ASSIGNMENT NO. 01

TITLE: RMI

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

OBJECTIVE: To study:-

- Multi-threading
- How Remote Method Invocation works.
- How RMI allows objects to invoke methods on remote objects.
- How to write Distributed Object Application.

PROBLEM STATEMENT:-

Write a program to implement multi-threaded client/server Process communication using RMI.

TOOLS / ENVIRONMENT:

- **S/W:**
  - Can be developed on any platform Windows /Linux.
  - Java Language, Java Programming Environment, rmiregistry, jdk
- **H/W:**
  - Any basic configuration loaded machine (e.g. P IV)

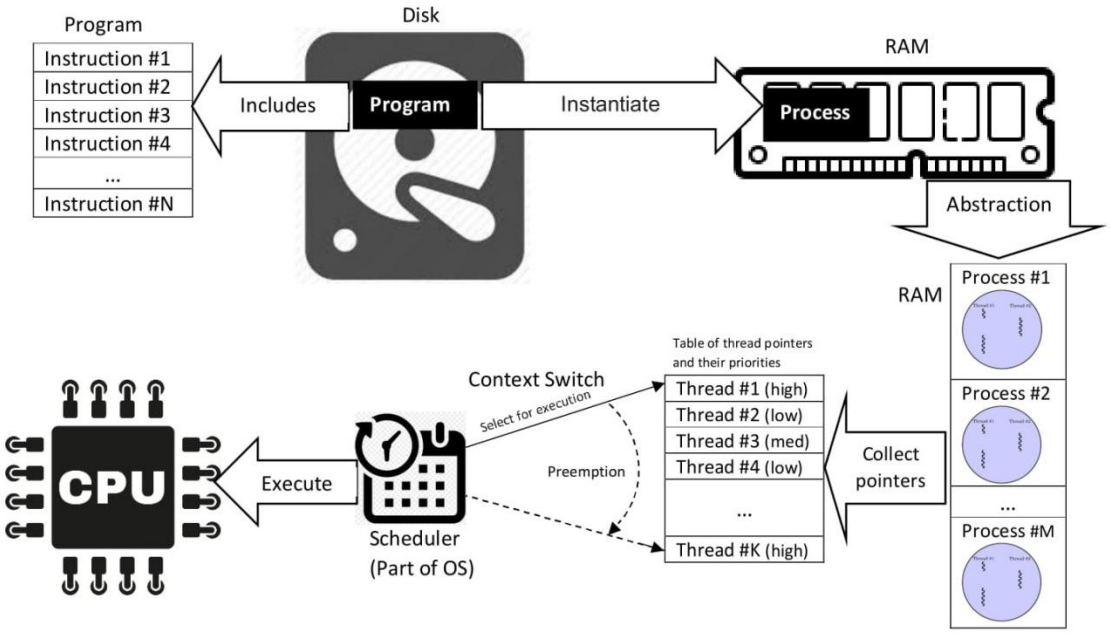
THEORY:

Thread:-

A thread of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system. The implementation of threads and processes differs between operating systems. A process consists of an execution environment together with one or more threads. A thread is the operating system abstraction of an activity (the term derives from the phrase 'thread of execution'). An execution environment is the unit of resource management: a collection of local kernel managed resources to which its threads have access. An execution environment primarily consists of:

- an address space;
- thread synchronization and communication resources such as semaphores and communication interfaces (for example, sockets);
- higher-level resources such as open files and windows.

Threads can be created and destroyed dynamically, as needed. The central aim of having multiple threads of execution is to maximize the degree of concurrent execution between operations, thus enabling the overlap of computation with input and output, and enabling concurrent processing on multiprocessors. This can be particularly helpful within servers, where concurrent processing of clients' requests can reduce the tendency for servers to become bottlenecks.



Multithreading in the Server

RMI automatically will execute each client in a separate thread. There will only be one instance of your server object no matter how many clients, so you need to make sure that shared data is synchronized appropriately. In particular, any data that may be accessed by more than one client at a time must be synchronized.

Also note that you cannot depend on data in the server to remain the same between two successive calls from a client. In our rental car example from the first class, a client may get a list of cars in one call, and in a later call try to reserve the car. In the meantime, another client may have already reserved that car. The server must double-check all data coming from the client to protect against this sort of error. Also for this reason, the server should never pass to the client any direct pointers to its internal data structures (such as an array index), as that type of data is highly likely to change before the client tries to use it.

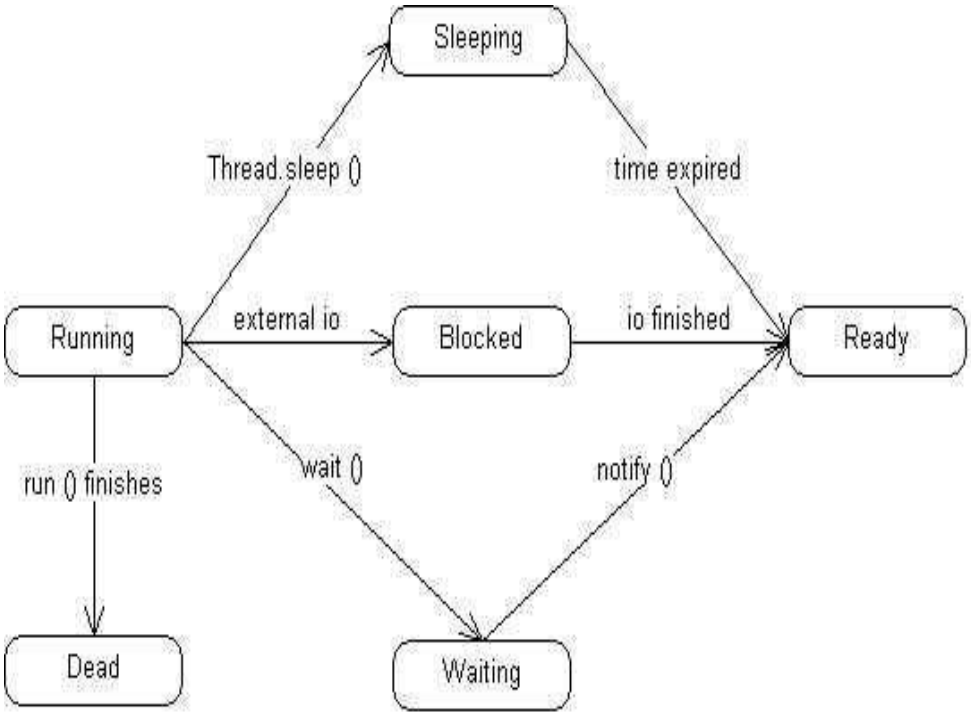
Multithreading can also introduce very difficult to find bugs into your program. The types of bugs introduced because of multithreading are called "race conditions". A race condition is a bug that is timing sensitive. In other words, the bug only happens when several conditions happen at exactly the same time. With multithreading, the possibility of race conditions increases.

Unfortunately, it's almost impossible to know if a multithreading program has bugs or not. Each thread runs at the same time as the other threads. A computer with a single CPU, however, cannot really run multiple instructions at the same time. So it runs instructions from one thread for a while, and then runs some instructions from another thread. You have no way of knowing exactly when a thread may be interrupted and another thread run.

Thread States

A thread can be in one of several states. These include:

- Running -- The thread is currently executing.
- Dead -- The thread has completely finished executing, and will never execute again. This typically means that the client call to the server has completed.
- Ready -- The thread can execute, but is currently not executing. The CPU has switched to another thread and will get back to this one in time.
- Blocked -- A blocked thread is waiting for some external event to finish. Examples would include a thread that is trying to read from a file. The thread will block at the call to read from the file, and not continue executing until the read is finished. In this way, a thread that starts an external task will allow other threads to run until that external task is finished.
- Waiting -- A thread can wait for other threads to finish work it needs to use, by calling the wait () method. A thread that is waiting for other threads to finish some work will not execute until those threads call the notify () method.
- Sleeping -- A thread can voluntarily put itself to sleep for a certain period of time. The thread will start executing again only after the given period of time has passed. The thread puts itself to sleep using the Thread.sleep () method. A thread should put itself to sleep if it only wants to execute every so often.



Java thread constructor and management methods:

- Thread(ThreadGroup group, Runnable target, String name)
  - Creates a new thread in the SUSPENDED state, which will belong to group and be identified as name; the thread will execute the run() method of target.
- setPriority(int newPriority), getPriority()
  - Sets and returns the thread's priority.
- run()
  - A thread executes the run() method of its target object, if it has one, and otherwise its own run() method (Thread implements Runnable).
- start()
  - Changes the state of the thread from SUSPENDED to RUNNABLE.
- sleep(long millisecs)
  - Causes the thread to enter the SUSPENDED state for the specified time.
- yield()
  - Causes the thread to enter the READY state and invokes the scheduler.
- destroy()
  - Destroys the thread.

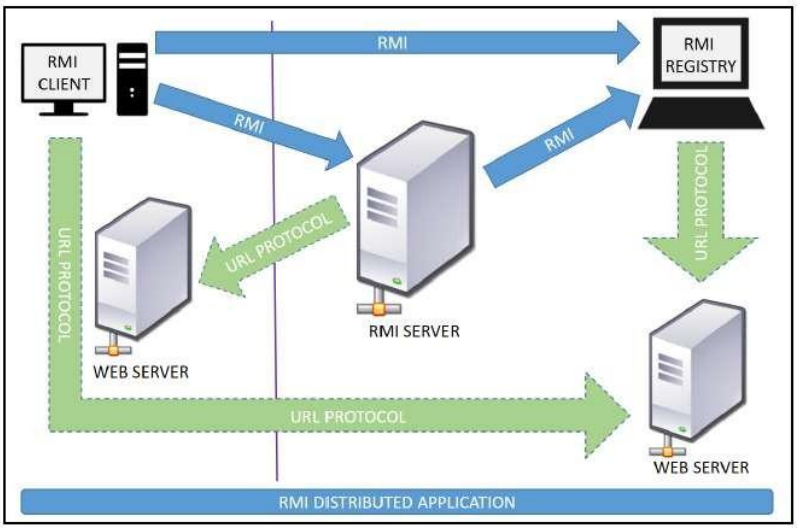
Java thread synchronization calls

- thread.join(long millisecs)
  - Blocks the calling thread for up to the specified time or until thread has terminated.
- thread.interrupt()
  - Interrupts thread: causes it to return from a blocking method call such as sleep().
- object.wait(long millisecs, int nanosecs)
  - Blocks the calling thread until a call made to notify() or notifyAll() on object wakes the thread, the thread is interrupted or the specified time has elapsed.
- object.notify(), object.notifyAll()
  - Wakes, respectively, one or all of any threads that have called wait() on object.

RMI:-

RMI provides communication between java applications that are deployed on different servers and connected remotely using objects called stub and skeleton. This communication architecture makes a distributed application seem like a group of objects communicating across a remote connection. These objects are encapsulated by exposing an interface, which helps access the private state and behavior of an object through its methods.

The following diagram shows how RMI happens between the RMI client and RMI server with the help of the RMI registry:



RMI Registry:

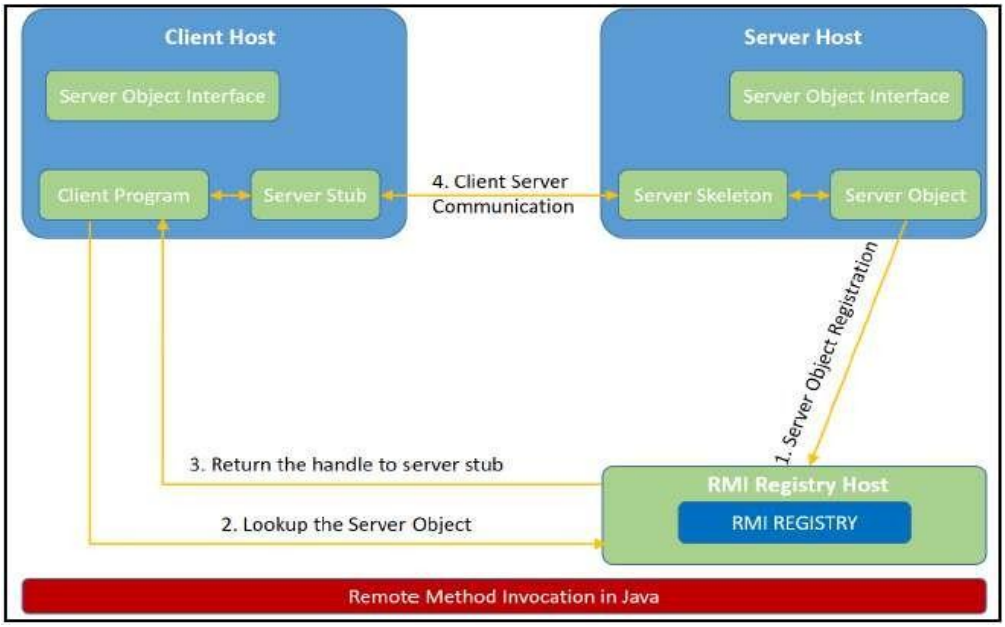
It is a remote object registry, a Bootstrap naming service, that is used by RMI SERVER on the same host to bind remote objects to names. Clients on local and remote hosts then look up the remote objects and make remote method invocations.

Key terminologies of RMI:

The following are some of the important terminologies used in a Remote Method Invocation.

- **Remote object:** This is an object in a specific JVM whose methods are exposed so they could be invoked by another program deployed on a different JVM.
- **Remote interface:** This is a Java interface that defines the methods that exist in a remote object. A remote object can implement more than one remote interface to adopt multiple remote interface behaviors.
- **RMI:** This is a way of invoking a remote object's methods with the help of a remote interface. It can be carried with a syntax that is similar to the local method invocation.
- **Stub:** This is a Java object that acts as an entry point for the client object to route any outgoing requests. It exists on the client JVM and represents the handle to the remote object.
- If any object invokes a method on the stub object, the stub establishes RMI by following these steps:
    1. It initiates a connection to the remote machine JVM.
    2. It marshals (write and transmit) the parameters passed to it via the remote JVM.
    3. It waits for a response from the remote object and unmarshals (read) the returned value or exception, then it responds to the caller with that value or exception.
- **Skeleton:** This is an object that behaves like a gateway on the server side. It acts as a remote object with which the client objects interact through the stub. This means that any requests coming from the remote client are routed through it. If the skeleton receives a request, it establishes RMI through these steps:
    1. It reads the parameter sent to the remote method.
    2. It invokes the actual remote object method.
    3. It marshals (writes and transmits) the result back to the caller (stub).

The following diagram demonstrates RMI communication with stub and skeleton involved:



Designing the solution:

The essential steps that need to be followed to develop a distributed application with RMI are as follows:

1. Design and implement a component that should not only be involved in the distributed application, but also the local components.
2. Ensure that the components that participate in the RMI calls are accessible across networks.
3. Establish a network connection between applications that need to interact using the RMI.

1. **Remote interface definition:** The purpose of defining a remote interface is to declare the methods that should be available for invocation by a remote client. Programming the interface instead of programming the component implementation is an essential design principle adopted by all modern Java frameworks, including Spring. In the same pattern, the definition of a remote interface takes importance in RMI design as well.
2. **Remote object implementation:** Java allows a class to implement more than one interface at a time. This helps remote objects implement one or more remote interfaces. The remote object class may have to implement other local interfaces and methods that it is responsible for. Avoid adding complexity to this scenario, in terms of how the arguments or return parameter values of such component methods should be written.
3. **Remote client implementation:** Client objects that interact with remote server objects can be written once the remote interfaces are carefully defined even after the remote objects are deployed.

Let's design a project that can sit on a server. After that different client projects interact with this project to pass the parameters and get the computation on the remote object execute and return the result to the client components.

Implementing the solution:
Consider building an application to perform diverse mathematical operations.
The server receives a request from a client, processes it, and returns a result. In this example, the request specifies two numbers. The server adds these together and returns the sum.

1. Creating remote interface, implement remote interface, server-side and client-side program and Compile the code.

This application uses four source files. The first file, `AddServerIntf.java`, defines the remote interface that is provided by the server. It contains one method that accepts two double arguments and returns their sum. All remote interfaces must extend the Remote interface, which is part of `java.rmi`. Remote defines no members. Its purpose is simply to indicate that an interface uses remote methods. All remote methods can throw a `RemoteException`.
The second source file, `AddServerImpl.java`, implements the remote interface. The implementation of the `add()` method is straightforward. All remote objects must extend `UnicastRemoteObject`, which provides functionality that is needed to make objects available from remote machines.
The third source file, `AddServer.java`, contains the main program for the server machine. Its primary function is to update the RMI registry on that machine. This is done by using the `rebind()` method of the Naming class (found in `java.rmi`). That method associates a name with an object reference. The first argument to the `rebind()` method is a string that names the server as "`AddServer`". Its second argument is a reference to an instance of `AddServerImpl`.
The fourth source file, `AddClient.java`, implements the client side of this distributed application. `AddClient.java` requires three command-line arguments. The first is the IP address or name of the server machine. The second and third arguments are the two numbers that are to be summed.

The application begins by forming a string that follows the URL syntax. This URL uses the rmi protocol. The string includes the IP address or name of the server and the string "`AddServer`". The program then invokes the `lookup( )` method of the Naming class. This method accepts one argument, the rmi URL, and returns a reference to an object of type `AddServerIntf`. All remote method invocations can then be directed to this object.
The program continues by displaying its arguments and then invokes the remote `add()` method. The sum is returned from this method and is then printed.
Use `javac` to compile the four source files that are created.

2. Generate a Stub

Before using client and server, the necessary stub must be generated. In the context of RMI, a stub is a Java object that resides on the client machine. Its function is to present the same interfaces as the remote server. Remote method calls initiated by the client are actually directed to the stub. The stub works with the other parts of the RMI system to formulate a request that is sent to the remote machine.

All of this information must be sent to the remote machine. That is, an object passed as an argument to a remote method call must be serialized and sent to the remote machine. If a

response must be returned to the client, the process works in reverse. The serialization and deserialization facilities are also used if objects are returned to a client.
To generate a stub the command is `RMIcompiler` is invoked as follows:

```
rmic AddServerImpl.
```

This command generates the file `AddServerImpl_Stub.class`.

3. Install Files on the Client and Server Machines

Copy `AddClient.class`, `AddServerImpl_Stub.class`, `AddServerIntf.class` to a directory on the client machine.
Copy `AddServerIntf.class`, `AddServerImpl.class`, `AddServerImpl_Stub.class`, and `AddServer.class` to a directory on the server machine.

4. Start the RMI Registry on the Server Machine

Java provides a program called `rmiregistry`, which executes on the server machine. It maps names to object references. Start the RMI Registry from the command line, as shown here:

```
start rmiregistry
```

5. Start the Server

The server code is started from the command line, as shown here: `java AddServer`

The `AddServer` code instantiates `AddServerImpl` and registers that object with the name "`AddServer`".

6. Start the Client

The `AddClient` software requires three arguments: the name or IP address of the server machine and the two numbers that are to be summed together. You may invoke it from the command line by using one of the two formats shown here:

```
java AddClient 192.168.13.14 7 8
```

Writing the source code:

- Create the remote interface (SumServerInfr.java)
- Provide the implementation of the remote interface (SumServerImpl.java)
- Create the remote application (SumServer.java)
- Create the client application (SumClient.java)

(Students should write here the implementation for their program. Students should attach printout of their programs with commands used to run the programs. Also attach the proper outputs of programs.)

Compilation and executing the solution:

- Compile the implementation class (java *.java)
- Create the stub and skeleton objects using the rmic tool (rmic SumServerImpl)
- Start the registry service by rmiregistry tool (rmiregistry)
- Start the remote application in new terminal (java SumServer)
- Start the client application in new terminal (java SumClient)

Conclusion:

Remote Method Invocation (RMI) allows you to build Java applications that are distributed among several machines. Remote Method Invocation (RMI) allows a Java object that executes on one machine to invoke a method of a Java object that executes on another machine. This is an important feature, because it allows you to build distributed applications.

REFERENCES: "Distributed Systems: Concepts and Design" by George Coulouris, J Dollimore and Tim Kindberg, Pearson Education, ISBN: 9789332575226, 5th Edition, 2017.

FAQ:

1. What is thread? What is difference between thread and process?
2. What are types of thread?
3. What is RMI? Explain the RMI architecture ?
4. What is a remote object ? Why should we extend UnicastRemoteObject ? What is Unicast and Multicast object?
5. What are the services provided by the RMI Object ?
6. What are the differences between RMI and a socket ?
7. How will you pass parameters in RMI ?
8. What is HTTP tunneling or how do you make RMI calls across firewalls ?
9. Why use RMI when we can achieve the same benefits from EJB ?
10. How many types of protocol implementations does RMI have?
11. Can RMI and Corba based applications interact ?
12. Explain the difference between RPC and RMI.
13. What is the difference between RMI and JMS?
14. What is Registry Service for RMI?
15. Explain how URL convention is used for accessing the registry.
16. Explain how to bind an object to the registry.
17. Explain the various methods of registering and gaining access to the Remote Object.
18. What are Remote callbacks?

19. What is Object Activation?
20. What is data transfer in RMI model.
21. What is object serialization in RMI?
22. What is RMI callback mechanism?
23. What is the role of Remote Interface in RMI?
24. Explain marshalling and demarshalling.
25. What is a skeleton in RMI? Explain the role of stub in RMI.
26. What are the layers on which RMI implementation is built? Explain them.
27. Explain how RMI clients contact remote RMI servers.
28. What are the basic steps to write client-service application using RMI?
29. Default port used by RMI Registry?
30. What is the difference between using bind() and rebind() methods of Naming Class?