**Assihnment 1 :**

This code demonstrates a simple example of using Java RMI (Remote Method Invocation) to perform string concatenation on a remote server. RMI allows a Java program to invoke methods on objects that reside on a remote machine. Here's an explanation of each file in the code:

1. `Client.java`: This is the client-side code. It prompts the user to enter the server address, connects to the server using RMI, and then requests the user to enter two strings. It calls the `concat` method on the remote server object (of type `ServerInterface`) to concatenate the two strings and displays the result.

2. `Servant.java`: This file implements the `ServerInterface` interface and extends `UnicastRemoteObject`, which provides functionality for remote object communication. The `concat` method in this class takes two strings as parameters and concatenates them. It throws a `RemoteException` because remote method calls can potentially result in communication errors.

3. `Server.java`: This is the server-side code. It creates an instance of the `Servant` class and binds it to a name (in this case, "Server") using the `Naming.rebind` method. This makes the server object available for remote invocation.

4. `ServerInterface.java`: This interface extends `Remote`, indicating that its methods can be invoked remotely. It declares a single method `concat`, which takes two strings as parameters and returns the concatenated string. It also throws a `RemoteException` to handle any remote communication errors.

To summarize the flow of execution:

1. The server is started by running the `Server` class.

2. The client is run by executing the `Client` class. It prompts for the server address and establishes a connection to the server using RMI.

3. The client prompts for two strings, and after receiving the inputs, it invokes the `concat` method on the remote server object to perform the string concatenation.

4. The server receives the method invocation and executes the `concat` method, returning the concatenated result.

5. The client displays the concatenated string.

Note: In order for this code to work, you need to have the RMI registry running on the server machine.

RMI :

RMI stands for Remote Method Invocation, which is a Java API used for implementing distributed applications. It allows Java objects to invoke methods on remote objects running on different Java Virtual Machines (JVMs), enabling communication between distributed components of an application.

RMI uses a client-server architecture, where the client initiates a method call on a remote object, and the server (remote object) executes the method and returns the result to the client. RMI handles the communication between the client and server transparently, making it appear as if the remote method call is a local method invocation.

**Assigment 2 :**

This code represents a client-server application for a basic calculator using CORBA (Common Object Request Broker Architecture) in Java. CORBA allows objects to communicate with each other across a network, regardless of the programming language or hardware platforms they are implemented on. The application consists of a client-side file (`CalcClient.java`) and a server-side file (`CalcServer.java`).

The `CalcClient.java` file:

- This file contains the client-side logic for interacting with the calculator server.

- It imports various packages and classes required for CORBA, input/output operations, and exceptions.

- The main logic is implemented inside the `main` method.

- The code creates and initializes the ORB (Object Request Broker) using the `ORB.init` method.

- It retrieves the root naming context and the object reference for the calculator server from the naming service.

- The code displays a menu to the user with options for different calculator operations: sum, subtraction, multiplication, division, and exit.

- Based on the user's choice, the corresponding calculator operation is invoked on the server-side object (`calcImpl`) obtained from the object reference.

- The result is displayed to the user.

- The program continues to display the menu until the user chooses to exit.

The `CalcServer.java` file:

- This file contains the server-side logic for the calculator.

- It imports various packages and classes required for CORBA, the Portable Object Adapter (POA), and properties.

- The `CalcImpl` class implements the `Calc` interface generated from the IDL (Interface Definition Language).

- The `CalcImpl` class provides implementations for the calculator operations: sum, subtraction, multiplication, and division.

- The division operation checks if the divisor is zero and throws a custom `DivisionByZero` exception if it is.

- The `CalcServer` class contains the main logic for the server.

- It creates and initializes the ORB.

- It retrieves the reference to the root POA (Portable Object Adapter) and activates the POA manager.

- An instance of `CalcImpl` is created and registered with the ORB.

- An object reference (`ref`) is obtained from the servant and narrowed to the `Calc` interface.

- The root naming context is retrieved, and the object reference is bound to a name in the naming context.

- The server is set to wait for invocations from clients using `orb.run()`.

In summary, this code establishes communication between a client and server using CORBA for a basic calculator application. The client interacts with the server by invoking calculator operations, and the server performs the requested operations and returns the results to the client.

CORBA (Common Object Request Broker Architecture) is a middleware technology and architecture developed by the Object Management Group (OMG). It is a standard that enables interoperability and communication between distributed objects implemented in different programming languages and running on different platforms.

**Assignment 3 :**

This code demonstrates parallel summation of an array using multiple threads. Let's go through it step by step:

1. The code begins by importing the necessary Java libraries: `java.util.Arrays` for array manipulation, `java.util.concurrent.ExecutorService` and `java.util.concurrent.Executors` for managing thread execution, and `java.util.concurrent.TimeUnit` for specifying time units.

2. The class `ParallelSum` is defined, which contains the `main` method where the execution of the program starts.

3. The constant variable `NUM\_THREADS` is declared and assigned the value of the number of available processors in the system, obtained using `Runtime.getRuntime().availableProcessors()`. This will determine the number of threads to be used for parallel processing.

4. An array `array` is created as a sample array to be summed.

5. The array is divided into equal parts to be processed by each thread. The variable `partitionSize` is calculated as the array length divided by the number of threads. An empty two-dimensional array `partitions` is created to store the divided parts of the array.

6. A loop is used to divide the array and populate the `partitions` array. Each partition is obtained using `Arrays.copyOfRange()` method, which creates a new array containing elements from the specified range of the original array.

7. An `ExecutorService` is created using `Executors.newFixedThreadPool(NUM\_THREADS)`, which creates a thread pool with a fixed number of threads equal to `NUM\_THREADS`.

8. An array of `SumTask` objects, named `tasks`, is created to store the tasks to be executed by the threads.

9. Another loop is used to create and submit `SumTask` objects to the executor. Each `SumTask` object is initialized with a partition of the array, and then the task is submitted to the executor using `executor.execute(tasks[i])`.

10. After all the tasks have been submitted, the executor is shut down using `executor.shutdown()` to indicate that no more tasks will be submitted.

11. The main thread waits for all tasks to complete by calling `executor.awaitTermination(Long.MAX\_VALUE, TimeUnit.NANOSECONDS)`. This method blocks until all tasks have completed execution or the specified timeout period has elapsed.

12. In case of an interruption during the waiting period, an `InterruptedException` is caught and the stack trace is printed.

13. Once all tasks have completed, the partial results from each `SumTask` object are summed up in the `sum` variable using a loop.

14. Finally, the total sum is printed to the console.

15. The code also defines a nested static class `SumTask` which implements the `Runnable` interface. Each `SumTask` represents a task to be executed by a thread. It contains an array to process, and the `run` method performs the actual summing of the array elements and stores the result in the `result` variable.

That's an overview of the code. It divides the array into multiple partitions, assigns each partition to a separate thread, and sums the partial results to obtain the total sum using parallel processing.

Own theory :

MPI is an communication protocol to program a parallel processing

**Assignment 4 :**

The provided code implements a simplified version of the Berkeley algorithm for time synchronization. The code combines the server and client functionality into a single program for simplicity. Let's go through the code step by step:

1. `BerkeleyAlgorithm` class:

- This class represents the main class of the program.

- It creates a `ServerSocket` on port 2000 to listen for incoming connections from clients.

- Inside an infinite loop, it accepts client connections and spawns a new thread (`ClientHandler`) to handle each client.

2. `ClientHandler` class:

- This class implements the logic to handle a client request.

- Upon receiving a client connection, it reads the request from the client, which is the client's local time.

- It calculates the current time on the server and sends it back to the client.

- It also calculates the clock difference between the client and the server and establishes a connection to the server itself.

- It sends the clock difference to the server and receives the average clock difference from the server.

- Finally, it adjusts the client's clock by adding the average clock difference and prints the adjusted time.

3. `Client` class:

- This class represents the client part of the program.

- It establishes a connection to the server on localhost and port 2000.

- It retrieves the current time on the client.

- It sends the current time to the server as a request.

- It receives the current time from the server.

- It calculates the clock difference between the server and the client.

- It sends the clock difference to the server.

- It receives the average clock difference from the server.

- It adjusts the client's clock by adding the average clock difference and prints the adjusted time.

- Finally, it closes the connection to the server.

Overall, the code simulates the Berkeley algorithm by exchanging time information between the server and client. The server acts as a central time synchronization point, where it receives time requests from clients, calculates the clock difference, and sends the average clock difference back to the clients. The clients adjust their clocks based on the received clock difference, aiming to achieve time synchronization with the server.

**Assignment 5 :**

This code implements the Token Ring algorithm for process synchronization in a distributed system with N processes. Let's go through the code step by step:

1. The code begins by importing the necessary Java library (`java.util`) for data structures and utilities.

2. The class `TokenRing` is defined, which contains the `main` method where the execution of the program starts.

3. Three constants are defined:

- `N`: Represents the number of processes in the distributed system.

- `TOKEN`: Represents the value of the token. In this case, `-1` is used.

- `CS\_TIME`: Represents the time (in milliseconds) spent in the critical section.

4. Two boolean arrays are declared:

- `hasToken`: Represents whether process i has the token. It is initialized with `N` elements, and all elements are initially set to `false`.

- `inCS`: Represents whether process i is in the critical section. It is also initialized with `N` elements, and all elements are initially set to `false`.

5. A variable `tokenHolder` is declared and initialized with the value `-1`, indicating that no process currently holds the token.

6. The `process` method is defined, which represents the behavior of each process in the system. It takes an `id` parameter to identify the process.

7. Inside the `process` method, there is an infinite loop that simulates the continuous execution of the process.

8. If the process has the token (`hasToken[id]` is `true`), it enters the critical section:

- It sets `inCS[id]` to `true` to indicate that the process is in the critical section.

- It prints a message indicating that the process is entering the critical section.

- It simulates the execution of the critical section by sleeping for `CS\_TIME` milliseconds.

- It prints a message indicating that the process is exiting the critical section.

- It releases the token by setting `hasToken[id]` to `false`.

- It determines the next process to hold the token by calculating `(id + 1) % N`.

- It assigns the token to the next process by setting `hasToken[nextId]` to `true` and updates the `tokenHolder` variable.

9. If the process does not have the token, it waits for it by sleeping for 100 milliseconds.

10. The `main` method is defined, which is the entry point of the program.

11. The token holder is initialized by setting `hasToken[0]` to `true` and `tokenHolder` to `0`, indicating that process 0 initially holds the token.

12. A list of `Thread` objects is created to store the threads representing each process.

13. A loop is used to create and start a thread for each process:

- The `id` variable is assigned the value of `i`.

- A new `Thread` object is created with a lambda expression representing the process behavior.

- The thread is added to the `threads` list.

- The thread is started.

14. Another loop is used to wait for all the processes to finish by calling `join()` on each thread.

The code simulates a distributed system where each process executes the `process` method. Only the process that holds the token is allowed to enter the critical section. The token is passed from one process to another in a circular manner, ensuring that each process gets a chance to enter the critical section. The program runs until all processes have completed execution.

**Assignment 6 a:**

The provided code represents a simplified implementation of the Bully algorithm for leader election in a distributed system. The Bully algorithm is used to elect a coordinator among a group of processes.

Here's a breakdown of the code:

1. Import statements: The code imports necessary classes from the Java standard library.

2. Class declaration: The code defines a class named "Bully."

3. Variables:

- `state`: An array of booleans representing the states of the processes. Each element corresponds to a process, and `state[i]` indicates if process `i + 1` is up or down.

- `coordinator`: An integer representing the current coordinator (process) ID.

4. Method: `up(int up)`:

- This method brings a process up.

- It takes an integer `up` as a parameter, which represents the ID of the process to bring up.

- If the process is already up, it displays a message.

- Otherwise, it sets the state of the process to up and initiates an election by sending election messages to all processes with higher IDs.

- Additionally, it sends an alive message to the highest-ranked active process.

5. Method: `down(int down)`:

- This method brings a process down.

- It takes an integer `down` as a parameter, which represents the ID of the process to bring down.

- If the process is already down, it displays a message.

- Otherwise, it sets the state of the process to down.

6. Method: `mess(int mess)`:

- This method sends a message from a process to elect a coordinator.

- It takes an integer `mess` as a parameter, which represents the ID of the process sending the message.

- If the process is up:

- If the coordinator is process 5, it displays an "OK" message.

- Otherwise, it initiates an election by sending election messages to all processes with higher IDs and sends a coordinator message to the highest-ranked active process.

- If the process is down, it displays a message.

7. `main` method:

- The main method is the entry point of the program.

- It initializes the states of all processes as up.

- It displays information about the active processes and the current coordinator.

- It presents a menu with options to bring a process up, bring a process down, send a message, or exit.

- Based on the user's choice, it invokes the respective methods (`up`, `down`, or `mess`) to perform the corresponding actions.

- The loop continues until the user chooses to exit.

The code allows the user to simulate bringing processes up or down and sending messages for leader election using the Bully algorithm.

**Assignment 6 b:**

The provided code implements the Ring algorithm for leader election in a distributed system. Here's a breakdown of the code:

1. Import statement: The code imports the `Scanner` class from the Java standard library.

2. Class declaration: The code defines a class named "Ring".

3. Class `Rr` declaration: The code defines an inner class named "Rr" to represent a process in the system. It has member variables for the process index, id, state, and f.

4. Variables:

- `temp`, `i`, `j`: Integer variables used for temporary storage and loop iterations.

- `str`: A character array to store characters (not used in the code).

- `proc`: An array of `Rr` objects to represent the processes in the system.

- `in`: An instance of the `Scanner` class used to read input from the console.

- `num`: An integer variable to store the number of processes.

5. Object initialization: The code initializes each element of the `proc` array with a new instance of the `Rr` class.

6. Getting input from users:

- The code prompts the user to enter the number of processes and reads the input into the `num` variable.

- It then iterates over the `proc` array and prompts the user to enter the ID of each process, sets its state as "active," and initializes `f` to 0.

7. Sorting the processes:

- The code sorts the `proc` array based on the process IDs in ascending order.

8. Printing the sorted processes:

- The code iterates over the `proc` array and prints the process index and ID.

9. Variable declarations:

- `init`: An integer variable to store the process number that initiates the election.

- `ch`: An integer variable to store the user's choice (1 for election, 2 to quit).

- `temp1`, `temp2`: Integer variables for temporary storage.

- `ch1`: An integer variable to store a user's input (not used in the code).

- `arr`: An integer array to store the IDs of the processes involved in the election.

10. Setting the last process as inactive:

- The code sets the state of the last process in the `proc` array as "inactive" to simulate the termination of a process.

11. Initiating the election process:

- The code enters an infinite loop that continues until the user chooses to quit.

- Within the loop, the code prompts the user to select a process to initiate the election.

- It then initializes `temp2` and `temp1` with the chosen process number and `init + 1`, respectively.

- Using a while loop, it iterates through the processes in a circular manner, sending messages to active processes with higher IDs.

- The process ID of the message sender is stored in the `arr` array.

- Finally, the process with the highest ID is selected as the coordinator based on the maximum value in the `arr` array.

12. Updating the state of the coordinator:

- The code updates the state of the process with the highest ID as "inactive" to simulate the coordinator election process.

13. Switch case for user choices:

- The code provides options for initiating an election or quitting the program based on the user's choice.

- If the user chooses to quit, the program terminates.

- Otherwise, an error message is displayed for an invalid response.

This code allows the user to simulate the Ring algorithm for leader election by inputting the number and IDs of processes and initiating elections.

**Sure! I'll explain both the Bully algorithm and the Ring algorithm for leader election in a distributed system.**

1. Bully Algorithm:

The Bully algorithm is a leader election algorithm used in distributed systems. It assumes that processes in the system are aware of each other's existence and have unique IDs. Here's how the Bully algorithm works:

- When a process detects that the current leader has failed, it starts an election process by sending an election message to all processes with higher IDs.

- If a process receives an election message, it responds with an OK message to indicate that it's alive and also starts its own election process by sending election messages to processes with higher IDs.

- The process with the highest ID in the system becomes the new leader and sends a coordinator message to inform all other processes of its leadership status.

- If a process doesn't receive an OK message after sending an election message, it assumes that the process with the higher ID is the coordinator and updates its own state accordingly.

The Bully algorithm ensures that the process with the highest ID becomes the leader. However, it has some limitations, such as the need for a central coordinator to initiate the election process and the assumption that processes are aware of each other.

2. Ring Algorithm:

The Ring algorithm is another leader election algorithm used in distributed systems. It assumes that processes are organized in a logical ring, where each process knows only its successor in the ring. Here's how the Ring algorithm works:

- Initially, all processes are active, and one process is designated as the coordinator/leader.

- When a process detects that the coordinator has failed, it starts the election process by sending an election message to its successor.

- The election message travels around the ring, with each process forwarding the message to its successor until it reaches the process that initiated the election.

- If a process receives an election message and has a higher ID than the initiator, it discards the message.

- If a process receives an election message and has a lower ID than the initiator, it becomes the new initiator and forwards the election message.

- The process that initiated the election receives the message again and realizes that it has completed the full ring, indicating that it has the highest ID and becomes the new coordinator/leader.

- The new coordinator sends a coordinator message to inform all other processes of its leadership status.

The Ring algorithm ensures that the process with the highest ID in the ring becomes the leader. It doesn't require a central coordinator and allows processes to elect a new leader when the current one fails. However, it assumes that processes are organized in a ring topology and can communicate only with their immediate neighbors.

Both the Bully algorithm and the Ring algorithm are used to elect a leader in a distributed system, but they employ different strategies and assumptions about process communication and organization.