RMI stands for Remote Method Invocation. It is a Java technology that allows an object running in one Java Virtual Machine (JVM) to invoke methods on an object running in another JVM. RMI is used to build distributed applications; it provides remote communication between Java programs.

To use RMI, you need to create a remote object and register it with an RMI registry. The RMI registry is a server that stores information about remote objects. When a client wants to invoke a method on a remote object, it contacts the RMI registry to get the object's location.

RMI works by using a stub and skeleton. The stub is a proxy object that resides on the client side. When the client calls a method on the stub, the stub marshals the arguments and sends them to the server. The skeleton is a proxy object that resides on the server side. When the skeleton receives the arguments, it unmarshals them and invokes the corresponding method on the remote object. The result of the method invocation is then marshaled and sent back to the client.

Marshaling is the process of converting an object or data structure into a format that can be easily transmitted or stored. Unmarshaling is the opposite process, converting a serialized object or data structure back into its original form.

Assignment 1

AddClient.java

The code you provided is a simple example of how to use RMI to add two numbers. The code first gets a reference to the remote object by calling the `Naming.lookup()` method. The `Naming.lookup()` method takes the URL of the remote object as its argument. In this case, the URL is `rmi://` followed by the hostname of the server where the remote object is running, followed by `/AddServer`.

Once the code has a reference to the remote object, it can invoke its methods. In this case, the code invokes the `add()` method, passing in the two numbers to be added. The `add()` method returns the sum of the two numbers.

The code catches any exceptions that may occur and prints them to the console.

AddServer.java

The code you provided is a simple example of how to create a remote object that can be used to add two numbers. The code first creates a new instance of the `AddServerImpl` class. The `AddServerImpl` class implements the `AddServerIntf` interface, which defines the `add()`

method. The `add()` method takes two numbers as its arguments and returns the sum of the two numbers.

Once the code has created a new instance of the `AddServerImpl` class, it binds the remote object to the name `AddServer`. The `Naming.rebind()` method takes the name of the remote object and a reference to the remote object as its arguments. The `Naming.rebind()` method binds the remote object to the name, so that it can be accessed by other programs.

The code catches any exceptions that may occur and prints them to the console.

AddServerImpl.java

The code you provided is a valid Java RMI server implementation. It extends the `UnicastRemoteObject` class and implements the `AddServerIntf` interface. The `AddServerIntf` interface declares a single method, `add`, which takes two doubles as input and returns their sum. The `AddServerImpl` class implements this method by simply adding the two doubles together.

To use this server, you would need to create a client application that creates a proxy object for the server. The proxy object would then be used to invoke the `add` method on the server.

AddServerIntf.java

The code you provided is a valid Java RMI remote interface. It extends the `Remote` interface and declares a single method, `add`, which takes two doubles as input and returns their sum. The `Remote` interface is a marker interface that indicates that an object can be remotely accessed. The `add` method must throw the `RemoteException` exception, which is a checked exception that is thrown when an error occurs during remote method invocation.

Assignment 2

CORBA provides a way for objects to communicate with each other, even if they are running on different platforms or using different programming languages. CORBA is based on the object-oriented model, and it uses an object request broker (ORB) to mediate communication between objects.CORBA also provides a standard way to define objects, called the Interface Definition Language (IDL). IDL is used to describe the methods and attributes of an object, as well as the types of data that can be passed to and from the object.

ReverseClient.java

The code you provided is a valid Java RMI client application. It imports the `ReverseModule` interface, which defines the `reverse_string` method, and the `org.omg.CosNaming`

package, which provides the `NamingContext` class for looking up remote objects. The `ReverseClient` class creates an `ORB` object, which is used to communicate with the remote server, and then looks up the `Reverse` object in the `NameService`. The `Reverse` object implements the `ReverseModule` interface, which means that it has a `reverse_string` method that can be used to reverse a string. The `ReverseClient` class then calls the `reverse_string` method on the `Reverse` object and prints the result.

To use this client application, you would need to start the RMI server that implements the `ReverseModule` interface. Once the server is started, you can run the `ReverseClient` application and enter a string to be reversed. The application will then print the reversed string.

ReverseImpl.java

The code you provided is a valid Java RMI server implementation. It extends the `ReversePOA` class, which is a generated proxy class for the `ReverseModule` interface. The `ReverseImpl` class implements the `reverse_string` method, which is defined in the `ReverseModule` interface. The `reverse_string` method reverses the string that is passed to it and returns the reversed string.

ReverseModule.idl

The code you provided is a valid Java RMI module definition. It defines a module named `ReverseModule` that contains an interface named `Reverse`. The `Reverse` interface defines a single method, `reverse_string`, which takes a string as input and returns the reversed string.

ReverseServer.java

The `ReverseServer` class is a Java program that implements a CORBA server for the `Reverse` object. The `Reverse` object is a simple object that can reverse the order of a string. The `ReverseServer` class uses the `ORB` (Object Request Broker) to create a CORBA object and bind it to a name in the naming service. Clients can then use the naming service to lookup the `Reverse` object and call its methods.

The `ReverseServer` class has the following steps:

1. Initialize the ORB.
2. Initialize the BOA/POA.

3. Create the `ReverseImpl` object.
4. Get the object reference from the `ReverseImpl` object.
5. Narrow the object reference to a `Reverse` object.
6. Resolve the naming service object.
7. Narrow the naming service object to a `NamingContextExt` object.
8. Bind the `Reverse` object to a name in the naming service.
9. Start the ORB.

Assignment 3

Message Passing Interface (MPI) is a standardized and portable message-passing system developed for distributed and parallel computing. MPI provides parallel hardware vendors with a clearly defined base set of routines

The `ArrSum` class is a Java program that implements a parallel algorithm for summing the elements of an array. The `ArrSum` class uses the MPI (Message Passing Interface) library to communicate between processes.

The `ArrSum` class has the following steps:

1. Initialize MPI.
2. Get the rank and size of the communicator.
3. Set the data for distribution.
4. Scatter the data to processes.
5. Calculate the sum at non-root processes.
6. Gather the data from processes.
7. Aggregate the output from all non-root processes.
8. Finalize MPI.

Assignment 4

1) An individual node is chosen as the master node from a pool node in the network. This node is the main node in the network which acts as a master and the rest of the nodes act as slaves. The master node is chosen using an election process/leader election algorithm.

2) Master node periodically pings slaves nodes and fetches clock time at them using Cristian's algorithm.

3) Master node calculates the average time difference between all the clock times received and the clock time given by the master's system clock itself. This average time difference is added to the current time at the master's system clock and broadcasted over the network.

client.py

The `initiateSlaveClient` function is a Python program that implements a client for synchronizing the clock time between the client and the server. The `initiateSlaveClient` function uses the `socket` library to communicate with the server.

The `initiateSlaveClient` function has the following steps:

1. Create a socket object.
2. Connect to the server.
3. Start a thread to send the current time to the server.
4. Start a thread to receive the synchronized time from the server.
5. Wait for the threads to finish.

The `initiateSlaveClient` function can be used to synchronize the clock time between the client and the server. The `initiateSlaveClient` function is efficient because it uses threads to send and receive data from the server.

server.py

The `initiateClockServer` function is a Python program that implements a clock server. The `initiateClockServer` function uses the `socket` library to communicate with clients.

The `initiateClockServer` function has the following steps:

1. Create a socket object.
2. Bind the socket to a port.
3. Listen for connections.
4. Start a thread to receive clock time from clients.
5. Start a thread to synchronize clocks.
6. Wait for clients to connect.

Assignment 5

Mutual Exclusion ensures that concurrent processes access shared resources or data in a serialized way.

Critical Section: It contains the code that need to be protected from concurrent execution. If a process pi is executing in its critical section than no other processes can be executing in critical section.

Requirements of Mutual exclusion Algorithm:

- **No Deadlock**: Two or more site should not endlessly wait for any message that will never arrive.
- **No Starvation:** Every site who wants to execute critical section should get an opportunity to execute it in finite time. Any site should not wait indefinitely to execute critical section while other site are repeatedly executing critical section
- **Fairness**: Each site should get a fair chance to execute critical section. Any request to execute critical section must be executed in the order they are made i.e Critical section execution requests should be executed in the order of their arrival in the system.
- **Fault Tolerance**: In case of failure, it should be able to recognize it by itself in order to continue functioning without any disruption.

The code you provided implements a simple token passing algorithm for a ring network. The algorithm works as follows:

1. The token is initially held by node 0.
2. When a node has the token, it can send data.
3. When a node has finished sending data, it passes the token to the next node in the ring.
4. If a node does not have any data to send, it passes the token to the next node without doing anything.

1. A unique **token** is shared among all the sites.

5. If a site possesses the unique token, it is allowed to enter its critical section
6. This approach uses sequence number to order requests for the critical section.
7. Each requests for critical section contains a sequence number. This sequence number is used to distinguish old and current requests.
8. This approach insures Mutual exclusion as the token is unique

Assignment 6

Bully.java

The code you provided implements the Bully algorithm for electing a coordinator in a distributed system. The algorithm works as follows:

1. Each process maintains a list of the other processes in the system, along with their process IDs.
2. When a process detects that the current coordinator has failed, it sends an election message to all of the other processes in the system.
3. Each process that receives an election message sends an OK message back to the process that sent the election message.
4. The process with the highest process ID that receives an OK message becomes the new coordinator.
5. The new coordinator sends a coordinator message to all of the other processes in the system.

Ring.java

The code you provided is a Java implementation of the ring election algorithm. The algorithm works as follows:

1. Each process in the ring maintains a list of the processes that are up.
2. When a process fails, the next process in the ring sends a message to all of the processes in its list, asking them to vote for a new coordinator.
3. Each process that receives the message votes for the process with the highest ID in its list.
4. The process with the most votes becomes the new coordinator.

The code you provided implements the following functions:

- `Ring()`: Creates a new ring with the specified number of processes.
- `displayProcesses()`: Displays the list of processes that are up.
- `upProcess()`: Marks a process as up.
- `downProcess()`: Marks a process as down.
- `initElection()`: Initiates an election for a new coordinator.