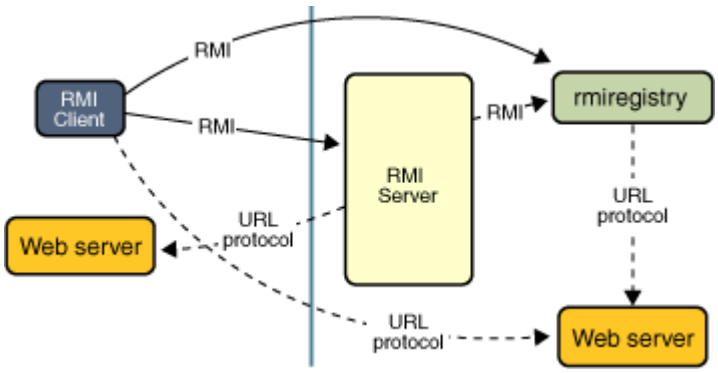
Remote Method Invocation (RMI)

RMI applications often comprise two separate programs, a **Server** and a **Client**. A typical server program creates some **remote objects**, makes references to these objects accessible, and waits for clients to invoke methods on these objects. A typical client program obtains a **remote reference** to one or more remote objects on a server and then invokes methods on them. RMI provides the mechanism by which the server and the client communicate and pass information back and forth. Such an application is sometimes referred to as a **Distributed Object Application**.

RMI is implemented only through Java which satisfies the concept "**write once, run any where**" paradigm. RMI uses a registry server for registering and locating objects. RMI allows Java programs to register their class methods with the server. Once this has been set up, sending messages, or invoking methods on remote processes, is as simple as invoking methods on local objects. An object is said to be distributed, if it is available on the network.

An RMI distributed application uses the **RMI registry** to obtain a reference to a remote object. The **Server** calls the registry to associate (or **bind**) a name with a remote object. The **Client looks up** the remote object by its name in the server's registry and then invokes a method on it.



RMI registry keeps track of the addresses of remote objects that are being exported. All the objects are assigned unique names that are used to identify them. The method **bind()** is used to bind an object in the registry. If there is an object with the same name, then this method raises **AlreadyBoundException**. To avoid this, we can use **rebind()** method to replace the earlier one.

Advantages of Dynamic Code Loading

One of the central and unique features of RMI is its ability to download the definition of an object's class if the class is not defined in the receiver's Java virtual machine. All of the types and behavior of an object, previously available only in a single Java virtual machine can be transmitted to another, possibly remote, Java virtual machine. RMI passes objects by their actual classes, so the behavior of the objects is not changed when they are sent to another Java virtual machine. This capability enables new types and behaviors to be introduced into a remote Java virtual machine, thus dynamically extending the behavior of an application.

Remote Interfaces, Objects, and Methods

Like any other Java application, a distributed application built by using Java RMI is made up of interfaces and classes. The interfaces declare methods. The classes implement the methods declared in the interfaces. Objects with methods that can be invoked across Java Virtual Machines are called remote objects.

An object becomes remote by implementing a **Remote** interface. RMI treats a remote object differently from a non-remote object when the object is passed from one Java Virtual Machine to another Java Virtual Machine. Rather than making a copy of the implementation object in the receiving Java Virtual Machine, RMI passes a remote **stub** for a remote object. The stub acts as the local representative for the remote object. The client invokes a method on the local stub, which is responsible for carrying out the method invocation on the remote object.

A stub for a remote object implements the same set of remote interfaces that the remote object implements. This property enables a stub to be cast to any of the interfaces that the remote object implements. However, only those methods defined in a remote interface are available to be called from the receiving Java Virtual Machine.

The package **java.rmi** contains classes, interfaces and sub packages that can help in creating remote objects, access the remote methods of the objects.

Designing and Implementing the Application Components

First, determine your application architecture, including which components are local objects and which components are remotely accessible.

1. Defining the Remote Interface:

A **Remote Interface** specifies the methods that can be invoked remotely by a client. The design of such interfaces includes the determination of the types of objects that will be used as the parameters and return values for these methods. Your interface should implement the **java.rmi.Remote** interface.

2. Implementing the Remote Interface.

The methods of the Remote interface must be implemented by a Remote object class.

3. Create Remote Objects in the RMI Registry:

Remote objects must implement one or more remote interfaces. The remote object class may include implementations of other interfaces and methods that are available only locally. If any local classes are to be used for parameters or return values of any of these methods, they must be implemented as well.

4. Invoke the methods of Remote Object:

Clients that use remote objects can be implemented at any time after the remote interfaces are defined, including after the remote objects have been deployed.

RMI Application should include 4 blocks of program

1. Interface	contains only the prototype for methods
2. Implementation	implements the methods of the interface
3. Server	creates an object for implementation class
4. Client	uses the remote object created by the server

In the **Server** program, **Naming.rebind()** method is used to bind the object at **"rmiregistry"** location with a specified object name. The **Client** program uses **Naming.lookup()** method to access the methods of the remote object created by the Server.

Procedure to execute an RMI Application

1. Create Class Files (Compile all the 4 blocks of programs).
2. Generate **Stub** and **Skeleton** for the Implementation class with the help of rmic utility as follows.
   rmic -vcompat Implementation-File-Name
3. Create two directories named Client and Server.
4. Copy the following files into **Client** directory.
   - Interface Class File
   - Implementation **Stub** Class File
   - Client Class File
5. Copy the following files into **Server** Directory.
   - Interface Class File
   - Implementation Class File
   - Implementation **Skeleton** Class File
   - Server Class File
6. Start the **rmiregistry**.
7. Run the Server program.
8. Run the Client program.