`Naming.lookup(addServerURL)` is used to retrieve a reference to the remote object (the server object) that is registered with the RMI registry.

RMI (Remote Method Invocation):

- RMI is a Java API that allows objects to communicate with each other over a network, even if they are running on different machines.

- **MPI Initialization:**
  - The program uses **MPI (Message Passing Interface)** for parallel computing.
  - `MPI.Init(args)` initializes the MPI environment, which is required before any MPI communication can occur.
  - `MPI.COMM_WORLD.Rank()` retrieves the rank of the current process, i.e., its ID.
  - `MPI.COMM_WORLD.Size()` retrieves the total number of processes involved.

- **Data Initialization (Root Process):**
  - `if (rank == root)` ensures only the **root process (rank 0)** generates the data to be distributed.
  - The root process creates an array of integers (`send_buffer`) and populates it with numbers from 0 to `total_elements - 1`. This array will be scattered across all processes.

- **Scatter Operation:**
  - The `MPI.COMM_WORLD.Scatter()` function distributes portions of the array from the root process to all other processes. Each process receives a chunk of the array into the `recieve_buffer`.
  - **Key**: The array is divided into chunks of size `unitsize`, and each process receives one chunk.

- **Sum Calculation (Local Computation):**
  - After scattering, each process computes the **sum** of its chunk of the array. This is done by iterating through the `recieve_buffer` and adding the values into `recieve_buffer[0]`.
  - This produces an **intermediate sum** at each process.
  - For example, if a process receives the numbers `[1, 2, 3, 4, 5]`, the sum will be `15`, which is stored at `recieve_buffer[0]`.

- **Gather Operation:**
  - `MPI.COMM_WORLD.Gather()` is used to collect the intermediate sums from all processes back to the root process.
  - The root process will receive an array (`new_recieve_buffer`) that contains the summed results from all the processes.

- **Final Aggregation (Root Process):**
  - The root process aggregates the intermediate sums from all processes by summing them up, producing the **final sum** of all the numbers.
  - The final sum is printed by the root process.
- **MPI Finalization:**
  - `MPI.Finalize()` is called to clean up the MPI environment once all communication is complete.

Key Terminologies of RMI:

1. **Remote Object**: An object in a JVM whose methods are accessible remotely by another program in a different JVM.
2. **Remote Interface**: A Java interface defining methods that are exposed by a remote object for remote invocation.
3. **RMI (Remote Method Invocation)**: Allows invoking methods on a remote object using syntax similar to local method invocation, enabling communication across different JVMs.
4. **Stub**: A client-side object that acts as a proxy for the remote object. It initiates the connection and handles communication with the remote JVM.
5. **Skeleton**: A server-side object that receives requests from the client (via the stub), invokes the actual method on the remote object, and sends the results back to the client.

- **Threading**: To handle multiple client requests concurrently, the server can be multi-threaded. Each client can be assigned a separate thread for processing, enabling efficient handling of multiple clients.

- **Concurrency Management**: The server can spawn multiple threads for handling requests, and the RMI mechanism ensures proper synchronization when invoking methods on the remote object.

```
MPI.COMM_WORLD.Scatter(
    send_buffer,   // Data array at root to distribute
    0,             // Starting index in send_buffer
    unitsize,      // Number of elements to send to each process
    MPI.INT,       // Datatype of send_buffer elements
    recieve_buffer,// Buffer to store received elements
    0,             // Starting index in recieve_buffer
    unitsize,      // Number of elements to receive
    MPI.INT,       // Datatype of recieve_buffer elements
    root           // Rank of root process doing the scatter
);
```

What is CORBA?

Answer:

- CORBA stands for **Common Object Request Broker Architecture**.
- It is a **middleware** that allows programs written in different languages and running on different platforms to work together by communicating over a network.
- It uses an **ORB (Object Request Broker)** to handle communication between client and server objects.
- It allows **distributed object communication** across heterogeneous systems.

---

2. What is IDL (Interface Definition Language) in CORBA?

Answer:

- IDL is used to **define interfaces** that client and server will use for communication.
- It is **language-independent** — from IDL, language-specific stubs and skeletons are generated automatically.

---

3. Why do we need a Naming Service in CORBA?

Answer:

- Naming Service helps to **register** server objects with a human-readable **name** (like "Reverse").
- Clients **look up** the object using the name to get the reference and start communication.

---

4. Explain client-server communication steps in your program.

Answer:

- Server creates `ReverseImpl` object and **registers** it with Naming Service.
- Client **uses ORB** to connect to Naming Service and **resolves** the "Reverse" object reference.
- Client sends a string to server, server reverses it and **sends back** the reversed string.

---

5. What is the role of POA (Portable Object Adapter)?

Answer:

- POA **manages** server-side objects.
- It **activates**, **registers**, and **binds** servant objects (like ReverseImpl) to the CORBA environment.
- Helps in object lifecycle management.

---

6. How does the client find the server object?

Answer:

- The client uses ORB to contact the Naming Service, then **resolves** the server object by its **registered name** ("Reverse").

---

7. What is the use of servant in CORBA?

Answer:

- A **servant** is the server-side implementation (like `ReverseImpl`) that actually **processes client requests**.

---

8. How is ORB used in client and server?

Answer:

- In **server**, ORB initializes communication and connects the servant to Naming Service.
- In **client**, ORB is used to connect to Naming Service and invoke methods on server object.

What is a Distributed System?

Answer:
A distributed system is a collection of independent computers (nodes) connected by a high-speed network, communicating and coordinating their actions by message passing. They share resources and work together to perform tasks.

---

2. Why is Clock Synchronization needed in Distributed Systems?

Answer:
Clock synchronization ensures that all nodes have a consistent view of time, which is important for:

- Correct ordering of events
- Coordinating resource sharing
- Preventing conflicts
- Maintaining consistency

---

3. What are External and Internal Clock Synchronization?

Answer:

- **External Synchronization:** Synchronization with an external time source (like UTC servers).
- **Internal Synchronization:** Synchronization among nodes without any external reference, adjusting their local clocks based on each other's time.

---

4. What is the Berkeley Algorithm?

Answer:
Berkeley Algorithm is a **centralized clock synchronization** technique where:

- A **master node** polls the time from all slave nodes.
- It calculates the average time difference.
- It adjusts its own clock and tells all slave nodes how much to adjust their clocks.

---

5. Why is Berkeley Algorithm called a Centralized Algorithm?

Answer:
Because it depends on a single master node to collect, calculate, and distribute the clock adjustments. Failure of the master leads to synchronization failure.

---

6. What improvements can be made to the Berkeley Algorithm?

Answer:

- Ignoring **outlier clock times** while averaging.
- Having a **backup master node** ready in case of master failure.
- Broadcasting **relative inverse time difference** instead of absolute time to reduce latency.

---

7. What is the difference between Centralized and Distributed Synchronization Algorithms?

Answer:

- **Centralized:** One master/server controls synchronization (example: Berkeley).
- **Distributed:** All nodes participate equally, without a master (example: NTP - Network Time Protocol).

---

8. What is Cristian's Algorithm?

Answer:
Cristian's Algorithm is used to synchronize clocks by:

- Requesting the current time from a time server.
- Adjusting for network delay by measuring round-trip time.

---

9. What happens if the master node fails during Berkeley Synchronization?

Answer:

- The synchronization process halts.
- It is necessary to have a **backup master node** already elected to take over immediately.

---

10. Explain the working steps of the Berkeley Algorithm.

Answer:

- Master sends a request to slaves asking for their local time.
- Slaves respond with their current clock time.
- Master calculates the average time difference.
- Master adjusts its own clock.
- Master sends the adjustment values to slaves to correct their clocks.

If the examiner asks you to **draw a diagram**, quickly draw:

```
rust
CopyEdit
Master Node -> Sends Time Request -> Slave Nodes
Slave Nodes -> Send Back Their Time -> Master Node
Master Node -> Calculate Average -> Sends Adjustment Instructions -> Slave
Nodes
```

What is the Berkeley algorithm? How does it work?

Answer: Berkeley Algorithm is a **centralized, internal synchronization algorithm**.

- One node is elected as a **master** (leader).
- The master **polls** all other nodes for their times.
- It **calculates the average time** difference, ignoring outliers if needed.

- The master then **instructs each node** to adjust its clock accordingly.

. Explain how server.py is implementing the Berkeley Algorithm.

Answer:

- `server.py` acts as the **master node**.
- It **accepts connections** from multiple slave clients.
- It **collects the time** from each client.
- It calculates the **average time difference** using `getAverageClockDiff()`.
- It **sends the corrected time** back to each client using `synchronizeAllClocks()`.

---

5. Explain the role of client.py.

Answer:

- `client.py` acts as a **slave node**.
- It **sends its local clock time** periodically to the server.
- It **receives the synchronized time** from the server.
- It **displays the updated synchronized time** to the user.

---

6. Why are you using threads in both server.py and client.py?

Answer:

- Threads allow **parallel activities**.
- In the client:
  - One thread **sends local time** to the server.
  - Another thread **receives synchronized time**.
- In the server:
  - One thread **handles each client** connection separately.
  - Another thread **handles synchronization cycles**. This way, communication is **non-blocking** and efficient.

---

10. How is the average clock difference calculated in the server?

Answer:

- In `getAverageClockDiff()`:
  - It **collects all time differences** between server and each client.
  - It **sums** these differences.
  - Divides by the **number of clients** to get the **average** time difference.

What is the purpose of implementing the Token Ring algorithm?
A1. The Token Ring algorithm ensures **mutual exclusion** in a distributed system. Only the process holding the token can access the critical section (shared resource), guaranteeing exclusive access without conflicts.

Q2. What are the main properties a mutual exclusion algorithm must satisfy?
A2.

- **Safety**: Only one process accesses the critical section at a time.
- **Liveness**: Every process can eventually access the critical section.
- **Fairness**: No process suffers indefinite postponement (no starvation).

Q3. How does the Token Ring algorithm work?
A3.

- A logical ring of processes is created.
- A token circulates around the ring.
- A process can enter the critical section only when it holds the token.
- After completing its task or if it doesn't need access, the token is passed to the next process.

Q5. What are the advantages of the Token Ring algorithm?
A5.

- Simple and easy to implement.
- Ensures fairness — no process is starved.
- Mutual exclusion is guaranteed.

Q6. What is the main drawback of the Token Ring algorithm?
A6. If the token is lost (e.g., due to process crash or network failure), it needs to be regenerated. Detecting and regenerating a lost token is complex.

How does the code implement the token passing mechanism?

Answer:

- The token is represented by an integer (`token` variable).
- The code moves the token through nodes by incrementing index `i` using modulus operation `(i + 1) % n` until the sender is reached.
- After sending data, the token is set to the sender (`token = s`) to continue the cycle from there in the next iteration.

4. Explain how data is transmitted from sender to receiver in the code.

Answer:

- After the sender is reached, the sender sends the data.
- The data is then **forwarded** by each intermediate node (using a loop) until the **receiver** is reached.
- Finally, the receiver **receives** the data and prints a message.

What is the Bully Algorithm? Why is it called "bully"?

Answer:

- **Bully Algorithm** is an election algorithm used when **any process can directly communicate with any other process**.
- In this algorithm, the process with the **highest process ID** always wins the election.
- It's called "**Bully**" because **higher-numbered processes** "bully" lower-numbered ones by taking over elections and becoming the coordinator.

3. What are the basic steps in the Bully Algorithm?

Answer:

1. A process detects that the coordinator is not responding.
2. It sends an **ELECTION** message to **all higher-numbered processes**.
3. If **no higher-numbered process** responds, it becomes the coordinator.
4. If a higher process responds with an **OK** message, that process will take over the election.
5. The winning process sends a **COORDINATOR** message to all others, announcing itself as the new coordinator.

WSDL (Web Services Description Language) is an XML document that describes the functionality of a web service. It defines the operations that can be performed, the parameters to be passed, and the structure of the response. WSDL allows a service consumer to understand how to interact with the web service without needing to know the implementation details.

1. The client creates a SOAP message with the request content.
2. The client sends this SOAP message to the web service via HTTP POST.
3. The web service receives and processes the SOAP request.
4. The server returns the response in the form of a SOAP message.
5. The client reads the SOAP response and processes the result.