Inquiry: The University of Arkansas Undergraduate Research Journal

Volume 5 Article 12

Fall 2004

A Comparative Evaluation of .net Remoting and JAVA RMI

Taneem Ibrahim
University of Arkansas, Fayetteville

86 INQUIRY Volume 5 2004

A COMPARATIVE EVALUATION OF .NET REMOTING AND JAVA RMI

By: Taneem Ibrahim
Department of Computer Science and Computer Engineering

Faculty Mentor: Dr. Amy Apon
Department of Computer Science and Computer Engineering

Abstract:

Distributed application technologies such as Micrososoft.NET Remoting, and Java Remote Method Invocation (RMI) have evolved over many years to keep up with the constantly increasing requirements of the enterprise. In the broadest sense, a distributed application is one in which the application processing is divided among two or more machines. Distributed middleware technologies have made significant progress over the last decade. Although Remoting and RMI are the two of most popular contemporary middleware technologies, little literature exists that compares them. In this paper, we study the issues involved in designing a distributed system using Java RMI and Microsoft.NET Remoting. In order to perform the comparisons, we designed a distributed distance learning application in both technologies. In this paper, we show both similarities and differences between these two competing technologies. Remoting and RMI both have similar serialization process and let objects serialization to be customized according to the needs. They both provide support to be able to connect to interface definition language such as Common Object Request Broker Architecture (CORBA). They both contain distributed garbage collection support. Our research shows that programs coded using Remoting execute faster than programs coded using RMI. They both have strong support for security although implemented in different ways. In addition, RMI also has additional security mechanisms provided via security policy files. RMI requires a naming service to be able to locate the server address and connection port. This is a big advantage since the clients do not need to know the server location or port number, RMI registry locates it automatically. On the other hand, Remoting does not require a naming service; it requires that the port to connect must be pre-specified and all services must be well-known. RMI applications can be run on any operating system whereas Remoting targets Windows as the primary platform. We found it was easier to design the distance learning application in Remoting than in RMI. Remoting also provides greater flexibility in regard to configuration by providing support for external configuration files. In conclusion, we recommend that before deciding which application to choose careful considerations should be given to the type of application, platform, and resources available to program the application.

Introduction:

A *distributed system* is a collection of loosely coupled processors interconnected by a communication network [8]. From the point view of a specific processor in a distributed system, the rest of the processors and their respective resources are *remote*, whereas its own resources are local. Generally, one host at one site or machine, the server, has a resource that another host at another site or machine, the client (or the user), would like to use [8]. The purpose of the distributed system is to provide an efficient and convenient environment for such sharing of resources.

A *distributed application* is one in which the application processing is divided among two or more machines [8]. This division of processing also implies that the data involved is also distributed. Distributed application technologies such as Micrsosoft.NET Remoting, Distributed Component Object Model (DCOM), Java Remote Method Invocation (RMI), and Common Object Request Broker Architecture (CORBA) have evolved over many years to keep up with the constantly increasing requirements of the enterprise. They all are based on objects that have identity and they either have or can have state [1]. Developers can use remote objects with virtually the same semantics as local objects. This simplifies distributed programming by providing a single, unified programming model. They are also associated with a component model. A *component* is a separate, binary-deployable unit of functionality [3]. Using components in a distributed application increases its deployment flexibility.

In this paper, we focus on Microsoft.NET Remoting and Java RMI. These two are by far the most popular distributed technology at present. Java RMI, acronym for Remote Method Invocation, is designed by Sun Microsystems which targets working on distributed objects on Java virtual machines [2]. RMI allows Java developers to make calls to objects in different Java Virtual Machines, whether they are in different processes or on different hosts. .NET Remoting is designed by Microsoft Corporation as a successor to DCOM. .NET Remoting is the manner in which .NET makes objects callable over a network. In contrast to RMI's emphasis on Java-only development, .NET Remoting supports multi-language interoperability. Both of these technologies share similarities and differences. For

developing distributed application, a lot of times developers are confronted with the question of which technology to choose. This is often a daunting task. This paper attempts to make a comparative evaluation between these two very popular distributed technologies in various aspects of designing a distributed application.

In order to perform these comparisons, we have designed a simple distance learning application in both technologies. In the rest of the paper, we give an architectural background of Remoting and RMI, describe the distributed distance learning application, and present the experimental results of the evaluation. In conclusion, we show our observations in regard to which application to choose, and recommend any future research work that can be done in this area.

Overview of Architectures:

Microsoft.NET Remoting Architecture

.NET Remoting enables client programs to call methods of remote objects. When the client creates a connection to the server object, the .NET Framework creates a proxy object on the client [3]. The proxy object provides the client with the same view of the server object that it would have if the server objects were in its application space. It can call the server object's methods through the proxy object. Figure 1 depictsthis process. The figure shows a client method that calls a method on the remote object through the proxy object created at run time. Any data passed by the client method to the proxy is packaged by a formatter so that it can be sent across the network. The process of packaging the data for transaction is called *marshalling* [3].

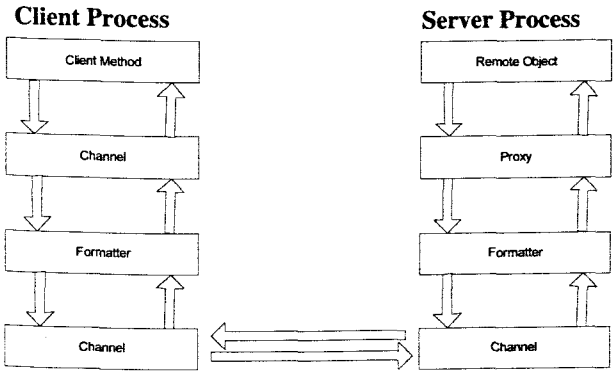

Figure 1: **How Remoting Works**

After the data is marshaled by the formatter, the data is sent through the channel, out across the network to the server. A formatter on the server unmarshals the data and calls the appropriate method on the server. It passes the data to the method. When the server object's method finishes its processing, it send any data it might need to return back to the formatter, and the entire process is reversed.

Java RMI Architecture

The Java RMI architecture is based on the *broker* pattern [4]. The *broker* pattern is a broker with indirect communication between proxies. The Java RMI architecture consists of three layers: the *stub/skeleton* layer, the *remote reference* layer and the *transport* layer [4].

RMI is a layer on top of the Java Virtual Machine which leverages the Java system's built-in garbage collection, security and class- loading mechanisms [6]. The application layer sits on top of the RMI system. A Remote Method Invocation from a client to a remote server object travels down through the layers of the RMI system to the client-side transport [8]. Next, the invocation is sent - potentially via network communication - to the server-side transport, where it then travels up through the transport to the server.

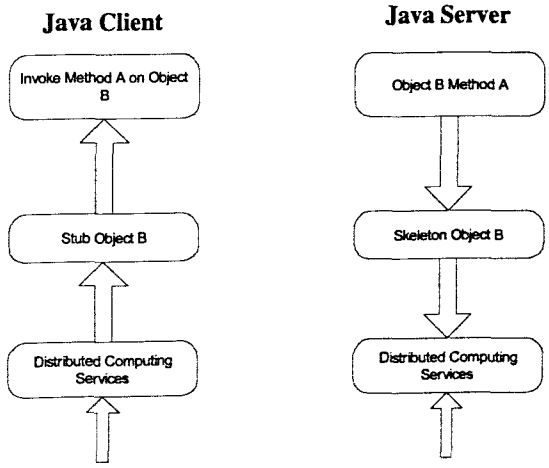

Figure 2: **How Java RMI works**

A client invoking a method on a remote server object actually uses a stub or proxy as a conduit to the remote object [6]. A client-held reference to a remote object is a reference to a local stub, which is an implementation of the remote interfaces of the object and which forwards invocation requests to it via the remote reference layer.

Distance Learning Application:

The distributed application we have designed is a simple distance learning application. The remote methods in the application includes:- adding a course, deleting a course, view schedule, view a class description, view and submit homework/

quizzes/handouts. These remote methods are defined in the remote server implementation class. The client makes a call to these remote methods with an object initiated during the connection to the server. We use TCP as the communication protocol. For storing information regarding courses, schedule, or course assignments we used simple text files. The reason for choosing text files as storage location is due to the fact if we use database such as Microsoft Access or MySQL and used Open Database Connectivity (ODBC) or Java Database Connectivity (JDBC) as the application programming interface, the comparison between RMI and Remoting will not be fair since database transactions will significantly affect the performance. We wanted to use the same storage facility for both applications and focused our measurements on the distributed programming aspects only. While designing the application we tried to keep very similar programming techniques and algorithms.

object in this case is the attempt the client makes to ask for a service from the server. A student can also submit his or her assignments, participate in quizzes, and view handouts for a particular class. The student also receives immediate feedback on his or her quiz grades online. We are currently adding support for instructors so that they can login as a course administrator and update homework, handouts, quizzes and lecture material online.

Comparative Evaluation:

In this section we list some of the similarities and differences between Java RMI and .NET Remoting. .NET Remoting and Java Remote Method (RMI) are functionally equivalent. Both systems allow applications to communicate betweeen processes and machines, enabling objects in one application to manipulate objects in another. Some of the key similarities and differences are listed in the table below:-

	RMI	Remoting
Inheritance	Single Class, Multiple interface inheritance	Single Class, multiple interface inheritance
Communication	Socket	Channel
Naming Service	RMI Registry, mapping from named server object to URL	Hash table of object references
Configuration	System Property	XML file
Remotable	Remote interface	MarshallByRef
Protocol	JRMP, IIOP	HTTP, TCP, SOAP
Activation	Can be activated	Singlecall, Singleton, CAO
Format	Serialization	SOAP or Binary Formatter
Distributed Garbage Collector	Yes	Yes
Error	Remote Exception	Remote Exception
Skeletons	Integrated within the framework	Integrated within the framework

Figure 3: Key Differences between RMI and Remoting [2]

When a client (student) connects to the server (the host university/college), he or she has to verify his login name first. After that a menu prompts with options to add or delete classes, view the student's current schedule of classes, homework, quizzes or handouts. After a student selects an option a remote call to the method is made with appropriate parameters and after processing the request the result is printed back to the client. The remote

Similarities:

Although Microsoft.NET Remoting and Java Remote Method Invocation (RMI) are implemented quite differently and are based on different business philosophies, they are remarkably similar in many ways. These similarities include:

i) Copies and References:

In common with RMI, Remoting provides the distinction between classes that will be referenced remotely and class that will be copied across the network via serialization [5]. Serialization is the process of converting a set of object instances that contain references to each other into a linear stream of bytes, which can then be sent through a socket, stored to a file, or simply manipulated as a stream of data [3]. Serialization is the mechanism used by RMI to pass objects between Java Virtual Machines (JVMs), either as arguments in a method invocation from a client to a server or as return values from a method invocation. Similarly, all of the .NET primitive types are annotated with the *Serializable* attribute. Following is an example of how to make a class serializable in .NET Remoting:

```
Using System;

[Serializable]

public class View_Grades{

 //do something

}
```

In Java RMI:-

```
import java.io.*;

public access class View_Grades extends AbstractList
implements List,cloneable,java.io.Serializable {

//do something

}
```

ii) Customizing Object Serialization:

The .NET framework allows a type attributed with the SerializableAttribute custom attribute to handle its own serialization by implementing the ISerializable interface [5]. This interface defines one method, *GetObjectData:*

```
void GetObjectData(Serialization info,
StreamingContext context);
```

The Java RMI provides a similar functionality *java.io.Externizable,* which allows programmer to take responsibility of the serialization process. Whereas in Remoting the *ISerializable* interface contains only one method *GetObjectData,* in RMI *Externizable* contains two methods [3]:

```
public void readExternal (ObjectInput
in);

public void writeExternal (ObjectOutput
out);
```

iii) Object-Oriented Remote Procedure Call (RPC):

Remote Procedure Calls (PRC) is a traditional mechanism that allows applications to call procedures that exist on other computers [2]. RPC makes use of proxy methods that have the same signature as the remote method but also has code Remote Procedure Calls (RPC) is a traditional mechanism that allows applications to call procedures that exist on otherfor transferring data between the client and the server [2]. Parameters are bundled and sent to the server where they are unbundled and passed into the requested method. The return values are treated in the same way.

Both Java RMI and .NET Remoting implement an object oriented approach in remote method calls built on top of existing RPC mechanism. While RPC allows the program to call procedures over the network, RMI and .Net Remoting permits to call an object's methods over the network [2]. In order to make a remote method call over the network, the program needs to call the method through the server object that was initiated during the connection.

The following code snippet shows how to make remote method calls in .NET Remoting:-

```
MySearchIntf      MyObject      =
MySearchIntf)Activator.GetObject

typeof(MySearchIntf),"tcp://
localhost:8085/MySearch");

MyObject.Add_Course(course);
```

Here we have activated a server object called *MyObject* and invoked a remote method named *Add_Course* on that object.

The following code snippet shows how to make remote method calls in Java RMI:-

```
SimpleRMIInterface  myServerObject  =
(SimpleRMIInterface) Naming.lookup("//
" +            serverName    +    "/
SimpleRMIImpl");

myServerObject.Add_Course(temp);
```

Here we bind the server object *myServerObject* to the object in the client and then invoke a remote method *Add_Course* on that object.

iv) Interface Definition Language (IDL):

CORBA, the acronym for Common Object Request Broker Architecture, is a widely used communications model for building distributed (multi-tier) applications that connect both cross-platform and cross-language clients to server-based services. Neither RMI nor .Net Remoting require a secondary language for defining the remote interfaces, such as CORBA IDL [2]. However both .NET Remoting and Java RMI provides support for building CORBA Server. Making that connection requires a way to describe .NET objects as CORBA objects so that J2EE

.NET objects, so that managed .NET code can interact with them [11]. In other words, you need some mediating code that can translate objects and method calls from CORBA's representation to the .NET framework's representation [10].

v) Remote Object Lifetime:

Java RMI provides support for Distributed Garbage Collection. Server tracks clients who have its stub and keeps a count of such clients [2]. Count is decremented when client explicitly relinquishes reference. If the reference count reaches zero, the object is garbage collected. An objects *lease* is essentially a counter that specifies its lifetime [3]. RMI also lets distributed references to be leased. Clients automatically try to renew leases as long as a stub has not been garbage collected [4]. .NET Remoting employs similar concept of object leasing. Leases are controlled by the server's lease manager object, which is created in the server's application domain. By default, .NET Remoting gives client activated and singleton objects a lease of five minutes [3]. It decrements the lease at certain intervals. Each time a client accesses the object, the lease manager increases the lease by two minutes [3].

Differences:

i) Naming Service:

In order to create a socket connection, it is necessary to have a machine address and a port. However, you also want to avoid hard coding the server locations into a client application. In order to solve this problem, RMI makes the client "ask" a dedicated server which machine and port they can use to communicate with a particular server [3]. This dedicated server is often known as a *naming service* [3]. In RMI, the default naming service that ships with Sun Microsystem's version of the JDK is called the *RMI registry*. Messages sent to the registry via static methods that are defined in the *java.rmi.Naming* class [3]. RMI registry is usually started as a standalone server. Unfortunately there is not a reliable way to be backwards-compatible with the RMI registry in terms of being backward-compatibles with already existing naming services and with future versions of the naming services. One advantage to having a naming server is that you do not need to know the server address or the port number.

NET does not rely on a registry to locate instances of remote classes. Instead of using naming services, a client communicates with the server on a pre-specified port [2]. Services must be *well-known*, meaning that the client must know the location of the remote service at run time. In Remoting on the server side you create an instance of a *TCPChannel* or *HttpChannel* class, and pass its constructor a port number. Thus, the port number is the port on which the server listens for the client. Then the program needs to register the channel via the static method *RegisterChannel* [6]. In this way, .NET Remoting eliminates any need for have a separate naming services in order to locate the remote services.

ii) Language and Platform Interoperability:

A significant difference between RMI and Remoting is that when you develop with Java, it provides a single language targeted at multiple operating systems, whereas, .NET provides multiple languages (C#.NET, Visual Basic.NET, and C++.NET) targeted primarily to a single operating system (Windows). RMI application can be run on any operating system that has Java Virtual Machine (JVM). Both Java RMI and Remoting are tightly coupled with their languages which indicate that these technologies do not interoperate with each other [2].

.NET Remoting meets the interoperability goal by supporting open standards such as HTTP, Simple Object Access Protocol (SOAP), Web Service Description Language (WSDL), and Extensible Markup Language (XML) [6]. To communicate with non .NET clients and servers, a developer can implement a SOAP formatter.

RMI Inter-ORB Protocol (RMI-IIOP) provides a convenient way for any language that "speaks CORBA" to talk to Java [2]. The CORBA IIOP protocol is part of the JDK 1.3 specification. One of the advantages of RMI-IIOP over CORBA is the developers do not need to learn the CORBA Interface Definition Language (IDL).

iii) Performance Speed:

In order to compare the speed of Remoting and RMI, we performed clock timing on remote method calls. We have instantiated a *date* object before the method call and instantiated another *date* object after the method call returned to the calling class. After that, we took the difference in time in the process in milliseconds. The performance depended on how fast the method was able to retrieve data from a text file and then write that data to another text file and add it to the student's record. We performed this benchmarking on two remote methods, namely, *Add_Course* and *Delete_Course*. For adding courses we counted from the time it took for adding up to ten courses and same for deleting courses as well.The following two graphs show the performance comparison of RMI and Remoting when called with *Add_Course* and *Del_Course* remote method:

In both of these graphs, .NET Remoting runs about twice as fast as RMI does and the differences are greater as the number of courses increases. We tried to perform this benchmarking as much fair as possible. Two important things to point out here that, we used Visual Studio.NET 2003 and Borland JBuilder 8.0 SE as editors and we ran these benchmarks on a Windows machine. Following are the code snippets for the test methods:

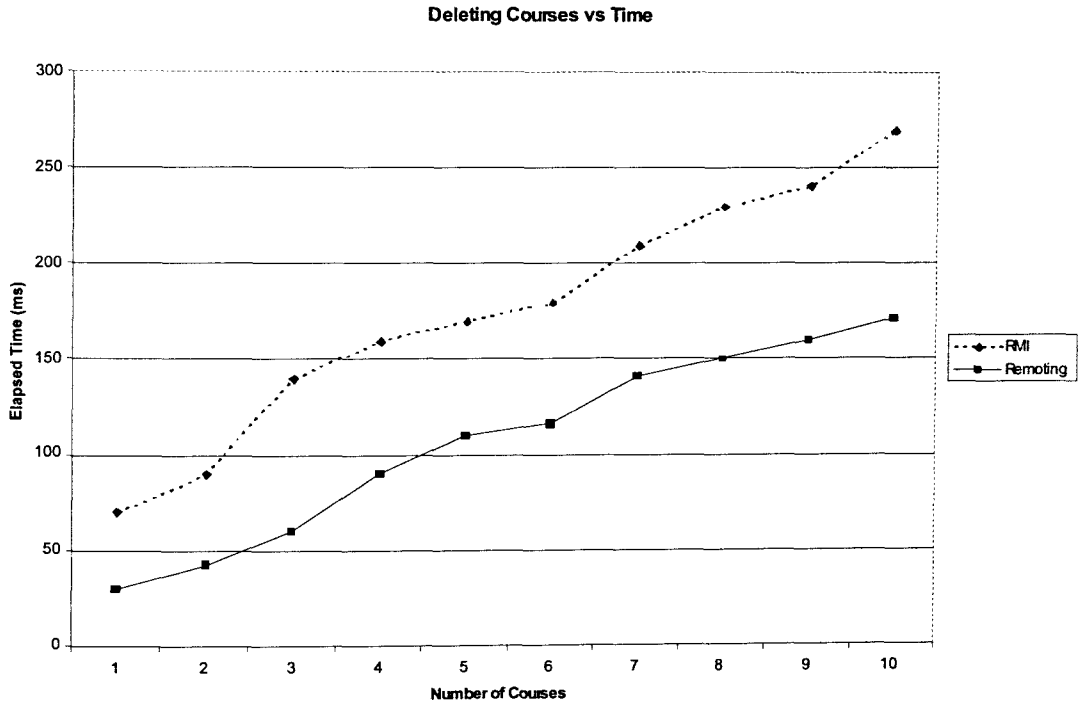

Figure 4: **Performance Comparison of RMI and Remoting for adding courses**

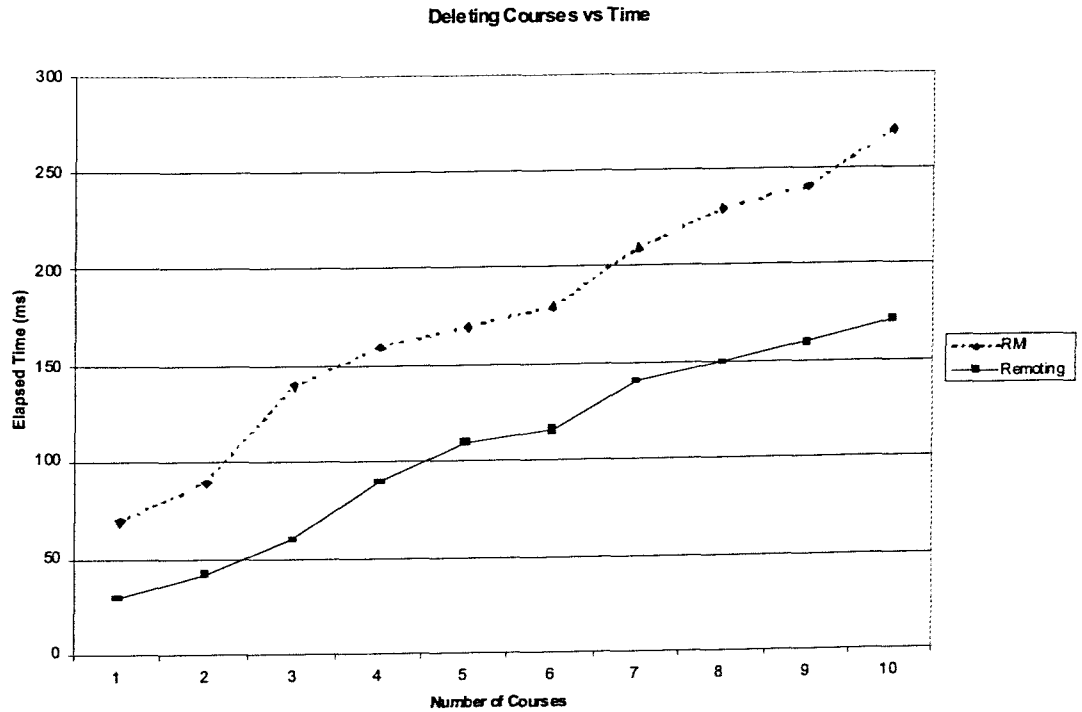

Figure 5: **Performance Comparison of RMI and Remoting for deleting courses**

In .NET Remoting:

```
DateTime s = DateTime.Now;
start = s.Millisecond;
MyObject.Add_Course(course);
DateTime f = DateTime.Now;
finish = f.Millisecond;
total = total+ (finish-start);
```

In Java RMI:

```
Date d = new Date();
sTime = d.getTime();
myServerObject.Add_Course(temp);
Date d2 = new Date();
fTime = d2.getTime();
dTime = dTime + (fTime-sTime);
```

iv) Security:

No aspect of distributed systems has gotten more attention lately than security. The .NET's Common Language Runtime (CLR) automatically provides a minimal form of checking to ensure that none of the program's assemblies have been altered or replaced [3]. It utilizes a technique called *hashing* to generate an identifier for each assembly based on assembly's contents. Every assembly generates a unique hash which is stored by Visual Studio in a hash file [3]. Besides this automatic checking, .NET also supports strong-named assemblies. This helps the .NET framework prevent *spoofing* on a network. .NET also provides a Signcode tool called *SIGNCODE.EXE* with Visual Studio so that developers can assign trust levels to the assemblies [3]. It also provides tools to authenticate signatures and validate custom certificates issued by entities such as VeriSign. However, the best way to implement a secure remoting system is to host the remote server inside Internet Information Service (IIS) [6]. The best part of hosting inside IIS is that you can use strong security features without changing client's code or the server's code.

Java RMI adopts a different approach. It provides security policy files that defines what kind of permission a program has. There are nine basic type of permissions:- AWT, File, Network, Socket, Property, Reflection, Runtime, Security, and Serializable [3]. Every RMI application needs to contain a *security.policy* file that will indicate the type of permissions available. Java2 also comes with a simple GUI application, called *policytool*, that helps you edit policy files. Within a running JVM, permissions are enforced by an instance of the *SecurityManager* class [3]. When a program attempts to do something that requires permission, the instance of *SecurityManager* is queried to see whether the operation succeeds.

v) Ease of Programming:

In developing this simple distance learning application, we found developing remote applications in .NET Remoting is easier than Java RMI and other texts [4] also concur with similar opinions. .NET has a rich debugging API. .NET Remoting is quite flexible in terms of building application. You can configure a Remoting application using a configuration file or programmatically. Both server and client can be configured in this way. Using configuration files allow the administrators to configure the application's Remoting behavior without recompiling the code [6]. Remoting also has more options in terms of publishing and activating remote objects. The framework can be configured depends on the application needs [2]. For example, an application can use either HTTP or TCP as the communication protocol, and either SOAP formatter or binary formatter as the object serialization. Following is an example of a MyServer.config.exe file configured using a configuration instead of programmatically:

```
<configuration>
 <system.runtime.remoting>
   <application name="MyServer">
    <service>
       <wellknown mode="Singleton"
       type="MyServerLib.MyServeImpl,
       MyServerLib" objectUri="JobURI" />
    </service>
    <channels>
       <channels ref="http" port ="8085" />
    </channels>
   </application>
 </system.runtime.remoting>
</configuration>
```

vi) Publishing and Activation Object Service:

Java RMI includes a generic and reusable factory implementation, called the *Activation Framework*, which handles the details of launching servers on remote machines easily and transparently [2]. Instead of using *UnicastRemoteObject*, the remote implementation extends the *Activation* class.

In Remoting there are two types of activations- *client activation* and *server activation* [4]. When a server publishes a service, the activation type defines how and when the object will be created, and how lifecycle of the object will be controlled. When a client registers for an activated service, the runtime is provided with information about how to create new proxies to represent the remote type. There are two variants in the server activation of objects- *Singleton* and *SingleCall* [4].

vii) Implementation of Remote Objects:

In Java RMI the developer has to create an Interface where the developer declares all the remote methods with appropriate remote exceptions. This Interface class is implemented by the server implementation class where these remote operations are defined. However, in Remoting you are not required use an

Interface class. However, a client must be able to obtain the metadata describing the remote type [6]. A solution to this problem is to have the client add a reference to the assembly containing remote objects implementation.

.Net remoting cannot run a non-default constructor when connecting to well-known objects [2]. Java RMI does not have this limitation.

Conclusion:

Although .NET Remoting and Java TMI share some common traits mostly due to the fact that they both are object-oriented distributed technology, the basis and structure of the .NET Remoting is different from Java RMI. .NET Remoting service is easier to program and provides greater flexibility features such as the configuration files. RMI was added to Java after the original release of the platform, while the Remoting system has always been a part of the .NET framework from the beginning. This provides .NET Remoting a deep integration with the underlying platform. Both Java RMI and .NET Remoting preserves the security features provided by their individual run time environment. In addition to that, Java RMI also provides support for security policy files for stronger security. Remoting is much faster and has excellent debugging support. .NET Remoting does not define a standard protocol; it only has a set of channels, formatter and message sinks, adding more flexibility to the developer. Java RMI on the other hand is free.

To conclude, both .NET Remoting and Java RMI are great solutions to develop distributed applications, which to choose depends on the type of application, platform, resources and tools available for designing the application.

References:

[1] Campione, Mary. *The Java Tutorial Continued*. First Edition. Addison Wesley Press, Massachusetts, USA, 2000

[2] Chen, Li. "Java RMI vs .NET Remoting", http://students.cs.tamu.edu/jchen/cpsc689-608/comparison/10c0607-framework_comp_lichen.pdf, 03.12.2004

[3] Conger, David. Remoting with C# and .NET. First Edition. Wiley Publishing, Indiana, USA 2003

[4] Grosso, William. *Java RMI*. First Edition. O'Reilly and Associates, California, USA 2001

[5] Jones, Allen and Freeman, Adam. *C# for Java Developers*. First Edition. Microsoft Press, Washington, USA 2003

[6] McLean, Scott. *Microsoft .NET Remoting*. First Edition. Microsoft Press, Washington, USA 2003

[7] Plasil and Stal. "An Architectural View of Distributed Objects and Components in CORBA, Java RMI, and COM/DCOM." Software Concepts and Tools, Germany, 1998

[8] Sharp, John. *Visual C#.NET: Step by Step*. First Edition. Microsoft Press, Washington, USA 2002

[9] Silberschatz, Abraham. *Operating Systems Concepts*. Sixth Edition. John Wiley & Sons, INC, New York, USA 2002

[10] Sun Microsystems. http://java.sun.com/marketing/collateral/rmi_ds.html, 02.16.2003

[11] Swart, Bob. "Connecting CORBA to .NET", http://www.devx.com/interop/Article/19916?trk=DXRSS_WEBDEV, 03.02.2004

Faculty Comment:

Mr. Ibrahim's faculty mentor, Professor Amy Apon, made the following comments about her student's work:

> The computer industry is a very rapidly changing field of study. New software tools and versions of tools from vendors such as Sun Microsystems, IBM, The Open Software Group, and Microsoft, become available on a regular basis. However, the personnel and training effort required by companies that want to use these new tools is enormous, so that very often old tools with less capability continue to be used even when newer, more capable tools are available. In addition, it is difficult for college students to learn new tools since professors also must learn how to use these tools in order to incorporate them into their classes. In general, there is a lack of understanding in the industry about what may be gained by using a new tool as compared to an existing tool or other newer tools.
>
> For his research, Taneem has performed an unbiased study and comparison of two competing technologies, Microsoft .Net, and Java-based tools for distributed computing. The problem of comparing the entire programming capability of .Net and Java-based tools is too large for a single project, and Taneem has chosen to limit his comparison to the remoting capability of .Net and the Java RMI system. Some of the study is quantitative. As a part of his project, Taneem has learned each of these new tools and has implemented a substantial test software system using each tool. This example provided by this code in both systems is a very nice contribution alone. He has written benchmarking code that executes equivalently in both systems and has compared the relative speed of execution of the two tools. In addition to the quantitative study, a portion of Taneem's research is qualitative, and includes a literature search and comparative study along several specific criteria, including inheritance, communication, naming service, configuration, protocol, activation, format, distributed garbage collection, error handling, and skeletons, and perceived ease of use.
>
> The results of Taneem's study show that it is possible to perform a good comparison of Microsoft .NET and Java RMI as an undergraduate research project. It also demonstrates that there is potential for companies and universities to move to new tools with a reasonable amount of effort. Taneem found significant differences in the speed of execution of the two tools, with .NET outperforming Java RMI by about a factor of two. Taneem found similarities and differences on several criteria. He found .NET to be an easer programming environment and describes in experiences in detail. The results of this research are particularly important because there are not many studies of this type, and a contribution is greatly needed in this area.