RMI

Implement multi-threaded client/server Process communication using RMI.

RMI (Remote Method Invocation) is a Java-based technology that allows communication between distributed systems. It enables a Java object running in one JVM (Java Virtual Machine) to invoke methods on a Java object running in another JVM, possibly on a different machine.

Certainly! Let's go through each of the provided code snippets and explain their functionality:

Client.java:
This code represents the client-side of an RMI (Remote Method Invocation) application. It performs the following steps:

1. It creates a `Scanner` object to read input from the user.

2. The user is prompted to enter the server address.

3. The entered server address is used to construct the complete server URL in the RMI registry format.

4. The `Naming.lookup` method is used to look up the remote object with the specified server URL. The returned object is cast to the `ServerInterface` interface.

5. The user is prompted to enter two strings.

6. The `concat` method of the remote `ServerInterface` object is invoked with the entered strings as arguments.

7. The concatenated string returned by the server is printed to the console.

8. The `Scanner` object is closed.

Servant.java:
This code represents the server-side implementation of the remote object. It includes the following details:

1. The `Servant` class extends the `UnicastRemoteObject` class and implements the `ServerInterface` interface.

2. The constructor of `Servant` calls the constructor of `UnicastRemoteObject` and throws a `RemoteException`.

3. The `concat` method is implemented as per the `ServerInterface` interface requirements. It concatenates two strings and returns the result.

Server.java:
This code represents the server program that publishes the `Servant` object for remote access. It performs the following steps:

1. The `main` method is the entry point of the program.

2. An instance of the `Servant` class is created.

3. The `Naming.rebind` method is used to bind the `Servant` object to the name "Server" in the RMI registry. This makes the object accessible to remote clients.

In summary, the client program prompts the user for input, invokes a remote method on the server, and receives the result. The server program publishes the remote object for clients to access through RMI. The `ServerInterface` defines the contract for remote communication between the client and server.

CORBA

CORBA (Common Object Request Broker Architecture) is a middleware technology that enables communication between distributed objects across different platforms and programming languages. It provides a standardized mechanism for objects to interact with each other seamlessly, regardless of their location or implementation language.

The provided code is a Java server that implements a remote object service using CORBA (Common Object Request Broker Architecture). Here's a summary of what the code does:

1. The server creates and initializes the ORB (Object Request Broker) to handle communication.

2. It obtains the reference to the Root POA (Portable Object Adapter) and activates it to manage objects.

3. An instance of the `HelloImpl` class, which implements the `Hello` interface, is created. This class will serve as the remote object.

4. The server registers the servant object reference with the Root POA.

5. The server obtains the object reference for the naming service and narrows it to the appropriate type.

6. It binds the servant object reference to a name in the naming context.

7. The server starts the ORB event processing loop, allowing it to receive invocations from clients.

In short, the server sets up the necessary infrastructure for a CORBA-based remote object service, registers and binds the servant object with the naming service, and then waits for client invocations.

MPI

MPI (Message Passing Interface) is a standardized message-passing system that facilitates communication and coordination among multiple processes in a parallel computing environment. It is commonly used in high-performance computing (HPC) applications to achieve parallel execution across a cluster of machines.

Sure! The code you provided calculates the sum of elements using distributed computing with MPJ (Message Passing Interface for Java).
1. The code initializes MPI and gets the rank and size of the communicator (the group of processes).

2. It defines the unit size (number of elements each process will handle), the root process (process with rank 0), and creates arrays for sending and receiving data.

3. If the current process is the root process, it prompts the user to enter a certain number of elements and stores them in the send buffer array.

4. The data is scattered from the root process to all the other processes using the MPI.COMM_WORLD.Scatter() method. Each process receives a portion of the data into its receive buffer.

5. Each process calculates the sum of its received data and stores the result in the first index of its receive buffer.

6. The intermediate sum at each process is printed to the console.

7. The calculated results from all processes are gathered back to the root process using the MPI.COMM_WORLD.Gather() method. Each process sends its result to the root process, which collects them in the new_receive_buffer array.

8. The root process then calculates the final sum by summing up all the results in the new_receive_buffer array.

9. The final sum is printed to the console by the root process.

10. Finally, MPI is finalized and the program terminates.

This code demonstrates the basic usage of MPI in Java to distribute data and perform parallel computations across multiple processes. Each process calculates its part of the sum, and the results are combined to obtain the final sum.

Clock Synchronize

Clock synchronization refers to the process of aligning the clocks of multiple devices or systems to a common time reference. It is essential in distributed systems where coordination and time-dependent operations are required.

In distributed systems, different machines or nodes may have their own local clocks, which can be affected by various factors such as clock drift, network delays, and hardware variations. Clock synchronization aims to minimize the discrepancies between these local clocks to ensure consistency and coordination across the system.

Certainly! Here's a brief explanation of both the client and server codes:

Client Code:
The client code represents a client that connects to a server using a TCP socket. It performs the following steps:
1. Creates a socket and establishes a connection with the server.
2. Receives a message from the server requesting the client's local clock value.
3. Sends its local clock value back to the server.
4. Receives a message from the server containing the clock adjustment offset.
5. Updates its local clock based on the received offset.
6. Closes the socket and exits.

Server Code:
The server code represents a server that accepts connections from multiple clients and synchronizes their local clocks. It performs the following steps:
1. Creates a socket and binds it to a specified port.
2. Listens for incoming connections from clients.
3. Accepts connections from clients and stores their socket descriptors, IP addresses, and port numbers.
4. Waits until a sufficient number of clients have connected.
5. Requests the local clock values from all connected clients.
6. Calculates the average clock value based on the received values.
7. Computes the clock adjustment offsets for each client and sends them back.
8. Adjusts its own local clock based on the average clock value.
9. Closes the server socket and exits.

Both the client and server use TCP sockets for reliable communication. The client exchanges messages with the server to synchronize its local clock, while the server coordinates the clock adjustment among multiple clients.

Token Ring

A token ring-based mutual exclusion algorithm is a distributed algorithm that enables processes in a distributed system to acquire exclusive access to a shared resource in a mutually exclusive manner. This algorithm is based on the concept of a token, which is passed sequentially among the participating processes in a logical ring structure.

The provided code consists of three Java classes: `TokenClient1`, `TokenClientHelper`, and `TokenServer`. These classes represent a token-based communication system involving a client and a server.

The `TokenClient1` class represents a client program. It creates instances of the `TokenClientHelper` class, sets up network connections, and manages the flow of data. The client alternates between two modes: sending data and receiving data. If the client has the token, it prompts the user to enter data and sends it to the server. If the client does not have the token, it enters the receiving mode and waits to receive data from the server.

The `TokenClientHelper` class serves as a helper class for the client program. It handles tasks such as setting up ports, sending data, and receiving data. It uses `DatagramSocket` and `DatagramPacket` classes for network communication. The class has instance variables to track the local host, send and receive ports, and the state of sending data and having the token. It provides methods to set the ports, send data, and receive data.

The `TokenServer` class represents the server program. It continuously listens for incoming data from the client. It creates an instance of the `Server` class within each iteration, sets the receive port, and calls the `recData()` method to receive data.

The `Server` class handles the server's functionality. It has instance variables to track the state of having the token and sending data, as well as the receive port. The class provides methods to set the receive port and receive data from the client.

Together, these classes implement a token-based communication system where clients and servers take turns sending and receiving data using tokens. The client program manages user input and interacts with the `TokenClientHelper` class for sending and receiving data. The server program continuously listens for incoming data from clients using the `Server` class.

Bully

The Bully algorithm is a distributed algorithm used for leader election in a decentralized system. It allows a group of processes to elect a leader when the current leader fails or becomes unavailable. The algorithm assumes that each process has a unique identifier, and the process with the highest identifier is elected as the leader.

This Java program simulates a simplified version of the Bully Algorithm for leader election in a distributed system. Here's a brief explanation of its functionality:

The program starts by initializing an array called `state` which represents the states of five processes (p1, p2, p3, p4, and p5). The `coordinator` variable is used to keep track of the current coordinator process.

The program then enters a loop where the user is presented with a menu of options:

1. **Up a process**: This option brings a specific process up by setting its corresponding state to `true` in the `state` array. If the process being brought up is p5, it becomes the coordinator.

2. **Down a process**: This option brings a specific process down by setting its corresponding state to `false` in the `state` array.

3. **Send a message**: This option allows a process to send a message. If the process is up (as indicated by its state in the `state` array), it checks if the current coordinator process (p5) is up. If the coordinator is up, the message is considered successfully delivered. If the coordinator is down, an election process starts. The process sends an election message to higher-numbered processes to determine if any of them are still up and can become the new coordinator. The process with the highest number that responds becomes the new coordinator, and a coordinator message is sent by the new coordinator to all processes.

4. **Exit**: This option exits the program.

The program uses a `Scanner` object to read user input from the console.

Overall, this program provides a basic implementation of the Bully Algorithm to demonstrate the process of electing a leader in a distributed system.

Ring

The Ring algorithm for leader election is a distributed algorithm that allows a group of processes in a ring topology to elect a leader among themselves. The leader is typically chosen based on specific criteria, such as the highest process identifier or a predefined priority.

The code you provided is an implementation of the Ring Election algorithm in Java. This algorithm is used to elect a coordinator in a distributed system where processes are organized in a ring.

Here's a breakdown of the code:

1. The code defines a class called `Ring` with the `main` method as the entry point of the program. It also defines another class called `Rr`, which is a helper class representing a process in the ring.

2. The code creates an array of `Rr` objects called `proc` to hold the processes.

3. The user is prompted to enter the number of processes and then input the details (id) for each process.

4. The processes are sorted based on their IDs using a simple bubble sort algorithm.

5. The last process in the sorted array is set as the coordinator by marking its state as "inactive".

6. The program enters a loop where the user can choose to initiate an election or quit the program.

7. If the user chooses to initiate an election (option 1), they need to enter the process number that initializes the election.

8. The election process begins by passing a token/message around the ring until it reaches the process with the highest ID. Each process updates its status and forwards the token to the next active process.

9. Once the token reaches the process with the highest ID, it selects itself as the coordinator and informs all other processes about its selection by updating their states.

10. If the user chooses to quit the program (option 2), the program terminates.

That's an overview of the code's functionality. It implements the basic steps of the Ring Election algorithm for coordinator selection in a distributed system.