and sends it back to the client.

3. **MPI Code**:

  ```c
  #include <mpi.h>
  #include <stdio.h>

  int factorial(int n) {
      if (n <= 1) return 1;
      return n * factorial(n - 1);
  }

  int main(int argc, char** argv) {
      MPI_Init(&argc, &argv);
  ```

```
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == 0) {
        // Client process
        int number = 5;
        MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
        int result;
        MPI_Recv(&result, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Factorial of %d is %d\n", number, result);
    } else if (rank == 1) {
        // Server process
        int number;
        MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        int result = factorial(number);
        MPI_Send(&result, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }

    MPI_Finalize();
    return 0;
}
```

4. **Execution**:
   - Compile with `mpicc program.c -o program`.

  - Run with `mpirun -np 2 ./program`.

These MPI experiments demonstrate basic distributed computations using message passing. Each experiment follows a similar structure of sending data, performing computation on the server, and returning the result to the client. Let me know if you need any further clarification or additional code examples!

Here’s a detailed explanation of the concepts involved in each experiment to help you prepare for your oral and viva.

### 1. **OpenMP API (Open Multi-Processing API)**

  - **Purpose**: OpenMP is a parallel programming model that allows developers to write code that runs on multiple processors, using shared memory.

  - **Key Concepts**:

    - **Parallel Regions**: Code blocks that can run in parallel.

    - **Threads**: Units of execution; multiple threads can perform tasks simultaneously.

    - **Directives**: Pragma-based commands (`#pragma omp parallel`) used to specify parallel execution.

  - **Applications**: Suitable for tasks like matrix multiplication, where operations can be divided across multiple processors.

### 2. **Message Passing Interface (MPI)**

  - **Purpose**: MPI is a standardized and portable message-passing system to program parallel computers, enabling processes to communicate with each other.

  - **Key Concepts**:

- **Process Rank**: Unique identifier for each process in the MPI world.

- **MPI_Send and MPI_Recv**: Functions used for sending and receiving messages between processes.

- **Communicators**: Groups of processes that can communicate with each other (e.g., `MPI_COMM_WORLD`).

- **Applications**: Ideal for distributed applications where tasks are divided among different processes, such as factorization and unit conversions in experiments.

### 3. **Java Sockets and RMI (Remote Method Invocation)**

- **Sockets**:

  - **Purpose**: A low-level communication mechanism that allows a client and server to communicate over a network.

  - **Key Concepts**:

    - **Client-Server Model**: The client initiates the connection, and the server responds.

    - **Input/Output Streams**: Streams for reading and writing data across the network.

- **RMI**:

  - **Purpose**: Enables remote execution of methods in a distributed application, allowing clients to call methods on objects residing on a server.

  - **Key Concepts**:

    - **Stub and Skeleton**: Stub on the client side represents the remote object; skeleton on the server side manages method calls.

    - **Registry**: A naming service where clients can locate remote objects.

### 4. **Publish-Subscribe Messaging System**

- **Purpose**: A pattern where publishers send messages to topics, and subscribers receive messages based on subscriptions.

- **Key Concepts**:

  - **Topics**: Named channels through which messages are routed.

   - **Subscribers**: Clients that receive messages sent to specific topics.

  - **Applications**: Often used in real-time notification systems and chat applications.

### 5. **Web Services**

  - **Purpose**: Web services allow communication over the internet, usually using HTTP, allowing different applications to interact with each other.

  - **Key Concepts**:

   - **SOAP and REST**: Protocols used for building web services (SOAP uses XML; REST is more flexible and can use JSON).

   - **WSDL**: XML-based language for describing web services.

  - **Applications**: Common in systems where different applications need to communicate, like web-based APIs.

### 6. **CORBA (Common Object Request Broker Architecture)**

  - **Purpose**: CORBA enables communication between programs written in different languages, across different machines, by standardizing how objects are accessed.

  - **Key Concepts**:

   - **IDL (Interface Definition Language)**: Used to define the interface for CORBA objects.

   - **ORB (Object Request Broker)**: Middleware that handles method calls between clients and servers.

  - **Applications**: Useful in legacy systems and environments with multi-language interoperability requirements.

### 7. **Remote Method Invocation (RMI) - Practical Experiments**

  - **Concatenation, Substring Check, Reverse Check, Vowel Count**:

   - Each of these RMI experiments focuses on different operations on strings. The client sends strings to the server, which performs operations like concatenation, checking if one string is a substring/reverse of another, or counting vowels.

  - **Power Calculation**:

- For exponentiation, the client sends two numbers (base and exponent) to the server, which calculates the result and sends it back.

- **Key Concepts**:

- **Distributed Objects**: Objects residing on different machines.

- **Remote Interface**: Defines the methods that can be invoked from a remote client.

- **Execution**: Typically involves defining an interface, implementing it on the server, and using a client to invoke remote methods.

### 8. **MPI - Practical Experiments**

- **Prime Check, Feet to Meters, Degrees to Radians, Factorial**:

- Each of these MPI experiments involves a specific calculation:

- **Prime Check**: Determines if an integer is prime.

- **Feet to Meters** and **Degrees to Radians**: Convert units as per given formulas.

- **Factorial Calculation**: Computes the factorial of a given integer.

- **Key Concepts**:

- **Blocking Communication**: In `MPI_Send` and `MPI_Recv`, the sender/receiver waits until the message is completely transmitted.

- **MPI Initialization and Finalization**: `MPI_Init` and `MPI_Finalize` are used to start and end MPI programs.

- **Execution**: MPI code is compiled and run with multiple processes (`mpirun -np X ./program`), where processes handle different parts of the task.

### Additional Oral/Viva Questions and Answers

Here are some sample questions and concepts that might come up during the viva:

1. **What are the benefits of using MPI over RMI, or vice versa?**

- MPI is more suitable for high-performance computing and tightly coupled processes, while RMI is Java-specific and more suited for applications where remote method calls between objects are required.

2. **What is the significance of message passing in distributed systems?**

   - Message passing is crucial as it allows independent processes to communicate and coordinate actions without shared memory, making it ideal for distributed applications.

3. **How does RMI handle remote exceptions?**

   - RMI has built-in mechanisms to handle network and remote exceptions. The client must handle exceptions like `RemoteException`, which is thrown if the remote method invocation fails.

4. **Why is synchronization important in OpenMP?**

   - Synchronization is essential to avoid race conditions when multiple threads access shared data, ensuring data consistency.

5. **What role does the MPI rank play?**

   - The rank identifies each process uniquely in the MPI communicator, allowing processes to perform different tasks based on their rank.

6. **Explain the difference between Publish-Subscribe and Client-Server models.**

   - In Publish-Subscribe, the publisher doesn't directly send messages to subscribers; it posts to a topic, and subscribers receive messages based on subscriptions. In Client-Server, the client directly requests services from the server.

7. **How does CORBA differ from RMI?**

   - CORBA is language-independent and can be used across various programming languages, while RMI is Java-specific.

8. **How is load distribution handled in MPI?**

   - MPI doesn't automatically handle load distribution; the programmer must divide tasks among ranks based on application needs.

9. **What is an IDL in CORBA, and why is it necessary?**

- IDL defines the interface for CORBA objects in a language-independent way, enabling interoperability across different programming languages.

10. **What is the role of an ORB in CORBA?**

- ORB manages communication between clients and servers, handling requests and responses transparently.

Understanding these concepts should give you a solid foundation for your oral and viva. Let me know if you'd like to dive deeper into any specific topic!

Here are a few additional tips and deeper concepts that may help you prepare further:

### 1. **Synchronization and Deadlocks in Distributed Systems**

- **Synchronization**: In distributed systems, managing access to shared resources is crucial to avoid conflicts and maintain consistency.

- **Deadlocks**: When two or more processes are waiting indefinitely for each other to release resources, they enter a deadlock state. In MPI and RMI applications, care should be taken to avoid conditions that can lead to deadlocks.

- **Handling Deadlocks**: Techniques like ordering resource requests and using timeouts can help prevent deadlocks.

### 2. **Latency and Network Delays**

- In distributed systems, network latency can impact performance significantly. Experiments using RMI, Sockets, or MPI may suffer delays depending on network conditions.

- **Strategies to Mitigate Latency**:

    - **Caching**: Store frequently accessed data closer to the client.

    - **Load Balancing**: Distribute requests across multiple servers to avoid bottlenecks.

    - **Asynchronous Calls**: In RMI or MPI, asynchronous calls allow tasks to proceed without waiting for responses, improving responsiveness.

### 3. **Fault Tolerance**

  - Distributed applications should be designed to handle node failures gracefully.

  - **Techniques for Fault Tolerance**:

    - **Replication**: Maintain copies of data on multiple nodes to prevent data loss in case of failure.

    - **Heartbeat Mechanism**: Regularly check if a node or process is active and replace it if it fails.

    - **Retries**: Implement retry logic in RMI and MPI applications for operations that fail due to temporary network issues.

### 4. **Security Considerations**

  - Distributed systems need to address security concerns, particularly in client-server and remote method invocation contexts.

  - **Key Security Concepts**:

    - **Authentication and Authorization**: Ensure that only authorized users can access or modify resources.

    - **Encryption**: Encrypt data in transit to prevent eavesdropping.

    - **Firewalls and Access Control**: Control network access to prevent unauthorized communication between clients and servers.

### 5. **Serialization in RMI**

  - **Serialization**: In RMI, objects that need to be transmitted between client and server are serialized, meaning their state is converted into a byte stream.

  - **Use Cases**: Objects with primitive data types, strings, and collections can be serialized.

- **Limitations**: Not all objects are serializable, such as transient or sensitive information, so you may need to manage data carefully.

### 6. **Scalability and Load Distribution**

- **Scalability**: In distributed systems, adding resources (e.g., processors or nodes) should ideally improve performance.

- **Horizontal vs. Vertical Scaling**:

  - **Horizontal Scaling**: Adding more machines to distribute the load.

  - **Vertical Scaling**: Increasing the power of existing machines (e.g., adding more CPUs).

- **Load Balancing**: For MPI and socket-based applications, load balancing can help evenly distribute tasks to prevent any one node from becoming overloaded.

### 7. **Concurrency and Race Conditions**

- **Concurrency**: When multiple processes access shared resources, concurrency issues arise, potentially leading to incorrect outcomes.

- **Race Conditions**: A situation where multiple processes or threads attempt to read and write shared data simultaneously, leading to unpredictable results.

- **Avoiding Race Conditions**: Use synchronization mechanisms like locks, semaphores, or atomic operations (e.g., in OpenMP, use `#pragma omp critical`).

### 8. **Advanced Concepts in MPI and OpenMP**

- **Collective Communication**: MPI offers collective operations (e.g., `MPI_Bcast`, `MPI_Reduce`) where data is distributed among processes or results are gathered, optimizing communication.

- **Data Decomposition**: In OpenMP and MPI, problems can be divided based on data distribution (data parallelism) or functional distribution (task parallelism) to improve efficiency.

- **Dynamic vs. Static Scheduling (OpenMP)**: OpenMP allows you to control how iterations of a loop are scheduled across threads. Dynamic scheduling assigns chunks of work at runtime, which can improve load balancing but adds overhead.

### 9. **Code Optimization Techniques**

- **Minimizing Communication Overhead**: For MPI and RMI, communication is costly. Group data into larger messages or use fewer remote calls when possible.

- **Efficient Use of Memory**: Avoid unnecessary copying of data, especially in MPI and OpenMP where memory access patterns can impact performance.

- **Thread Affinity (OpenMP)**: Binding threads to specific cores can improve cache utilization and reduce context-switching.

### 10. **Real-World Applications and Use Cases**

- Think about where distributed systems are used in real-world applications:

  - **Scientific Computing**: MPI is widely used in scientific computations (e.g., simulations and large-scale data processing).

  - **Financial Services**: RMI and other middleware are often used in real-time trading applications.

  - **Web Services and Microservices**: Web services (SOAP, REST) enable communication in microservices architectures, often used in large-scale web applications.

- **Examples**: Having a few real-world examples of distributed systems (like Google's MapReduce, Hadoop for data processing) can help demonstrate your understanding.

### Potential Viva Questions on Advanced Topics

- **What are the advantages of using MPI collective operations?**

- **How do you handle serialization in Java RMI when sending complex objects?**

- **How does load balancing work in distributed systems, and why is it necessary?**

- **Explain how to prevent deadlocks in a distributed system.**

- **Discuss a real-world example of distributed computing and how it benefits from the architecture.**

- **Why is security challenging in distributed systems, and what measures can you take?**

These concepts will deepen your understanding of distributed systems, enhance your responses, and demonstrate your knowledge of both theoretical and practical aspects of each experiment. Let me know if you need further clarification on any of these points!