**Assignment 1:**

1. **RMI Registry**
   * Think of it like a *phonebook* where remote objects are registered with names.
   * The **server** adds (binds) its objects here so that **clients** can find them using the names.
2. **Remote Object** 
   * An object (like a class instance) on one computer that can be used (its methods called) from another computer.
   * It lives in its own Java program running on a different machine (or JVM).
3. **Remote Interface**
   * A Java interface that lists the methods that a remote object allows other programs to call.
   * It’s like a *contract* saying, “These are the methods you can use remotely.”
4. **RMI (Remote Method Invocation)**
   * A way for one Java program to call a method on an object that’s running in a different Java program (possibly on another computer), just like calling a local method.
5. **Stub**
   * A helper object on the **client side**.
   * When the client calls a method on the stub, it:
     1. Connects to the remote server.
     2. Sends the method and parameters.
     3. Waits for a reply and returns the result to the client.
   * You can think of the stub as a *remote control* for using the real object from far away.
6. **Skeleton** *(Used in older Java versions)*
   * A helper object on the **server side**.
   * It receives the request from the stub, calls the real method on the server, and sends the result back.
   * It's like a *translator* between the client and the real remote object.

**📌 Designing the Solution – Key Points:**

1. **Plan What to Share:**  
   Decide which part of your program will be shared and used by other computers (remote), and which will be used locally.
2. **Network Access:**  
   Make sure the parts that need to talk to each other over the network are accessible (not blocked by firewalls, etc.).
3. **Connection Between Apps:**  
   Make sure the apps (client and server) can connect over a network to exchange data using RMI.

**📘 Important Components:**

1. **Remote Interface:**
   * This is like a *menu card* showing which methods the server provides for clients to use.
   * It extends Remote interface.
   * All methods in it must say they might throw a RemoteException.
2. **Remote Object (Server-side Class):**
   * This class actually *does the work* of the methods described in the remote interface.
   * It extends UnicastRemoteObject.
   * It can also have other normal methods if needed.
3. **Remote Client:**
   * This is the program that *calls* the remote methods.
   * You can write the client after the remote interface is ready, even if the server isn't running yet.

**What does this program do?**

It uses Java RMI to allow a client and a server to **communicate over a network**.

* The **client** asks: *“Is this year a leap year?”*
* The **server** checks and responds **yes or no**.

**⚙️ Breakdown of Each Part:**

**✅ 1. ServerInterface**

📁 File: ServerInterface.java

* This is an interface that lists the method the client can call remotely.
* Method: isLeapYear(int year)
* It says: “I'll tell you if a year is a leap year.”

**✅ 2. ServerImplementation**

📁 File: ServerImplementation.java

* This class implements the interface and contains the actual leap year logic.
* It extends UnicastRemoteObject, so it's ready to be used remotely.

return (year % 4 == 0 && year % 100 != 0) || (year % 400 == 0);

✔ Follows the correct leap year rule.

**✅ 3. Server**

📁 File: Server.java

* Creates an object of the implementation class.
* Registers it (binds it) with the name **"LeapYearServer"** in the RMI registry.

So now clients can find and call it using that name.

**✅ 4. Client**

📁 File: Client.java

* Asks the user to enter a year.
* Connects to the server using Naming.lookup().
* Calls the isLeapYear() method remotely.
* Displays the result to the user.

**🖥️ How it Works (Flow):**

1. **Start the RMI Registry** (e.g., start rmiregistry)
2. **Run the Server** – it binds the object as "LeapYearServer".
3. **Run the Client** – it looks up the server, sends a year, and gets the result.

**Assignment 3:**

**Message Passing in Parallel Programming (MPI Basics)**

**Message Passing** is a way for different parts (processes) of a program to **talk to each other**, especially in **parallel or distributed computing**.

**🔌 Why use MPI?**

* Helps multiple processes run at the same time and **communicate**.
* Works well on both shared memory (like one computer) and distributed systems (like a network of computers).

**💡 Key MPI Concepts:**

| **Term** | **Simple Explanation** |
| --- | --- |
| **Group** | A set of processes that talk to each other. |
| **Communicator** | A tool for managing communication among processes. Default: MPI\_COMM\_WORLD. |
| **MPI\_Init** | Starts MPI. Must be called before using MPI functions. |
| **MPI\_Finalize** | Ends MPI and cleans everything up. |
| **MPI\_Comm\_size** | Tells how many processes are in a communicator. |
| **MPI\_Comm\_rank** | Tells the unique ID (rank) of a process. |
| **MPI\_Send** | Sends a message (data) from one process to another. |
| **MPI\_Recv** | Receives a message (data) from another process. |

**📦 Send & Receive Example**

* MPI\_Send(...): Used by sender to send data.
* MPI\_Recv(...): Used by receiver to receive that data.

**📢 Communication Collectives**

These are ways to send messages **to many processes at once**.

**🔄 Scatter**

* One process (root) has an array of data.
* It splits the array and sends **a part to each process**.
* Example:
  + Root has: A = [1, 2, 3, 4]
  + 4 processes => each gets one number: B = [1] for P0, [2] for P1, etc.

**What the Program Does**

* The program splits an array of 10 elements into 4 parts (using 4 processors).
* Each processor calculates a **partial (local) sum** of its part.
* Then all local sums are combined into one **global sum** using MPI.

**🧠 Key Concepts and Keywords**

| **Keyword** | **Simple Meaning** |
| --- | --- |
| MPI\_Init | Starts the MPI environment. Must be the first MPI call. |
| MPI\_Comm\_rank | Gets the ID (rank) of the current processor. |
| MPI\_Comm\_size | Gets the total number of processors. |
| rank | The unique ID of each processor (from 0 to n-1). |
| size | Total number of processors used. |
| local\_sum | The sum calculated by each individual processor. |
| global\_sum | The final total sum, computed by combining all local sums. |
| MPI\_Reduce | Combines data (like sum) from all processors into one. |
| MPI\_Send and MPI\_Recv | Used for communication: one processor sends, another receives. |
| MPI\_Finalize | Ends the MPI environment. Last MPI call. |

**⚙️ What Happens Step-by-Step**

1. **Array Initialization**  
   Array = [1, 2, 3, ..., 10]
2. **Processor Check**  
   The program must run using **exactly 4 processors**, otherwise it stops.
3. **Work Division**  
   Each processor works on a slice:
   * Rank 0 → elements 0–2 (1, 2, 3)
   * Rank 1 → elements 3–4 (4, 5)
   * Rank 2 → elements 5–6 (6, 7)
   * Rank 3 → elements 7–9 (8, 9, 10)
4. **Local Sum**  
   Each processor adds its values (e.g., Rank 0 → 1+2+3 = 6).
5. **Reduce to Global Sum**  
   All local sums are sent to Rank 0 using MPI\_Reduce.
6. **Printing Results**  
   Rank 0:
   * Prints each local sum it receives using MPI\_Recv.
   * Prints final total (global\_sum), e.g., 6+9+13+27 = **55**.

**Assignment 6:**

**Bully Algorithm:**

1. A process detects the coordinator is down.
2. It sends an **ELECTION** message to all higher-ID processes.
3. If no one replies, it becomes the **coordinator**.
4. If any higher process replies with **OK**, it waits; the higher process takes over.
5. The new coordinator sends a **COORDINATOR** message to all.
6. If a previously down process comes back and has the highest ID, it can **bully** and become the coordinator.

**🔁 Ring Algorithm:**

1. A process notices the coordinator is down and starts an **ELECTION**.
2. It sends a message with its ID to its **next active neighbor** in the ring.
3. Each process adds its ID and passes the message.
4. When the message returns to the initiator, it finds the **highest ID** as the new coordinator.
5. It then sends a **COORDINATOR** message around the ring.

**💻 Implementation Steps (Ring):**

1. Create a Process class with:
   * State (Active/Inactive)
   * Index
   * ID
2. Use Scanner to get input from the user.
3. Ask for number of processes and their details.
4. Sort the processes by ID.
5. Mark the process with the highest ID as inactive (simulate failure).
6. Show menu:
   * 1. Start Election
     2. Quit
7. Ask the user to select a process to start the election.

**urpose:**

This Java program simulates the **Bully Algorithm** for leader election in a distributed system. The **process with the highest ID that is active becomes the coordinator**.

**🔑 Important Concepts / Keywords:**

| **Keyword** | **Meaning** |
| --- | --- |
| boolean[] | An array to track whether each process is UP (true) or DOWN (false) |
| coordinator | Stores which process is currently the leader |
| Scanner | Takes user input |
| switch-case | Used for menu selection |

**🧠 Main Methods Explained Simply:**

1. **getStatus()**  
   ➤ Displays which processes are **UP/DOWN** and shows the **current coordinator**.
2. **up(int up)**  
   ➤ Turns a DOWN process back UP.  
   ➤ Starts an election from that process if not process 5.  
   ➤ The highest active process becomes the new **coordinator**.
3. **down(int down)**  
   ➤ Turns a process DOWN.  
   ➤ If the down process was the coordinator, a new coordinator is selected.
4. **mess(int mess)**  
   ➤ Simulates sending a message from a process.  
   ➤ If coordinator is DOWN, it starts an **election**.
5. **setCoordinator()**  
   ➤ Finds and sets the **highest active process** as the new coordinator.

**🖥️ Output (Initial):**

plaintext

CopyEdit

+------Current System State-----+

| P1: UP |

| P2: UP |

| P3: UP |

| P4: UP |

| P5: UP <-- COORDINATOR |

+-------------------------------+

* All 5 processes are **UP**
* **P5** is the coordinator by default because it has the **highest ID**

**📋 Menu Options:**

plaintext

CopyEdit

+........MENU........+

1. Activate a process.

2. Deactivate a process.

3. Send a message.

4. Exit.

+....................+

You can:

* **1:** Turn ON a process (if it's DOWN)
* **2:** Turn OFF a process
* **3:** Send a message to test if the coordinator is alive
* **4:** Exit the program

**🧪 Example Scenario:**

* You turn off P5 → P4 becomes coordinator.
* You send a message from P2 → detects P5 is down → election starts → P4 wins.

**Assignment 7:**

A **web service** allows different systems or applications to communicate with each other over a network (e.g., the internet or intranet). It's designed using open protocols and standards, mainly XML, to ensure compatibility across platforms and programming languages. Key characteristics of a web service:

* **Discoverable**: Can be found easily.
* **XML-based messaging**: Standard format for exchanging information.
* **Accessible across networks**: Can work over the internet or intranet.
* **Self-describing**: The service provides its details in XML format.
* **Platform-independent**: Can work across different systems, e.g., Java to .NET communication.

**Types of Web Services:**

1. **SOAP (Simple Object Access Protocol)**:
   * An XML-based protocol for accessing web services.
   * **Platform and language-independent**.
   * Used for more complex enterprise solutions.
2. **REST (Representational State Transfer)**:
   * An architectural style for web services.
   * More lightweight and easier to use than SOAP.
   * Uses **URIs** to uniquely identify resources.

**Key Components of Web Services:**

* **Service Provider**: The program that implements the service.
* **Service Requestor**: The program that calls and interacts with the service.
* **Service Registry**: A directory that stores references to services.

**SOAP Web Service Steps:**

1. **Client** creates a request in SOAP format.
2. **Client** sends the request to the **server** via HTTP.
3. **Server** processes the request and generates a response.
4. **Server** sends the response back in SOAP format to the **client**.
5. **Client** reads the SOAP response.

**SOAP Web Services Components:**

* **UDDI (Universal Description, Discovery, and Integration)**: A directory for discovering web services.
* **WSDL (Web Services Description Language)**: An XML document that describes the web service's operations and how to interact with it.

**RESTful Web Services:**

* **REST** is an architectural style, not a protocol.
* It is simpler and more performance-efficient compared to SOAP.

**Java APIs for Web Services:**

* **JAX-WS**: Java API for SOAP-based web services.
* **JAX-RS**: Java API for RESTful web services (uses annotations for simplicity).

**Implementing Web Services:**

**Step 1**: Create a web service project in your IDE (e.g., GlassFish or Tomcat).

* Select **Web Application** and create the project.

**Step 2**: Create a web service from a Java class.

* Right-click the project, select **New > Web Service**, and define the service details.

**Step 3**: Add operations to the web service.

* Define operations like "add" by specifying input and output parameters.

**Step 4**: Deploy and test the web service.

* Deploy to a server (GlassFish or Tomcat), and test the service using built-in test clients.

**Step 5**: Consume the web service.

* Create a client to interact with the service (e.g., Java SE application or web application).

**Short Example of Using Web Service:**

1. **Server** creates a "calculator" web service with an "add" method.
2. **Client** (Java SE) calls this "add" method by passing two integers, and the result is displayed.

**Web Service Code (webservice.py):**

1. **Flask**: A web framework for Python used to create web applications, including web services.
2. **@app.route('/add', methods=['POST'])**: This defines an endpoint /add for the web service, which listens for POST requests. The function add() will be called when this route is accessed.
3. **request.get\_json()**: It retrieves JSON data sent with the POST request.
4. **Addition Logic**:
   * Extracts num1 and num2 from the JSON data.
   * Adds the two numbers together.
5. **return {'result': result}**: Returns the sum in JSON format to the client.
6. **app.run()**: Runs the Flask server locally at http://127.0.0.1:5000.

**Output Explanation**:

* The server starts and listens on http://127.0.0.1:5000.
* A POST request is made to /add with the numbers 5 and 10.
* The server responds with the result: {"result": 15}.

**Distributed Application Code (app.py):**

1. **requests**: A library used to make HTTP requests to a web service.
2. **url = 'http://localhost:5000/add'**: Defines the URL of the web service endpoint (the add method in webservice.py).
3. **data = {'num1': num1, 'num2': num2}**: Prepares the data to be sent in JSON format to the web service.
4. **response = requests.post(url, json=data)**: Sends a POST request to the web service with the data.
5. **response.json()['result']**: Parses the JSON response from the server and extracts the result.
6. **Return the result**: The add\_numbers function returns the sum of the numbers.

**Output Explanation**:

* The client (app.py) sends numbers 5 and 10 to the web service.
* The web service returns the sum (15), and the client prints the result.

**Console Output**:

csharp

CopyEdit

The result of adding 5 and 10 is: 15

**Keywords Explanation in Simple Words:**

* **Flask**: A tool to create web services and applications.
* **POST request**: A way to send data to a web service.
* **JSON**: A format to send data between applications (like a dictionary).
* **Web Service**: A program that performs operations (like adding numbers) and can be used by other programs over a network.
* **requests**: A Python library for making web requests.
* **Result**: The outcome of a computation or operation (in this case, the sum).