

Author: Santosh Kumar
ID#: 000080911
E-Mail: s0k0911@cs.tamu.edu
Date Submitted: 3rd November 2003

FRAMEWORK COMPARISON (.NET and J2EE) ON BASIS OF REMOTING

I. Remoting Overview

Remoting technologies provided by frameworks allow developers to create distributed applications in which the methods of remote objects can be invoked from other machines. This is usually done by getting a reference to the remote object through the means of a naming service and then invoking the methods of this object. Both .NET and J2EE provide extensive support for Remoting. The remoting support provided by the J2EE framework uses the Remote Method Invocation (RMI) technology whereas the remoting support provided by the .NET framework falls under the .NET Remoting category. This paper attempts to compare the J2EE and .NET framework on basis of Remoting. Some of the factors on which the comparison will be made are the remoting architecture, parameter passing techniques, naming services used, exception handling etc. among other things.

II. J2EE Remoting Overview

Prior to RMI, sockets and RPC were some of the technologies used in the making of Distributed systems. Sockets require the client and server to engage in applications-level protocols to encode and decode messages for exchange, and the design of such protocols is cumbersome and can be error-prone. And RPC does not translate well into distributed object systems, where communication between program-level *objects* residing in different address spaces is needed. In order to match the semantics of object invocation, distributed object systems require *remote method invocation* or RMI.

The Java platform's remote method invocation system has been specifically designed to operate in the Java application environment. The Java programming language's RMI

system assumes the homogeneous environment of the Java virtual machine (JVM), and the system can therefore take advantage of the Java platform's object model whenever possible.

III. .NET Remoting Overview

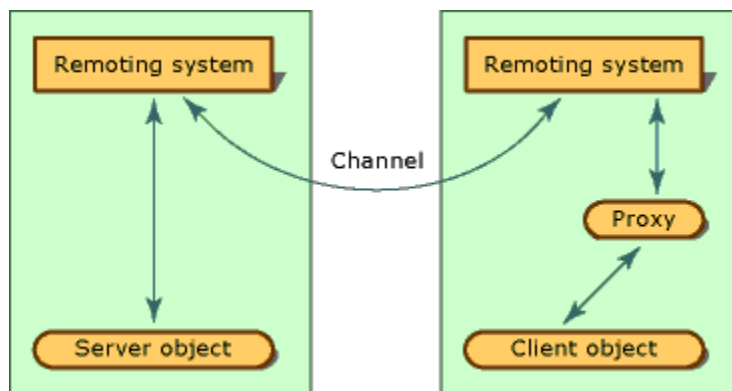
.NET Remoting is J2EE's answer to RMI. It provides a set of classes all part of the System.Runtime.Remoting namespace that provides support for communication between program-level objects residing in different address spaces. .NET Remoting enables one to build widely distributed applications easily, whether application components are all on one computer or spread out across the entire world. One can build client applications that use objects in other processes on the same computer or any other computer that is reachable over the network. We can also use .NET remoting to communicate with other application domains in the same process.

.NET remoting provides an abstract approach to inter process communication that separates the remotable object from a specific client or server application domain and from a specific mechanism of communication. Consequently, it is easily flexible and customizable.

IV. Architecture

- .NET Remoting architecture

Remoting process



The figure above shows the architecture of the remoting system depicting the general remoting procedure. The steps involved in a client invoking a method of a remote object are as follows.

- The client creates a new instance of the server class.
- The remoting system creates a proxy object that represents the class and returns to the client object a reference to the proxy.
- When a client calls a method, the remoting infrastructure handles the call, checks the type information, and sends the call over the channel to the server process.
- A listening channel picks up the request and forwards it to the server remoting system, which locates (or creates, if necessary) and calls the requested object
- The process is then reversed, as the server remoting system bundles the response into a message that the server channel sends to the client channel.
- Finally, the client remoting system returns the result of the call to the client object through the proxy.

- **RMI architecture**

RMI uses a standard mechanism (employed in RPC systems) for communicating with remote objects: *stubs* and *skeletons*. A stub for a remote object acts as a client's local representative or proxy for the remote object. The caller invokes a method on the local stub which is responsible for carrying out the method call on the remote object. In RMI, a stub for a remote object implements the same set of remote interfaces that a remote object implements.

When a stub's method is invoked, it does the following:

- Initiates a connection with the remote JVM containing the remote object
- Marshals the parameter to the remote JVM
- Waits for the result of the method invocation

- Unmarshals the return value or exception returned
- Returns the value to the caller

The stub hides the serialization of parameters and the network-level communication in order to present a simple invocation mechanism to the caller.

In the remote JVM, each remote object may have a corresponding skeleton. The skeleton is responsible for dispatching the call to the actual remote object implementation. When a skeleton receives an incoming method invocation it does the following:

- Unmarshals the parameters for the remote method
- Invokes the method on the actual remote object implementation
- Marshals the result to the caller.

Thus the main difference in the remoting architecture in J2EE and .NET is that RMI uses stubs and skeletons for remote object invocation whereas .NET Remoting uses object proxies. Another key distinction in the architecture of these two remoting solutions is that the .NET remoting solution is based on the Windows API, where as RMI is a platform independent way of implementing "remoting solutions". Thus since the .NET remoting solution is based on the Windows API it would be more flexible however it is a solution that would not "port" to a UNIX environment.

V. Parameter passing

Both RMI as well as .NET remoting support parameter passing by value and reference.

.NET Remoting

There are two kinds of objects with respect to parameter passing

- Nonremotable objects:

Nonremotable objects cannot be copied or represented in another application domain. These objects are accessible only from their original application domain.

- Remotable objects:

Remotable objects can be accessed outside their application domain or context using a proxy, or they can be copied and these copies can be passed outside their application domain or context; that is, some remotable objects are passed by reference and some are passed by value.

Marshal-by-value (MBV) objects declare their serialization rules (either by implementing `ISerializable` to implement their own serialization, or by being decorated with `SerializableAttribute`, which tells the system to serialize the object automatically) but do not extend `MarshalByRefObject`. The remoting system makes a complete copy of these objects and passes the copy to the calling application domain. Once the copy is in the caller's application domain, calls to the copy go directly to that copy. Further, MBV objects that are passed as arguments are also passed by value. Other than declaring the **`SerializableAttribute`** attribute or implementing **`ISerializable`**, you do not need to do anything to pass instances of your class by value across application or context boundaries.

Marshal-by-reference (MBR) objects are remotable objects that extend at least `System.MarshalByRefObject`. Depending on what type of activation has been declared, when a client creates an instance of an MBR object in its own application domain, the .NET remoting infrastructure creates a proxy object in the caller's application domain that represents the MBR object, and returns to the caller a reference to that proxy. The client then makes calls on the proxy. Remoting marshals those calls, sends them back to the originating application domain, and invokes the call on the actual object.

RMI

A non-remote object, that is passed as a parameter of a remote method invocation or returned as a result of a remote method invocation, is passed by *copy*; that is, the object is serialized using the object serialization mechanism of the Java platform. When an exported remote object is passed as a parameter or return value in a remote method call, the stub for that remote object is passed instead. Remote objects that are not exported

will not be replaced with a stub instance. A remote object passed as a parameter can only implement remote interfaces.

Parameters in an RMI call are written to a stream that is a subclass of the class `java.io.ObjectOutputStream` in order to serialize the parameters to the destination of the remote call. The `ObjectOutputStream` subclass overrides the `replaceObject` method to replace each exported remote object with its corresponding stub instance. Parameters that are objects are written to the stream using the `ObjectOutputStream`'s `writeObject` method. The `ObjectOutputStream` calls the `replaceObject` method for each object written to the stream via the `writeObject` method (that includes objects referenced by those objects that are written). The `replaceObject` method of RMI's subclass of `ObjectOutputStream` returns the following:

- If the object passed to `replaceObject` is an instance of `java.rmi.Remote` and that object is exported to the RMI runtime, then it returns the stub for the remote object. If the object is an instance of `java.rmi.Remote` and the object is not exported to the RMI runtime, then `replaceObject` returns the object itself. A stub for a remote object is obtained via a call to the method `java.rmi.server.RemoteObject.toStub`.
- If the object passed to `replaceObject` is not an instance of `java.rmi.Remote`, then the object is simply returned.

VI. Distinction By Example

In this section we will attempt to illustrate some differences between .NET remoting and RMI with the help of an example. Here we will compare the two frameworks on basis of the steps involved in the creation and invocation of a distributed object. The steps involved will be

- The creation and implementation of the distributed objects
- The generation of the proxy and skeletons
- Lookup of the remote object, getting a reference to it and invoking a method starting from the customer.

Creation of the objects

Here we implement a simple HelloWorld server. The JAVA RMI mandates the creation of distributed interfaces whereas Remoting makes it possible to directly implement a server in a class.

HelloWorld server in .NET Remoting

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;

namespace NETRemotingSamples
{
    public class HelloServer: MarshalByRefObject
    {
        public HelloServer()
        {
            // activated with each call in "SingleCall"
            // activated once in "Singleton" (shared state)
            Activated distributed Console.WriteLine("Object!");
        }

        public string HelloMethod(string name)
        {
            Console.WriteLine("HelloMethod: { 0 } ", name);
            return "Hello" + name
        }
    }
}
```

- **RMI**

```
package hello;

// Interface defining the distant service sayHello

public interface Hello extends java.rmi.Remote
{
```

```

        public String helloMethod(String name) throws
            java.rmi.RemoteException;
    }

Hello.java
_____ Implementation _____

package hello;
import java.rmi.server.*;
// Class that implements the RMI interfaces Hello.

public class HelloImpl implements Hello, UnicastRemoteObject
{

    public String helloMethod(String name) throws
        java.rmi.RemoteException
    {
        System.out.println ("Hello.helloMethod called");
        return " Hello "+ name;
    }
}

```

The code of the server resembles any other object, it contains public or private methods and can derive from an unspecified interface, but it is not obligatory. In RMI, any distributed interface must be typified *java.rmi.server.Remote* and the implementation to derive from **UnicastRemoteObject**, whereas in Remoting no particular typing is necessary.

Registering of the server is of utmost importance as there should be a way for clients to locate this service.

The registering of a server in remoting is done via the call to the method `RemotingConfiguration.RegisterWellKnownServiceType()` taking parameters as shown below:

- The type of the distributed class (HelloServer)
- The URL of the waiter

- Mode of activation

```

using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;

namespace NETRemotingSamples
{
    public class MainServer
    {
        public static int Main(string [ ] args)
        {
            // Declaration of the channels
            TcpChannel chan = new TcpChannel(8085);
            ChannelServices.RegisterChannel(chan);
            // Registering of the server
            RemotingConfiguration.RegisterWellKnownServiceType (
            Type.GetType("HelloServer, helloserver"),
                "SayHello", WellKnownObjectMode.Singleton);
            System.Console.WriteLine("Hit < enter > to exit");
            System.Console.ReadLine();
            return 0;
        }
    }
}

```

Registering of the server in RMI is as shown below:

```

package hello;

import java.rmi.server.*;

```

```
public class MainServer
{
    public static void main(String args[ ])
    {
        // Creation of the authority
        HelloImpl obj = new HelloImpl("SayHello");
        // Registration of the object
        java.rmi.Naming.rebind ("rmi://localhost: " +
            port + "/SayHello ", obj);
        System.out.println("Bound RMI object in registry");
    }
}
```

Channels

Channels are objects that transport messages between applications across remoting boundaries, whether between application domains, processes, or computers. A channel can listen on an endpoint for inbound messages, send outbound messages to another endpoint, or both. This enables one to plug in a wide range of protocols, even if the common language runtime is not at the other end of the channel. The default protocols proposed by Remoting are HTTP and TCP whereas RMI proposes JRMP and IIOP over TCP. However in RMI, it is possible to enrich the existing protocols (IIOP and JRMP) via the RMI SocketFactory class.

The directory

Rmi uses a directory called RmiRegistry to store the references (Proxies) of the distant services. This service, which is itself a distributed object, must be launched manually or dynamically before any communication occurs between a customer and a service. NET adopts a different approach insofar as the customer communicates directly with the server on a port specified between the two parts on the URL level.

Implementation of the client

There is not much distinction between the clients in .NET remoting and RMI. The essence of this operation involves getting a reference to the distributed object and calling upon a method on the distributed object.

The code below is an implementation in .NET remoting

HelloClient.cs

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels.Http;

using NETRemotingSamples;

public class HelloClient
{
    public static void Main (String[ ] args)
    {
        // passes the URL of the server and its type
        // returns a proxy to the server through the activator
        HelloServer helloProxy = (HelloServer)
            Activator.GetObject(
                typeof (HelloServer),
                "tcp://localhost:8085/SayHello");

        // the call of the method is carried out
        helloProxy.HelloMethod("Hello World!!");
    }
}
```

```
}
```

The code below is an implementation in RMI.

```
package hello;
import java.rmi.*;

public class HelloClient
{

    public static void main(String args[])
    {
        // Get a reference to the distant server distributed object
        Hello helloProxy = (Hello) Naming.lookup(
            "/" + host + ":" + port + "/SayHello");

        // The call to the method is carried out
        String message = helloProxy.helloMethod("Hello World !!");
    }
}
```

An important distinction between the two approaches is that .NET Remoting uses the concept of Activators to get a reference to a remote object whereas RMI uses the Naming class to get a reference to a remote object. The naming class in turn uses the RmiRegistry to bind to the remote object on the remote machine.

VII. Conclusion

In this paper we have attempted to compare the .NET and J2EE frameworks on the basis of the remoting features provided by them. Though they are semantically similar in most respects there are some subtle difference between the two. The main differences between them are summarized in the table below.

	.NET Remoting	RMI
Proxy	Dynamics	Statics (rmic) or dynamics
Skeletons	Integrated into Framework	Integrated into Framework
Distributed object	Classes	Remote Interfaces
Configuration	File XML	System Property
Distributed directory	No (system interns containing tables of objRef	RmiRegistry
Addition of protocols	Channels	SocketFactoryImpl
Addition of formats	Formatters	Serialization
Activation	Single Call, Singleton or Customer Activated	API with dimensions server Activable objects
CustomProxy	CustomRealProxy	DynamicProxy
Existing protocols	HTTP, TCP, SOAP	JRMP, IIOP, ORMI (Orion), T3 (Web Logic)
Management of error	Remote Exceptions	Remote Exceptions

In spite of these differences from a developer's standpoint, the remoting features in these frameworks are pretty much similar and no one framework offers any substantial advantage over the other.

VIII. References

- <http://java.sun.com/j2se/1.4.2/docs/guide/rmi/>
- <http://msdn.microsoft.com/library/>
- <http://www.dotnetguru.org/articles/RemotingRmi.htm>
- **Rahmel Dan, .NET Framework – A Programmer's Reference**

