Client Server Application Programming

Week 10: Remote Method Invocation(Overview, How Remote Method Invocation Work, Defining an RMI Service Interface, Implementing an RMI Service etc

By Elubu Joseph - MSc.IS

Lecturer

Department of Information Technology

Kumi University

Email: josebulinda@gmail.com

jose@kumiuniversity.ac.ug

Summary of Previous Lecture

1. Java Servelets Overview
2. How Servlets Work,
3. Using Servlets,
4. Writing a Simple Servlet and Running it,
5. Single Thread Model,
6. Servlet Request and http Servlet Request,
7. Servlet Response and http Response)

Agenda

1. Overview of Remote Method Invocation
2. How Remote Method Invocation Work,
3. Defining a RMI Service Interface,
4. Implementing an RMI Service Interface,
5. Creating Stub and Skeleton Classes

Overview of Remote Method Invocation

Overview of Remote Method Invocation

Remote Method Invocation (RMI) is a Java-based technology that allows distributed systems to communicate and interact with each other. It allows one Java Virtual Machine (JVM) to invoke object methods that will be run on another JVM located elsewhere on a network.

This technology is extremely important for the development of large-scale systems, as it makes it possible to distribute resources and processing load across more than one machine.

Overview of Remote Method Invocation +

RMI provides a seamless and transparent mechanism for remote procedure calls (RPCs) within a Java environment.

Here's an overview of how RMI works:

1. **Interface Definition:** RMI starts with the definition of a remote interface. This interface specifies the methods that can be invoked remotely by clients. Both the client and server must have access to this interface and its implementation.
2. **Remote Object Implementation:** On the server side, a class is implemented that provides the actual implementation of the remote interface. This class is referred to as the **"remote object"** and must extend the java.rmi.server.UnicastRemoteObject class, which provides the necessary functionality for remote invocation.

Overview of Remote Method Invocation ++

3. **Server Setup:** The server program instantiates the remote object and registers it with the RMI runtime using the java.rmi.Naming class or the java.rmi.registry.Registry class. This step allows clients to locate the remote object by a unique name or address.

4. **Client Lookup:** The client program uses the java.rmi.Naming class or the java.rmi.registry.Registry class to look up the remote object on the server. It obtains a reference to the remote object, which can then be used to invoke methods remotely.

5. **Method Invocation:** Once the client has obtained a reference to the remote object, it can invoke methods on the remote object as if it were a local object. The RMI system takes care of marshalling/ordering the method parameters, sending them over the network to the server, executing the method on the server, and returning the result back to the client.

Overview of Remote Method Invocation +++

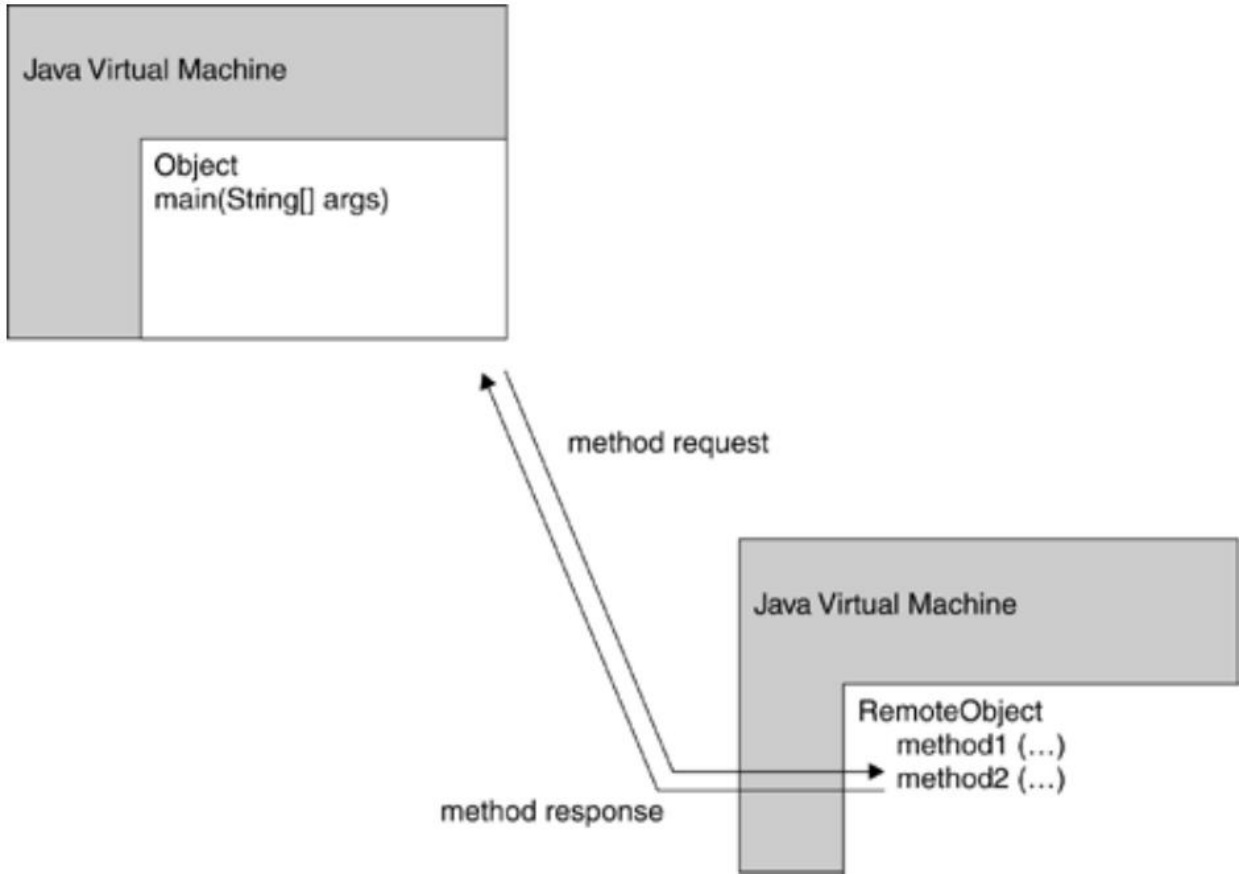
6. **Parameter and Result Marshalling:** RMI automatically serializes the method parameters and return values, allowing them to be transmitted across the network. The serialization and deserialization process is handled by the RMI system, providing a transparent mechanism for remote communication.

7. **Exceptions and Callbacks:** RMI supports exceptions during method invocations, allowing the server to propagate errors back to the client. RMI also supports callbacks, where the server can invoke methods on objects registered by the client. This enables bidirectional communication between the client and server.

RMI provides a high-level abstraction for remote communication in Java, making it easier to develop distributed applications. It abstracts the complexities of network communication and allows developers to focus on the application logic.

Overview of Remote Method Invocation +++

The **Figure below** provides an example of this process, whereby an object running on one JVM invokes a method of an object hosted by another.



Difference between Remote Method Invocation and Remote Procedure Calls

Even before object-oriented programming, technologies existed that allowed software to call functions and procedures remotely. Systems such as remote procedure calls (RPCs) have been in use for years and continue to be used today. A popular implementation of RPC was developed by Sun Microsystems and published as RFC 1057 (making obsolete an earlier version, published as RFC 1050).

Remote procedure calls were designed as a platform-neutral way of communicating between applications, regardless of any operating system or language differences.

The **principal difference between** the two goals is that **RPC supports multiple languages**, whereas **RMI only supports applications written in Java only**.

How Remote Method Invocation Work

How Remote Method Invocation Work

Systems that use RMI for communication typically are divided into two categories: clients and servers. A server provides a RMI service, and a client invokes object methods of this service.

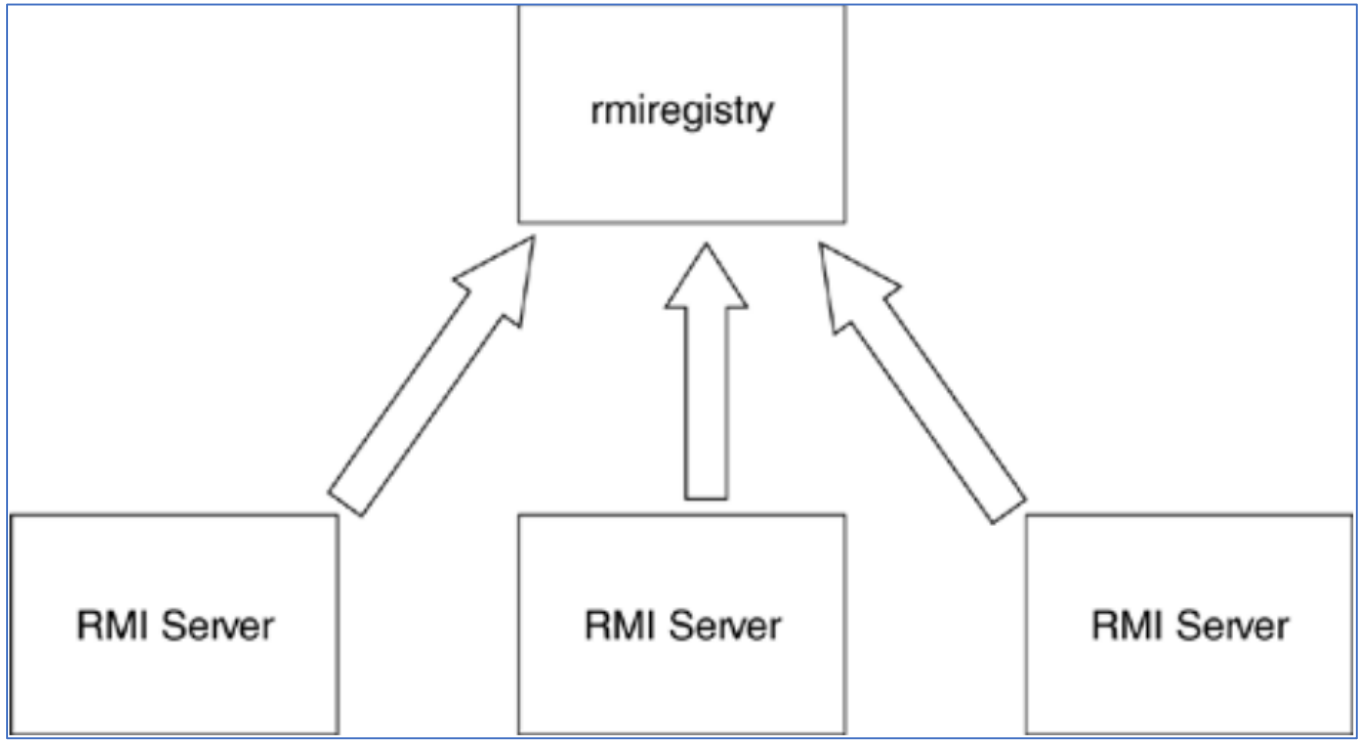
RMI servers must register with a lookup service, to allow clients to find them, or they can make available a reference to the service in some other fashion.

Included as part of the Java platform is an application called **rmiregistry**, which runs as a separate process and allows applications to **register** RMI services or obtain a reference to a named service.

Once a server has registered, it will then wait for incoming RMI requests from clients.

How Remote Method Invocation Work + Multiple services registered with the same registry

The Figure below illustrates services registering with a single RMI registry.



RMI registry contains all the services provided by various servers on the network.

Before any remote method invocation can occur, the client must have a remote object reference from the RMI registry.

How Remote Method Invocation Work +Multiple services registered with the same registry+

Associated with each service registration is a name (represented as a string), to allow clients to select the appropriate service.

If a service moves from one server to another, the client need only look up the registry again to find the new location. This makes for a more fault tolerant system—if the service is unavailable because a machine is down, a system administrator could launch a new instance of the service on another system and have it register with the RMI registry.

Provided the registry remains active, you can have your servers go online and offline or move from host to host. The registry doesn't care which host a service is offered from, and clients get the service location directly from the registry.

How Remote Method Invocation Work+++

Remember, URLs are not just for HTTP—most protocols can be represented using URL syntax. The following format is used by RMI for representing a remote object reference: `rmi://hostname:port/servicename`

where hostname represents the name of a server (or IP address), port the location of the service on that machine, and servicename a string description of the service.

Once an object reference is obtained (either through the rmiregistry, a custom lookup service, or by reading an object reference URL from a file), the client can then interact with the remote service.

The networking details of requests are completely transparent to the application developer—working with remote objects becomes as simple as working with local ones. This is achieved through a clever division of the RMI system into two components, **the stub** and **the skeleton**.

How Remote Method Invocation Work++++

The stub object acts as a proxy object, conveying object requests to the remote RMI server. Remember that every RMI service is defined as an interface, not as an implementation.

The stub object implements a particular RMI interface, which the client application can use just like any other object implementation. Rather than performing the work itself, however, the **stub passes a message** to a remote RMI service, **waits for a response**, and **returns this response** to the calling method.

In this case, the application developer doesn't need to be concerned about where the RMI resource is located, and on which platform it is running, or how it will fulfill the request.

The RMI client simply invokes a method of the proxy object, which handles all the implementation details. The Figure below illustrates how this is achieved; shown is an RMI client invoking an object method on the stub proxy, which conveys this request to the remote server.

How Remote Method Invocation Work++++The RMI client stub calls the RMI server skeleton

At the RMI server end, the skeleton object is responsible for listening for incoming RMI requests and passing these on to the RMI service. The skeleton object does not provide an implementation of an RMI service, however. It only acts as a receiver for requests, and passes these requests on further

Defining an RMI Service Interface

Defining an RMI Service Interface

Any system that uses RMI will use a service interface. The service interface defines the object methods that can be invoked remotely, and specifies parameters, return types, and exceptions that may be thrown. Stub and skeleton objects, as well as the RMI service, will implement this interface.

For this reason, developers are urged to define all methods in advance, and to freeze changes to the interface once development begins.

Can changes be made on interfaces?

Yes. It is possible to make changes to an interface, **but** all clients and servers must have a new copy of the service interface, and code for the stubs and skeletons must be rebuilt.

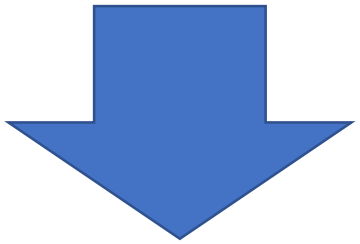
Defining an RMI Service Interface+

All RMI service interfaces extend the **java.rmi.Remote** interface, which assists in identifying methods that may be executed remotely.

To define a new interface for an RMI system, you must declare a new interface extending the Remote interface.

Note! Only methods defined in a java.rmi.Remote interface (or its subclasses) may be executed remotely—other methods of an object are hidden from RMI clients.

For example, to define an interface for a remote lightbulb system (a high-tech version of the traditional on-off switch for a networked world), we could define an interface such as the following:



Defining an RMI Service Interface++RMILightBulb interface

```
import java.rmi.*;
public interface RMILightBulb extends Remote {
    public void on () throws RemoteException;
    public void off() throws RemoteException;
    public boolean isOn() throws RemoteException;
 }
```

```
2   package cs;
3   import java.rmi.RemoteException;
4   import java.rmi.Remote;
    public interface RMILightBulb extends Remote{
            public void on () throws RemoteException;
            public void off() throws RemoteException;
            public boolean isOn() throws RemoteException;
9    }
```

The interface is identified as remotely accessible, by extending from the Remote interface. Each method is marked as public, and may throw a java. **rmi.RemoteException**. This is important, as network errors might occur that will prevent the request from being issued or responded to.

Defining an RMI Service Interface++RMILightBulb interface+

In an RMI client, a stub object that implements this interface will act as a proxy to the remote system—if the system is down, the stub will throw a RemoteException error that must be caught.

If a method is defined as part of an RMI interface, it must be marked as able to throw a `RemoteException`—if it is not, `stub` and `skeleton` classes cannot be generated by the `"rmic" tool` (a tool that ships with the Java SDK which automates the generation of these classes). Methods are not limited to throwing only a `RemoteException`, however.

They may throw additional exceptions that are already defined as part of the Java API (such as an `IllegalArgumentException` to indicate bad method parameters), or custom exceptions created for a system.

For example, the `on()` method could be modified to throw a `BrokenBulb` exception, if it could not be successfully activated.

Implementing an RMI Service Interface

Once a service interface is defined, the next step is to implement it. This implementation will provide the functionality for each of the methods, and may also define additional methods.

However, only those methods defined by the RMI service interface will be accessible remotely, even if they are marked public in scope or as being able to throw a RemoteException.

For our light bulb system, the following implementation could be created.

Implementing an RMI Service Interface+RMILightBulbImpl in one place

```
1  package cs; import java.rmi.RemoteException;
2  import java.rmi.server.UnicastRemoteObject;
3  public class RMILightBulbImpl extends UnicastRemoteObject implements
4          RMILightBulb {
5   // A constructor must be provided for the remote object
6      public RMILightBulbImpl() throws RemoteException {...4 lines }
10     // Boolean flag to maintain light bulb state information
11     private boolean lightOn;
12     // Remotely accessible "on" method - turns on the light
       public void on() throws RemoteException {...4 lines }
17     // Remotely accessible "off" method - turns off the light
       public void off() throws RemoteException{...4 lines }
22     // Remotely accessible "isOn" method, returns state of bulb
       public boolean isOn() throws RemoteException{...3 lines }
26
27     // Locally accessible "setBulb" method, changes state of bulb
28     public void setBulb (boolean value){...3 lines }
31     // Locally accessible "getBulb" method, returns state of bulb
32     public boolean getBulb (){...3 lines }
35 }
```

Remember that a class **extends** a class, a class implements an interface, meanwhile an interface extends an interface.

Implementing an RMI Service Interface+RMILightBulbImpl class

Lets expand RMILightbulbImpl class methods

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
// Extends UnicastRemoteObject class for remote object functionality
public RMILightBulbImpl extends UnicastRemoteObject implements
RMILightBulb {
	// A constructor must be provided for the remote object
	public RMILightBulbImpl() throws RemoteException {
		// Default value of off
		setBulb(false);
	}
```

Implementing an RMI Service Interface+RMILightBulbImpl class

```
// Boolean flag to maintain light bulb state information
      private boolean lightOn;
// Remotely accessible "on" method - turns on the light
public void on() throws RemoteException {
      // Turn bulb on
      setBulb (true);
}
// Remotely accessible "off" method - turns off the light
public void off() throws RemoteException {
      // Turn bulb off
      setBulb (false);
}
```

Implementing an RMI Service Interface+RMILightBulbImpl class

```
// Remotely accessible "isOn" method, returns state of bulb
public boolean isOn() throws RemoteException {
	return getBulb();
}
// Locally accessible "setBulb" method, changes state of bulb
public void setBulb(boolean value){
	lightOn = value;
}
// Locally accessible "getBulb" method, returns state of bulb
public boolean getBulb() {
	return lightOn;
}
}//End of code
```

Implementing an RMI Service Interface+RMILightBulbImpl code explained

The code above demonstrates the implementation of a remote light bulb interface using Java RMI (Remote Method Invocation). It defines a class called **`RMILightBulbImpl`** that extends **`UnicastRemoteObject`** and implements the **`RMILightBulb`** interface. The **`RMILightBulb`** interface specifies the methods that can be invoked remotely on the light bulb.

The **`RMILightBulbImpl`** class has a constructor that sets the initial state of the light bulb to off (false). It also includes a private **boolean** variable **lightOn** to track the state of the light bulb.

The class provides three remote methods: **`on()`**, **`off()`**, and **`isOn()`**. The **`on()`** method turns on the light bulb by setting **`lightOn`** to true. The **`off()`** method turns off the light bulb by setting **`lightOn`** to false. The **`isOn()`** method returns the current state of the light bulb.

Implementing an RMI Service
Interface+RMILightBulbImpl code explained+

Additionally, there are two local methods: `setBulb(boolean value)` and `getBulb()`. These methods are used internally within the class to set and retrieve the state of the light bulb.

The code demonstrates the basic structure of an RMI implementation, where objects can be accessed remotely and their methods can be invoked as if they were local objects. In this case, the `RMILightBulbImpl` class represents a remote light bulb that can be controlled by clients through RMI.

Implementing an RMI Service
Interface+RMILightBulbImpl code explained++

The above code represents the implementation of the remote light bulb on the server side. It includes the RMILightBulbImpl class, which extends UnicastRemoteObject and implements the RMILightBulb interface. This class defines the behavior and functionality of the remote light bulb object.

On the client side, there would typically be a separate program that interacts with the remote light bulb object by obtaining a reference to it and invoking its methods. However, the code you provided does not include the client-side code.

Creating Stub and Skeleton Classes

Creating Stub and Skeleton Classes

The `stub` and `skeleton` classes are responsible for dispatching and processing RMI requests. Developers should not write these classes, however. Once a service implementation exists, the `rmic tool`, which ships with the JDK, should be used to create them.

The implementation of an interface should be compiled, and then the following typed at the command line:

```
rmic implementation
```

where `implementation` is the name of the service implementation class.

Creating Stub and Skeleton Classes

For example, if the class files for the `RMILightBulb` system are in the current directory, the following would be typed to produce stub and skeleton classes:

```
rmic RMILightBulbImpl
```

Two files would then be produced in this case:

```
1.RMILightBulbImpl_Stub.class
2.RMILightBulbImpl_Skeleton.class
```

Creating Stub and Skeleton Classes

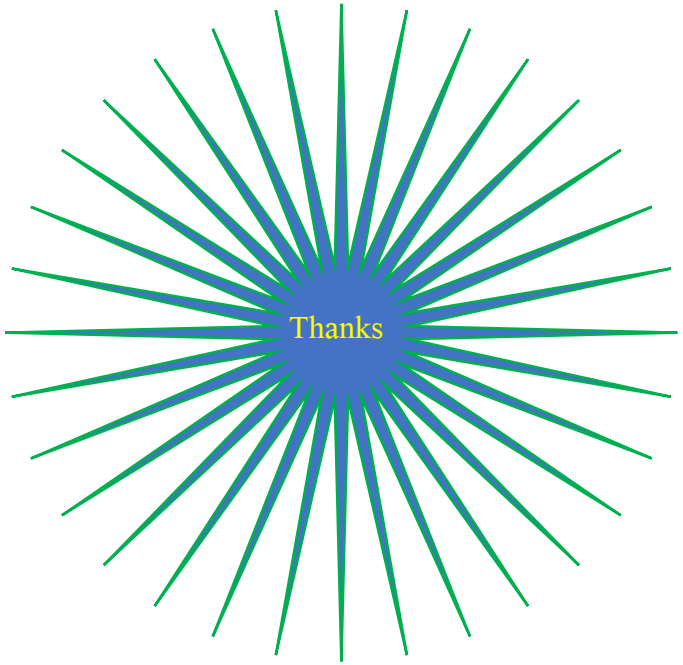
RMI clients, and the RMI registry, will require these classes as well as the service interface class. They can be copied to a local file system, or distributed remotely via a Web server using dynamic class loading (discussed earlier)

Summary

Summary

1. Overview of Remote Method Invocation
2. How Remote Method Invocation Work,
3. Defining an RMI Service Interface,
4. Implementing an RMI Service Interface,
5. Creating Stub and Skeleton Classes

Thank you for Listening



References

Java™ Network Programming and Distributed Computing, (David R.,Michael R. 2002), Publisher : Addison Wesley; ISBN: 0201710374