Java RMI

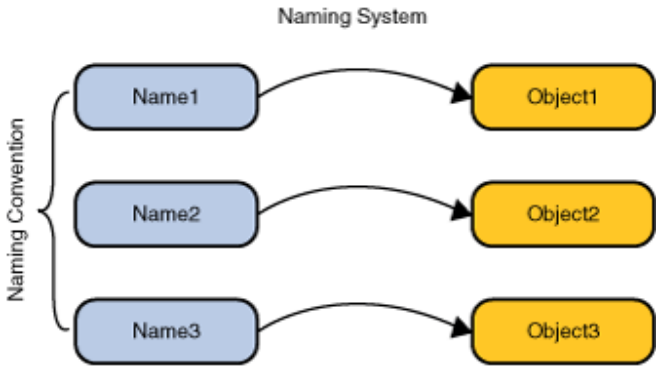
Introduction

One of the methods of communication between processes is Remote Procedure Call, in which a sub-routine can be executed in another address space as if it was executed locally. In Java, the object-oriented implementation of this paradigm is Remote Method Invocation. RMI allows objects in different virtual machines to communicate using normal method calls.

The server makes one or more objects available remotely. The client then calls the methods of these objects to get the results.

The remote objects must implement an interface that specifies which methods can be accessed remotely. This is done by writing an interface that extends Remote interface. (Remote interface does not define any methods, it is a Marker Interface). Each method of the interface declares java.rmi.RemoteException in its throws clause, in addition to any application-specific exceptions. The class of the remote object must extend RemoteObject and implement the previously defined interface.

In order for the client to find the object, the object must be registered in a naming registry.



A naming service allows you to look up an object given its name. The association of a name with an object is called a binding.

When the class corresponding to the remote object is compiled two additional classes are created:

- Stub : the class that translates method calls to be sent over the network.
- Skeleton: the class that accepts the incoming connections and translates them in method calls.

In effect the client calls the method on the stub which sends the call to the skeleton, taking care of all communication issues. The skeleton calls the actual method and then sends back the output data to the stub.

Before Java 5.0 you had to use an additional compiler to create the stubs and skeletons, rmic.

Let`s create a simple application: The server exposes RemoteObject that holds a number. It exposes two methods : getResult() which shows the value of the number and add(int no) which adds no to the number.

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface MyRemoteInterface extends Remote{

	public int getResult () throws RemoteException;
	public void add(int d) throws RemoteException;
}

import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class MyRemote extends UnicastRemoteObject implements
MyRemoteInterface{
int sum=0;
	protected MyRemote() throws RemoteException {
		super();
	}
	public void add(int d) throws RemoteException {
		sum+=d;
	}
	public int getResult() throws RemoteException {
		return sum;
	}
}

import java.net.MalformedURLException;
import java.rmi.AlreadyBoundException;
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;


public class ServerMain {

	public static void main(String[] args) throws RemoteException,
MalformedURLException, AlreadyBoundException {

	MyRemote myr=new MyRemote();
	Registry registry=LocateRegistry.createRegistry(2500);
	registry.bind("object", myr);
	}
}
```

```
import java.rmi.NotBoundException;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class Main2 {

	public static void main(String[] args) throws RemoteException,
NotBoundException, InterruptedException {

Registry registry = LocateRegistry.getRegistry(7500);
for(String s : registry.list()){
	System.out.println(s);
}
		MyRemoteInterface remote = (MyRemoteInterface)
registry.lookup("object");
		for(int i=0;i<10;i++){
		remote.add(15);
		System.out.print(remote.getResult());
		Thread.sleep(1500);
		}
	}
}
```

Arguments to or return values from remote methods can be of almost any type, including local objects, remote objects, and primitive data types. More precisely, any entity of any type can be passed to or from a remote method as long as the entity is an instance of a type that is a primitive data type, a remote object, or a serializable object, which means that it implements the interface java.io.Serializable.

Exercise

Create a distributed application using RMI for buying tickets at the Opera. The client application is a Java desktop application that allows seeing the available places and buying a ticket for a place. The client connects to a server application which has access to the ticket database.

1) Use primitive types for the parameters of the exposed methods. (Seat and row number).
2) Create a class Seat and pass objects of this class between the client and server application.