

A Detailed Comparison of CORBA, DCOM and Java/RMI

(with specific code examples)

Gopalan Suresh Raj

Introduction

Distributed object computing extends an object-oriented programming system by allowing objects to be distributed across a heterogeneous network, so that each of these distributed object components interoperate as a unified whole. These objects may be distributed on different computers throughout a network, living within their own address space outside of an application, and yet appear as though they were local to an application.

Three of the most popular distributed object paradigms are Microsoft's **Distributed Component Object Model (DCOM)**, OMG's **Common Object Request Broker Architecture (CORBA)** and JavaSoft's **Java/Remote Method Invocation (Java/RMI)**. In this article, let us examine the differences between these three models from a programmer's standpoint and an architectural standpoint. At the end of this article, you will be able to better appreciate the merits and innards of each of the distributed object paradigms.

CORBA relies on a protocol called the **Internet Inter-ORB Protocol (IIOP)** for remoting objects. Everything in the CORBA architecture depends on an **Object Request Broker (ORB)**. The ORB acts as a central Object Bus over which each CORBA object interacts transparently with other CORBA objects located either locally or remotely. Each CORBA server object has an interface and exposes a set of methods. To request a service, a CORBA client acquires an object reference to a CORBA server object. The client can now make method calls on the object reference as if the CORBA server object resided in the client's address space. The ORB is responsible for finding a CORBA object's implementation, preparing it to receive requests, communicate requests to it and carry the reply back to the client. A CORBA object interacts with the ORB either through the ORB interface or through an **Object Adapter** - either a **Basic Object Adapter (BOA)** or a **Portable Object Adapter (POA)**. Since CORBA is just a specification, it can be used on diverse operating system platforms from mainframes to UNIX boxes to Windows machines to handheld devices as long as there is an ORB implementation for that platform. Major ORB vendors like [Inprise](#) have CORBA ORB implementations through their [VisiBroker](#) product for Windows, UNIX and mainframe platforms and [Iona](#) through their [Orbix](#) product.

DCOM which is often called '*COM on the wire*', supports remoting objects by running on a protocol called the **Object Remote Procedure Call (ORPC)**. This ORPC layer is built on top of DCE's RPC and interacts with COM's run-time services. A DCOM server is a body of code that is capable of serving up objects of a particular type at runtime. Each DCOM server object can support *multiple interfaces* each representing a different behavior of the object. A DCOM client calls into the exposed methods of a DCOM server by acquiring a pointer to one of the server object's interfaces. The client object then starts calling the server object's exposed methods through the acquired interface pointer as if the server object resided in the client's address space. As specified by COM, a server object's memory layout conforms to the C++ vtable layout. Since the COM specification is at the binary level it allows DCOM server components to be written in diverse programming languages like C++, Java, Object Pascal (Delphi), Visual Basic and even COBOL. As long as a platform supports COM services, DCOM can be used on that platform. DCOM is now heavily used

on the Windows platform. Companies like [Software AG](#) provide COM service implementations through their **EntireX** product for UNIX, Linux and mainframe platforms; [Digital](#) for the Open VMS platform and [Microsoft](#) for Windows and Solaris platforms.

Java/RMI relies on a protocol called the **Java Remote Method Protocol (JRMP)**. Java relies heavily on Java Object Serialization, which allows objects to be marshaled (or transmitted) as a stream. Since Java Object Serialization is specific to Java, both the Java/RMI server object and the client object have to be written in Java. Each Java/RMI Server object defines an interface which can be used to access the server object outside of the current Java Virtual Machine(JVM) and on another machine's JVM. The interface exposes a set of methods which are indicative of the services offered by the server object. For a client to locate a server object for the first time, RMI depends on a naming mechanism called an **RMIRegistry** that runs on the Server machine and holds information about available Server Objects. A Java/RMI client acquires an object reference to a Java/RMI server object by doing a **lookup** for a Server Object reference and invokes methods on the Server Object as if the Java/RMI server object resided in the client's address space. Java/RMI server objects are named using URLs and for a client to acquire a server object reference, it should specify the URL of the server object as you would with the URL to a HTML page. Since Java/RMI relies on Java, it can be used on diverse operating system platforms from mainframes to UNIX boxes to Windows machines to handheld devices as long as there is a Java Virtual Machine (JVM) implementation for that platform. In addition to Javasoft and Microsoft, [a lot of other companies](#) have announced Java Virtual Machine ports.

Comparing Apples to Apples

CORBA 3.0 will add a middleware component model (much like MTS or EJB) to CORBA. Since it is still in a pre-spec stage, we do not know much about how the CORBA middleware component model is going to look like. As of CORBA 2.x, there is no middleware component model that CORBA defines. [Though I would really like to compare MTS, EJB and CORBA 3.0's middleware component model \(whatever it's going to be called\) , I reserve it for a future article \(Click Here...\)](#). By the way, there is a lot of comparison going on between COM and EJB. This is entirely wrong. This is like comparing apples to oranges. The competing technologies are MTS and EJB. Hence, the real comparison should be between **MTS and EJB**.

Application Sample - The StockMarket Server and Client

The StockMarket server reports the stock price of any given symbol. It has a method called `get_price()` to get the stock value of a particular symbol.

I have selected Java as the implementation language for these examples here for three reasons :

1. Java/RMI can only be implemented using Java.
2. Since I am comparing Java/RMI with other object technologies, I would have to implement the DCOM and CORBA objects too in Java.
3. Java is the best language to code CORBA and COM objects in since it keeps the implementation simple, easy to understand, and is most elegant.

Each of these implementations define an **IStockMarket** interface. They expose a `get_price()` method that returns a float value indicating the stock value of the symbol passed in. We list the sources from four files. The first set of files are the IDL and Java files that define the interface and its exposed methods. The second set of files show how the client invokes methods on these interfaces by acquiring references to the server object. The third set of files show the Server object implementations. The fourth set of files show the main program implementations that start up the Remote Server objects for CORBA and Java/RMI. No main program implementation is shown for

DCOM since the JavaReg program takes up the role of invoking the DCOM Server object on the Server machine. This means you have to also ensure that JavaReg is present on your server machine.

The IDL Interface

Whenever a client needs some service from a remote distributed object, it invokes a method implemented by the remote object. The service that the remote distributed object (Server) provides is encapsulated as an object and the remote object's interface is described in an Interface Definition Language (IDL). The interfaces specified in the IDL file serve as a contract between a remote object server and its clients. Clients can thus interact with these remote object servers by invoking methods defined in the IDL.

DCOM - The DCOM IDL file shows that our DCOM server implements a **dual interface**. COM supports both static and dynamic invocation of objects. It is a bit different than how CORBA does through its **Dynamic Invocation Interface (DII)** or Java does with **Reflection**. For the static invocation to work, The Microsoft IDL (MIDL) compiler creates the proxy and stub code when run on the IDL file. These are registered in the systems registry to allow greater flexibility of their use. This is the vtable method of invoking objects. For dynamic invocation to work, COM objects implement an interface called **IDispatch**. As with CORBA or Java/RMI, to allow for dynamic invocation, there has to be some way to describe the object methods and their parameters. **Type libraries** are files that describe the object, and COM provides interfaces, obtained through the **IDispatch** interface, to query an Object's type library. In COM, an object whose methods are dynamically invoked must be written to support **IDispatch**. This is unlike CORBA where any object can be invoked with DII as long as the object information is in the **Implementation Repository**. The DCOM IDL file also associates the **IStockMarket** interface with an object class **StockMarket** as shown in the **coclass** block. Also notice that in DCOM, each interface is assigned a **Universally Unique Identifier (UUID)** called the **Interface ID (IID)**. Similarly, each object class is assigned a unique UUID called a **CLasS ID (CLSID)**. COM gives up on multiple inheritance to provide a binary standard for object implementations. Instead of supporting multiple inheritance, COM uses the notion of an object having multiple interfaces to achieve the same purpose. This also allows for some flexible forms of programming.

DCOM - IDL	CORBA - IDL	Java/RMI - Interface definition
<pre>[uuid(7371a240-2e51-11d0- b4c1-444553540000), version(1.0)] library SimpleStocks { importlib("stdole32.tlb"); [uuid(BC4C0AB0-5A45-11d2- 99C5-00A02414C655), dual] interface IStockMarket : IDispatch { HRESULT get_price([in] BSTR p1, [out, retval] float * rtn);</pre>	<pre>module SimpleStocks { interface StockMarket { float get_price(in string symbol); }; };</pre>	<pre>package SimpleStocks; import java.rmi.*; import java.util.*; public interface StockMarket extends java.rmi.Remote { float get_price(String symbol) throws</pre>

<pre> } [uuid(BC4C0AB3-5A45-11d2- 99C5-00A02414C655),] coclass StockMarket { interface IStockMarket; }; }; </pre>		<pre> RemoteException; } </pre>
File : StockMarketLib.idl	File : StockMarket.idl	File : StockMarket.java

CORBA - Both CORBA and Java/RMI support multiple inheritance at the IDL or interface level. One difference between CORBA (and Java/RMI) IDLs and COM IDLs is that CORBA (and Java/RMI) can specify exceptions in the IDLs while DCOM does not. In CORBA, the IDL compiler generates type information for each method in an interface and stores it in the **Interface Repository (IR)**. A client can thus query the IR to get run-time information about a particular interface and then use that information to create and invoke a method on the remote CORBA server object dynamically through the **Dynamic Invocation Interface (DII)**. Similarly, on the server side, the **Dynamic Skeleton Interface (DSI)** allows a client to invoke an operation of a remote CORBA Server object that has no compile time knowledge of the type of object it is implementing. The CORBA IDL file shows the **StockMarket** interface with the **get_price()** method. When an IDL compiler compiles this IDL file it generates files for stubs and skeletons.

Java/RMI - Notice that unlike the other two, Java/RMI uses a .java file to define its remote interface. This interface will ensure type consistency between the Java/RMI client and the Java/RMI Server Object. Every remotable server object in Java/RMI has to extend the `java.rmi.Remote` class. Similarly, any method that can be remotely invoked in Java/RMI may throw a `java.rmi.RemoteException`. `java.rmi.RemoteException` is the superclass of many more RMI specific exception classes. We define an interface called **StockMarket** which extends the `java.rmi.Remote` class. Also notice that the **get_price()** method throws a `java.rmi.RemoteException`.

How remoting works

To invoke a remote method, the client makes a call to the client proxy. The client side proxy packs the call parameters into a request message and invokes a wire protocol like IIOP(in CORBA) or ORPC(in DCOM) or JRMP(in Java/RMI) to ship the message to the server. At the server side, the wire protocol delivers the message to the server side stub. The server side stub then unpacks the message and calls the actual method on the object. In both CORBA and Java/RMI, the client stub is called the **stub** or **proxy** and the server stub is called **skeleton**. In DCOM, the client stub is referred to as **proxy** and the server stub is referred to as **stub**.

Implementing the Distributed Object Client

DCOM Client - The DCOM client shown below calls into the DCOM server object's methods by first acquiring a pointer to the server object. The `new` keyword here instantiates the **StockMarket** DCOM Server object. This leads the Microsoft JVM to use the `CLSID` to make a `CoCreateInstance()` call. The `IUnknown` pointer returned by `CoCreateInstance()` is then cast to **IStockMarket**, as shown below:

```
IStockMarket market = (IStockMarket)new simplestocks.StockMarket();
```

The cast to **IStockMarket** forces the Microsoft JVM to call the DCOM server object's **QueryInterface()** function to request a pointer to **IStockMarket**. If the interface is not supported, a **ClassCastException** is thrown. Reference Counting is handled automatically in Java/COM and the Microsoft JVM takes up the responsibility of calling **IUnknown::AddRef()** and Java's Garbage Collector automatically calls **IUnknown::Release()**. Once the client acquires a valid pointer to the DCOM server object, it calls into its methods as though it were a local object running in the client's address space.

CORBA Client - The CORBA client will first have to initialize the CORBA ORB by making a call to **ORB.init()**. It then instantiates a CORBA server object by binding to a server object's remote reference. Though both Inprise's VisiBroker and Iona's Orbix have a **bind()** method to bind and obtain a server object reference like,

```
StockMarket market = StockMarketHelper.bind( orb ); // this is Visibroker or Orbix
specific
```

we will use the CORBA Naming Service to do the same thing so that we are compatible with any ORB. We first look up a **NameService** and obtain a CORBA object reference. We use the returned **CORBA.Object** to narrow down to a naming context.

```
NamingContext root = NamingContextHelper.narrow( orb.resolve_initial_references
("NameService") );
```

We now create a **NameComponent** and narrow down to the server object reference by resolving the name in the naming context that was returned to us by the **COSNaming** (CORBA Object Services - Naming) helper classes.

```
NameComponent[] name = new NameComponent[1] ;
name[0] = new NameComponent("NASDAQ","");
```

```
StockMarket market = StockMarketHelper.narrow(root.resolve(name)) ;
```

Once the client has acquired a valid remote object reference to the CORBA server object, it can call into the server object's methods as if the server object resided in the client's address space.

DCOM - Client implementation	CORBA - Client implementation	Java/RMI - Client implementation
<pre>// // // StockMarketClient // // import simplestocks.*; public class StockMarketClient { public static void main (String[] args) { try { IStockMarket market = (IStockMarket)new</pre>	<pre>// // // StockMarketClient // // import org.omg.CORBA.*; import org.omg.CosNaming.*; import SimpleStocks.*; public class StockMarketClient { public static void main(String[] args) { try { ORB orb = ORB.init();</pre>	<pre>// // // StockMarketClient // // import java.rmi.*; import java.rmi.regis import SimpleStocks.* public class StockMarketClient { public static void ma (String[] args)throws Exception { if</pre>

<pre> simplestocks.StockMarket(); System.out.println("The price of MY COMPANY is " + market.get_price ("MY_COMPANY")); } catch (com.ms.com.ComFailException e) { System.out.println("COM Exception:"); System.out.println (e.getHResult()); System.out.println (e.getMessage()); } } } </pre>	<pre> NamingContext root = NamingContextHelper.narrow (orb.resolve_initial_references ("NameService")); NameComponent[] name = new NameComponent[1] ; name[0] = new NameComponent ("NASDAQ", ""); StockMarket market = StockMarketHelper.narrow (root.resolve(name)) ; System.out.println("Price of MY COMPANY is " + market.get_price ("MY_COMPANY")); } catch(SystemException e) { System.err.println(e); } } } </pre>	<pre> (System.getSecurityMan ()== null) { System.setSecurityMan: (new RMISecurityManag) } StockMarket market = (StockMarket)Naming.l ("rmi://localhost/NASI StockMarketHelper System.out.println(" price of MY COMPANY is + market.get_price ("MY_COMPANY")); } } </pre>
File : StockMarketClient.java	File : StockMarketClient.java	File : StockMarketClient.

Java/RMI Client - The Java/RMI client first installs a security manager before doing any remote calls. You do this by making a call to **System.setSecurityManager()**. The **RMISecurityManager** provided by JavaSoft is an attempt by JavaSoft from having you to write your own implementation. However, JavaSoft does not force you to use it's own **RMISecurityManager** - you can write your own security manager and install it if you want to.

Note : It is not mandatory to set a security manager for the use of Java/RMI. The reason to do this is so that the Java/RMI client can handle serialized objects for which the client does not have a corresponding class file in its local CLASSPATH. If the security manager is set to the **RMISecurityManager**, the client can download and instantiate class files from the Java/RMI server. This mechanism is actually fairly important to Java/RMI, as it allows the server to generate subclasses for any **Serializable** object and provide the code to handle these subclasses to the client. It is entirely possible to use Java/RMI without setting the security manager, as long as the client has access to definitions for all objects that might be returned. Java/RMI's ability to handle the passing of any object at any time using Serialization and class file download is possible only because the JVM provides a portable and secure environment for passing around Java byte codes that form the Java executable from which Java objects can be reconstructed at run-time, if required.

The Java/RMI client then instantiates a Java/RMI server object by binding to a server object's remote reference through the call to **Naming.Lookup()**.

```
StockMarket market = (StockMarket)Naming.lookup("rmi://localhost/NASDAQ");
```

Once the client has acquired a valid object reference to the Java/RMI server object, it can call into the server object's methods as if the server object resided in the client's address space.

Implementing the Distributed Object Server

DCOM Server Object - All the classes that are required for Java/COM are defined in the **com.ms.com** package. The DCOM Server object shown below implements the **IStockMarket** interface that we defined in our IDL file. The **StockMarket** class and the **get_price()** method are declared as **public** so that they will be accessible from outside the package. Also notice the **CLSID**

specified and declared as `private`. It is used by COM to instantiate the object through `CoCreateInstance()` when a DCOM client does a `new` remotely. The `get_price()` method is capable of throwing a `ComException`.

CORBA Server Object - All the classes that are required for CORBA are defined in the `org.omg.CORBA` package. The CORBA Server object shown below extends the `_StockMarketImplBase` class that is a skeleton class generated by our CORBA IDL compiler. The `StockMarketImpl` class and the `get_price()` method are declared as public so that they will be accessible from outside the package. The `StockMarketImpl` class implements all the operations declared in our CORBA IDL file. We need to provide a constructor which takes in a name of type `String` for our CORBA object server class since the name of the CORBA Server class has to be passed on to the `_StockMarketImplBase` class object, so that it can be associated with that name with all the CORBA services.

DCOM - Server implementation	CORBA - Server implementation	Java/RMI - Server implementation
<pre>// // // StockMarketServer // // import com.ms.com.*; import simplestocks.*; public class StockMarket implements IStockMarket { private static final String CLSID = "BC4C0AB3-5A45- 11d2-99C5- 00A02414C655"; public float get_price(String symbol) { float price = 0; for(int i = 0; i < symbol.length(); i++) { price += (int) symbol.charAt(i); } price /= 5; return price; }</pre>	<pre>// // // StockMarketServer // // import org.omg.CORBA.*; import SimpleStocks.*; public class StockMarketImpl extends _StockMarketImplBase { public float get_price (String symbol) { float price = 0; for(int i = 0; i < symbol.length(); i++) { price += (int) symbol.charAt(i); } price /= 5; return price; } public StockMarketImpl (String name) { super(name); } }</pre>	<pre>// // // StockMarketServer // // package SimpleStocks; import java.rmi.*; import java.rmi.server.UnicastRemoteObject; public class StockMarketImpl extends UnicastRemoteObject implements StockMarket { public float get_price(String symbol) { float price = 0; for(int i = 0; i < symbol.length(); i++) { price += (int) symbol.charAt(i); } price /= 5; return price; } public StockMarketImpl(String name) throws RemoteException { try { Naming.rebind(name, this); } } catch(Exception e) { System.out.println(e); } }</pre>

<pre> } } </pre>	<pre> } } </pre>	<pre> } } </pre>
File : StockMarket.java	File : StockMarketImpl.java	File : StockMarketImpl.java

Java/RMI Server Object - All the classes that are required for Java/RMI are defined in the `java.rmi` package. The Java/RMI Server object shown extends the `UnicastRemoteObject` class that has all of Java/RMI's remoting methods defined and implements the `StockMarket` interface. The `StockMarketImpl` class and the `get_price()` method are declared as public so that they will be accessible from outside the package. The `StockMarketImpl` class implements all the operations declared in our Java/RMI interface file. We need to provide a constructor which takes in a name of type `string` for our Java/RMI object server class since the name of the Java/RMI Server class is used to establish a **binding** and associate a public name with this Java/RMI Server Object in the **RMIRegistry**. The `get_price()` method is capable of throwing a `RemoteException` since it is a remotable method.

The Server Main Programs

CORBA Server Main - The first thing that has to be done by the main program is to initialize the CORBA ORB using `ORB.init()`. An **Object Adapter (OA)** sits on top of the ORB, and is responsible for connecting the CORBA server object implementation to the CORBA ORB. Object Adapters provide services like generation and interpretation of object references, method invocation, object activation and deactivation, and mapping object references to implementations. You have to initialize either the **Basic Object Adapter(BOA)** or the **Portable Object Adapter (POA)** depending on what your ORB supports. (Note : I use [Inprise's VisiBroker](#) as my CORBA ORB and hence I conform to its implementation requirement where I need to init the BOA). You do this by calling `orb.BOA_init()`. We then create the CORBA Server object with the call

```
StockMarketImpl stockMarketImpl = new StockMarketImpl("NASDAQ");
```

Note that we pass in a name **"NASDAQ"** by which our object is identified by all CORBA services. We then inform the ORB that the Server Object is ready to receive invocations by the statement:

```
boa.obj_is_ready( stockMarketImpl );
```

Since we are using the CORBA Object Service's Naming Service for our clients to connect to us, we will have to bind our server object with a naming service, so that clients would be able to find us. The following code helps us to do that. (Note : This ensures that our code will work with any CORBA ORB. If our clients use the **bind()** method -specific to VisiBroker and Orbix- to connect to the server object we do not need to do this.)

```
org.omg.CORBA.Object object = orb.resolve_initial_references("NameService");
NamingContext root = NamingContextHelper.narrow( object );
NameComponent[] name = new NameComponent[1];
name[0] = new NameComponent("NASDAQ", "");
root.rebind(name, stockMarketImpl);
```

The next statement ensures that our main program sleeps on a daemon thread and does not fall off and exit the program.

```
boa.impl_is_ready();
```

We now enter into an event loop and are in that loop till the main program is shut down.

CORBA - Server Main	Java/RMI - Server Main
<pre>// // // StockMarketServer Main // // import org.omg.CORBA.*; import org.omg.CosNaming.*; import SimpleStocks.*; public class StockMarketServer { public static void main(String[] args) { try { ORB orb = ORB.init(); BOA boa = orb.BOA_init(); StockMarketImpl stockMarketImpl = new StockMarketImpl("NASDAQ"); boa.obj_is_ready(stockMarketImpl); org.omg.CORBA.Object object = orb.resolve_initial_references ("NameService"); NamingContext root = NamingContextHelper.narrow(object) ; NameComponent[] name = new NameComponent [1]; name[0] = new NameComponent("NASDAQ", ""); root.rebind(name, stockMarketImpl); boa.impl_is_ready(); } catch(Exception e) { e.printStackTrace(); } } }</pre>	<pre>// // // StockMarketServer Main // // import java.rmi.*; import java.rmi.server.UnicastRemoteObject; import SimpleStocks.*; public class StockMarketServer { public static void main(String[] args) throws Exception { if(System.getSecurityManager() == null) { System.setSecurityManager(new RMISecurityManager()); } StockMarketImpl stockMarketImpl = new StockMarketImpl("NASDAQ"); } }</pre>
File : StockMarketServer.java	File : StockMarketServer.java

Java/RMI Server Main - The Java/RMI client will first have to install a security manager before doing any remote calls. You do this by making a call to `System.setSecurityManager()`. We then create the Java/RMI Server object with the call

```
StockMarketImpl stockMarketImpl = new StockMarketImpl("NASDAQ");
```

and remain there till we are shut down.

DCOM - Notice that we have not provided a Main program for our DCOM Server implementation. The Java Support in Internet Explorer runs as an **in-process server**, and in-process servers cannot normally be remoted using the Windows NT 4.0 Distributed COM (DCOM). However, it is possible to launch a **"surrogate"**.EXE in its own process that then loads the in-process server.

This surrogate can then be remoted using DCOM, in effect allowing the in-process server to be remoted. You can use **JavaReg's** `/surrogate` option to support remote access to a COM class implemented in Java. When first registering the class, specify the `/surrogate` option on the command line. For example:

```
javareg /register /class:StockMarket /clsid:{FE19E681-508B-11d2-A187-000000000000} /surrogate
```

This adds a **LocalServer32** key to the registry in addition to the usual **InprocServer32** key. The command line under the **LocalServer32** key specifies JavaReg with the `/surrogate` but without the `/register` option.

```
HKEY_CLASSES_ROOT
CLSID
{BC4C0AB3-5A45-11d2-99C5-00A02414C655}
InprocServer32 = msjava.dll
LocalServer32 = javareg /clsid:{BC4C0AB3-5A45-11d2-99C5-00A02414C655} /surrogate
```

This causes JavaReg to act as the surrogate itself. When a remote client requests services from the COM class that you've implemented using Java, JavaReg is invoked. JavaReg then loads the Java Support in Internet Explorer with the specified Java class. (This means that when distributing your Java program, your installation program must install JavaReg along with the Java class.) You can remove the **LocalServer32** key by rerunning JavaReg with the `/class` option, specifying the same class name, but without the `/clsid` or `/surrogate` options

```
javareg /register /class:StockMarket
```

DCOM - Registry File
<pre>REGEDIT4 [HKEY_CLASSES_ROOT\CLSID\{BC4C0AB3-5A45-11d2-99C5-00A02414C655}] @="Java Class: StockMarket" "AppID"="{BC4C0AB3-5A45-11d2-99C5-00A02414C655}" [HKEY_CLASSES_ROOT\CLSID\{BC4C0AB3-5A45-11d2-99C5-00A02414C655}\InprocServer32] @="MSJAVA.DLL" "ThreadingModel"="Both" "JavaClass"="StockMarket" [HKEY_CLASSES_ROOT\CLSID\{BC4C0AB3-5A45-11d2-99C5-00A02414C655}\LocalServer32] @="javareg /clsid:{BC4C0AB3-5A45-11d2-99C5-00A02414C655} /surrogate" [HKEY_CLASSES_ROOT\CLSID\{BC4C0AB3-5A45-11d2-99C5-00A02414C655}\ImplementedCategories] [HKEY_CLASSES_ROOT\CLSID\{BC4C0AB3-5A45-11d2-99C5-00A02414C655}\ImplementedCategories\{BE0975F0-BBDD-11CF-97DF-00AA001F73C1}] [HKEY_CLASSES_ROOT\AppID\{BC4C0AB3-5A45-11d2-99C5-00A02414C655}] @="Java Class: StockMarket"</pre>
File : StockMarket.reg

Conclusion

The architectures of CORBA, DCOM and Java/RMI provide mechanisms for transparent invocation

and accessing of remote distributed objects. Though the mechanisms that they employ to achieve remoting may be different, the approach each of them take is more or less similar.

DCOM	CORBA	Java/RMI
Supports multiple interfaces for objects and uses the <code>QueryInterface()</code> method to navigate among interfaces. This means that a client proxy dynamically loads multiple server stubs in the remoting layer depending on the number of interfaces being used.	Supports multiple inheritance at the interface level	Supports multiple inheritance at the interface level
Every object implements <code>IUnknown</code> .	Every interface inherits from <code>CORBA.Object</code>	Every server object implements <code>java.rmi.Remote</code> (Note : <code>java.rmi.UnicastRemoteObject</code> is merely a convenience class which happens to call <code>UnicastRemoteObject.exportObject(this)</code> in its constructors and provide <code>equals()</code> and <code>hashCode()</code> methods)
Uniquely identifies a remote server object through its interface pointer, which serves as the object handle at run-time.	Uniquely identifies remote server objects through object references(<code>objref</code>), which serves as the object handle at run-time. These object references can be externalized (persistified) into strings which can then be converted back into an <code>objref</code> .	Uniquely identifies remote server objects with the <code>ObjID</code> , which serves as the object handle at run-time. When you <code>.toString()</code> a remote reference, there will be a substring such as "[1db35d7f:d32ec5b8d3:-8000, 0]" which is unique to the remote server object.
Uniquely identifies an interface using the concept of Interface IDs (IID) and uniquely identifies a named implementation of the server object using the concept of Class IDs (CLSID) the mapping of which is found in the registry.	Uniquely identifies an interface using the interface name and uniquely identifies a named implementation of the server object by its mapping to a name in the Implementation Repository	Uniquely identifies an interface using the interface name and uniquely identifies a named implementation of the server object by its mapping to a URL in the Registry
The remote server object reference generation is performed on the wire protocol by the Object Exporter	The remote server object reference generation is performed on the wire protocol by the Object Adapter	The remote server object reference generation is performed by the call to the method <code>UnicastRemoteObject.exportObject(this)</code>
Tasks like object registration, skeleton instantiation etc. are either	The constructor implicitly performs common tasks like object registration, skeleton	The <code>RMIRegistry</code> performs common tasks like object registration through the Naming class.

explicitly performed by the server program or handled dynamically by the COM run-time system.	instantiation etc	<code>UnicastRemoteObject.exportObject (this)</code> method performs skeleton instantiation and it is implicitly called in the object constructor.
Uses the Object Remote Procedure Call(ORPC) as its underlying remoting protocol	Uses the Internet Inter-ORB Protocol(IIOP) as its underlying remoting protocol	Uses the Java Remote Method Protocol(JRMP) as its underlying remoting protocol (at least for now)
When a client object needs to activate a server object, it can do a <code>CoCreateInstance()</code> - (Note:There are other ways that the client can get a server's interface pointer, but we won't go into that here)	When a client object needs to activate a server object, it binds to a naming or a trader service - (Note:There are other ways that the client can get a server reference, but we won't go into that here)	When a client object needs a server object reference, it has to do a <code>lookup()</code> on the remote server object's URL name.
The object handle that the client uses is the interface pointer	The object handle that the client uses is the Object Reference	The object handle that the client uses is the Object Reference
The mapping of Object Name to its Implementation is handled by the Registry	The mapping of Object Name to its Implementation is handled by the Implementation Repository	The mapping of Object Name to its Implementation is handled by the RMIRRegistry
The type information for methods is held in the Type Library	The type information for methods is held in the Interface Repository	Any type information is held by the Object itself which can be queried using Reflection and Introspection
The responsibility of locating an object implementation falls on the Service Control Manager (SCM)	The responsibility of locating an object implementation falls on the Object Request Broker (ORB)	The responsibility of locating an object implementation falls on the Java Virtual Machine (JVM)
The responsibility of activating an object implementation falls on the Service Control Manager (SCM)	The responsibility of locating an object implementation falls on the Object Adapter (OA) - either the Basic Object Adapter (BOA) or the Portable Object Adapter (POA)	The responsibility of activating an object implementation falls on the Java Virtual Machine (JVM)
The client side stub is called a proxy	The client side stub is called a proxy or stub	The client side stub is called a proxy or stub
The server side stub is called stub	The server side stub is called a skeleton	The server side stub is called a skeleton
All parameters passed between the client and server objects are defined in the Interface Definition file. Hence, depending on what the IDL specifies, parameters are passed either by value or by reference.	When passing parameters between the client and the remote server object, all <code>interface</code> types are passed by reference. All other objects are passed by value including highly complex data types	When passing parameters between the client and the remote server object, all objects implementing interfaces extending <code>java.rmi.Remote</code> are passed by remote reference. All other objects are passed by value
Attempts to perform	Does not attempt to	Attempts to perform distributed

distributed garbage collection on the wire by pinging. The DCOM wire protocol uses a Pinging mechanism to garbage collect remote server object references. These are encapsulated in the IOXIDResolver interface.	perform general-purpose distributed garbage collection.	garbage collection of remote server objects using the mechanisms bundled in the JVM
Allows you to define arbitrarily complex structs, discriminated unions and conformant arrays in IDL and pass these as method parameters. Complex types that will cross interface boundaries must be declared in the IDL.	Complex types that will cross interface boundaries must be declared in the IDL	Any <code>Serializable</code> Java object can be passed as a parameter across processes.
Will run on any platform as long as there is a COM Service implementation for that platform (like Software AG's <u>EntireX</u>)	Will run on any platform as long as there is a CORBA ORB implementation for that platform (like Inprise's <u>VisiBroker</u>)	Will run on any platform as long as there is a Java Virtual Machine implementation for that platform (provided by <u>a whole lot of companies</u> in addition to JavaSoft and Microsoft)
Since the specification is at the binary level, diverse programming languages like C++, Java, Object Pascal (Delphi), Visual Basic and even COBOL can be used to code these objects	Since this is just a specification, diverse programming languages can be used to code these objects as long as there are <u>ORB libraries</u> you can use to code in that language	Since it relies heavily on Java Object Serialization, these objects can only be coded in the Java language
Each method call returns a well-defined "flat" structure of type <code>HRESULT</code> , whose bit settings encode the return status. For richer exception handling it uses Error Objects (of type <code>IErrorInfo</code>), and the server object has to implement the <code>ISupportErrorInfo</code> interface.	Exception handling is taken care of by Exception Objects. When a distributed object throws an exception object, the ORB transparently serializes and marshals it across the wire.	Allows throwing exceptions which are then serialized and marshaled across the wire.

The Sources

DCOM	CORBA	Java/RMI
Get the DCOM Visual J++ 1.1 project in a zip file from here <u>StockDCOM.zip</u>	Get the CORBA JBuilder 2 project in a zip file from here <u>StockCORBA.zip</u>	Get the Java/RMI JBuilder 2 project in a zip file from here <u>StockRMI.zip</u>

[File : StockDCOM.zip](#)
[File : StockCORBA.zip](#)
[File : StockRMI.zip](#)

Acknowledgements

A lot of people took the time to review this write-up and offer their valuable comments. I thank all of them. In particular, I would like to thank **Don Box** (DevelopMentor), **Adrian Colley** (JavaSoft-RMI team), **Dr.Doug C. Schmidt** (Washington University), **Rajiv Delwadia**, **Tarak Modi** (both of Clarus Corp) and **Jon Abbey** (Applied Research Laboratories). Thanks to **Salil Deshpande** (Inprise) who in the first place instigated me to write a comparison and for giving me some good feedback and great suggestions.

I am grateful to **Nat Brown** (Microsoft-Author, COM spec.) for taking time to review this write-up and offer his invaluable comments.

I am still receiving excellent comments and suggestions from various people. This article reflects the discussions that I have had with a lot of them. The acknowledgements column is a way of showing my appreciation to all these people who have said '**We Care**'.

Where do we go from here...

The moment we talk of comparisons, we start comparing features one after the other. There is a lot more to these than what I have discussed above. In fact, whole new powerful models are being built based on or around these three remoting paradigms. Since the DCOM, CORBA and Java/RMI remoting models are the basis for a lot of other paradigms built on top of them, the moment we talk of these other paradigms, it is the start of a totally new and different comparison.

I could have gone on comparing **MTS and EJB next**, or **DNA and J2EE**, or **MSMQ and JMS**, or **Jini and Universal Plug and Play**... But they are entirely different comparisons. I have to stop at some point and I decided that I stop right here, giving myself an opportunity to write about other technology comparisons in future articles.

Author Bibliography

Gopalan Suresh Raj is a Software Architect, Developer and an active Author. He is contributing author to a couple of books "Enterprise Java Computing-Applications and Architecture" and "The Awesome Power of JavaBeans". His expertise spans enterprise component architectures and distributed object computing. Visit him at his **Web Cornucopia**® site (<http://gsraj.tripod.com/>) or mail him at gopalan@gmx.com.



[click here to go to](#)

[My **Advanced Java / J2EE Tutorial** HomePage...](#)

[click here to go to](#)

[My **COM+ / DNA Tutorial** HomePage...](#)

[click here to go to](#)

[My **CORBA Tutorial** HomePage...](#)