

**CPSC 689-608:  
Distributed Computing Frameworks  
Fall 2003**

**Project 2 –  
To Compare .Net, J2EE and CORBA  
on Remote Object Access**

**Zhibin Mai  
Email: [zbmai@cs.tamu.edu](mailto:zbmai@cs.tamu.edu)  
Due Date: November 3, 2003**

## Table of Contents

---

1. Introduction .....	3
2. Object passing .....	4
3. Object binding and locating .....	6
4. Object lifetime management .....	10
5. Reference .....	12

## 1. Introduction

Distributed systems require that computations running in different address spaces, potentially on different hosts, be able to communicate. For a basic communication mechanism, sockets are flexible and sufficient for general communication. However, 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.

Benefited from the convenient RPC mechanism, all current distributed computing frameworks have supported RPC mechanism based remote object access. Built on distributed computing framework, the programmer has the illusion of calling a local object, when in fact the arguments of the call are packaged up and shipped off to the remote target of the call. In this paper, we discuss remote object access in terms of object passing, object locating, object lifetime management based on the current three well-known distributed frameworks (.Net, J2EE and CORBA). Since both .Net and J2EE frameworks are consist of different kinds of applications and protocols, when discussing remote object access, we will discuss .NET Remoting and J2EE Enterprise Java Bean (EJB) solutions.

**CORBA IIOP** -- CORBA is a competing specification by the Object Management Group, which is a group of middleware vendors. CORBA is language neutral as well and is more widely implemented across different platforms than COM. But there are incompatibilities between different vendor implementations. CORBA uses a protocol called IIOP (Internet Inter-ORB Protocol) to communicate between different systems.

**Java RMI** -- The Java platform's remote method invocation system described in this specification 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. To communicate between different systems, Java provides another protocol, called RMI over IIOP. Using RMI over IIOP, Java applications can talk to any CORBA compatible applications.

**.NET Remoting** -- Microsoft® .NET remoting provides a framework that allows objects to interact with one another across application domains. The framework provides a number of services, including activation and lifetime support, as well as communication channels responsible for transporting messages to and from remote applications. If you need interoperation between heterogeneous systems, the use of .NET Remoting is never an interoperability solution, and web services approach that uses open standards (SOAP, XML, HTTP) is the right choice. For homogeneous systems where all participants are CLR managed, .NET Remoting may be the right choice. Because web service is an open standard, Sun Microsystems also supports this standard in its latest SDK and EJB 2.1 specification. As a comparison purpose paper, I won't discuss web service support for remote object access in .NET framework.

## 2. Object passing

One of the main objectives of any distributed framework is to provide the necessary infrastructure that hides the complexities of calling methods on remote objects and returning results. Generally speaking, there are two kinds of object passing models:

- Pass-By-Reference
  - Server returns reference to the object
  - Client calls the server with the reference
  - Server executes methods and returns the value
- Pass-By-Value
  - Server returns copy, reference to local copy
  - Client calls the local copy with the reference
  - Client executes method and returns the value

Since Pass-By-Value can improve the performance most when object is small and object is accessed multiple times by client, Pass-By-Value is an important approach to improve the system performance in remote object access.

### In CORBA IIOP:

Determination of whether a parameter is to be passed by value or reference is made by examining the parameter's formal type (i.e., the signature of the operation it is being passed to). If it is a value type then it is passed by value. If it is an ordinary interface then it is passed by reference (the case today for all CORBA objects). In the case of abstract interfaces, the determination is made at runtime.

When an instance of the value type is passed as a parameter to an operation of a remote interface, the effect in all cases shall be as if an independent copy of the instance is instantiated in the receiving context. If the receiving context does not currently hold an implementation with which to reconstruct the original type, an algorithm is used to find such an implementation.

For example:

```
valuetype EmployeeRecord {// note this is not a CORBA::Object  
// state definition  
    private string name;  
    private string email;  
    private string SSN;  
// initializer  
    factory init(in string name, in string SSN);  
};
```

### In Java RMI:

An argument to, or a return value from, a remote object can be any object that is serializable. This includes primitive types, remote objects, and local objects that implement the `java.io.Serializable` interface.

A local 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. So, when a local object is

passed as an argument or return value in a remote method invocation, the content of the non-remote object is copied before invoking the call on the remote object. When a non-remote object is returned from a remote method invocation, a new object is created in the calling virtual machine. Classes, for parameters or return values, which are not available locally, are downloaded dynamically by the RMI system.

When passing an exported remote object 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.

### **In .NET Remoting:**

Any object outside the application domain of the caller should be considered remote, even if the objects are executing on the same machine. Inside the application domain, all objects are passed by reference while primitive data types are passed by value. Since local object references are only valid inside the application domain where they are created, they cannot be passed to or returned from remote method calls in that form. All local objects that have to cross the application domain boundary have to be passed by value and should be marked with the [serializable] custom attribute, or they have to implement the ISerializable interface. When the object is passed as a parameter, the framework serializes the object and transports it to the destination application domain, where the object will be reconstructed. Local objects that cannot be serialized cannot be passed to a different application domain and are therefore nonremotable.

Any object can be changed into a remote object by deriving it from MarshalByRefObject. When a client activates a remote object, it receives a proxy to the remote object. All operations on this proxy are appropriately indirected to enable the remoting infrastructure to intercept and forward the calls appropriately. This indirection does have some impact on performance, but the JIT compiler and execution engine (EE) have been optimized to prevent unnecessary performance penalties when the proxy and remote object reside in the same application domain. In cases where the proxy and remote objects are in different application domains, all method call parameters on the stack are converted into messages and transported to the remote application domain, where the messages are turned back into a stack frame and the method call is invoked. The same procedure is used for returning results from the method call.

### **Summary:**

- Both Object-Pass-By-Reference and Object-Pass-By-Value are supported in the three frameworks. Object passed by value in Java RMI and .NET Remoting must implement serializable. Object passed by value in CORBA 3.0 must define as valuetype. All the three framework allows developer to define whether particular object to be passed by reference or value.

In many cases it may be useful to defer the determination of whether an object is passed by reference or by value until runtime. In CORBA 3.0, the IDL abstract interface provides this capability. In this sense, CORBA 3.0 provides more powerful

mechanism to support object passed by reference or by value than Java RMI and .NET Remoting do.

### 3. Object Binding and Locating

A common issue of distributed systems is the allocation of the logical objects of a system to physical nodes. In particular there is often a need for "location transparency", that is, the ability to map a single specification to a number of different physical environments without requiring changes to the code. Naming service provides a solution to object location transparency in distributed systems.

#### In CORBA IIOP:

The Naming Service is a standard service for CORBA applications. The Naming Service allows you to associate abstract names with CORBA objects and allows client to find those objects by looking up the corresponding names. Its role is to allow a name to be bound to an object and to allow that object to be found subsequently by resolving that name within the Naming Service. A server that holds an object reference can register it with the Naming Service, giving it a name that can be used by other components of the system to subsequently find the object. Even though every object in an ORB has a unique reference ID accessing services, from a clients' point of view, it is much easier if there is some directory listing of services, so that the client could use a descriptive name to access the object's service.

The following example first defines an interface, HelloApp by using the Interface Description Language (IDL).

```
module HelloApp {
    interface hello
    {
        string sayHello();
    };
};
```

It then defines an implementation of this interface, helloServant.

```
class helloServant extends HelloApp._helloImplBase {
    public String sayHello() {
        return "\nHello world !!\n" + new java.util.Date();
    }
}
```

Next, it creates an instance of helloServant and binds it to the directory, assigning it the name "Hello".

```
// create and initialize the ORB
ORB orb = ORB.init(args, null);

// get reference to rootpoa & activate the POAManager
POA rootpoa = POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
rootpoa.the_POAManager().activate();

// create servant and register it with the ORB
HelloImpl helloImpl = new HelloImpl();
```

```

helloImpl.setORB(orb);

// get object reference from the servant
org.omg.CORBA.Object ref = rootpoa.servant_to_reference(helloImpl);
Hello href = HelloHelper.narrow(ref);

// get the root naming context
org.omg.CORBA.Object objRef = orb.resolve_initial_references("NameService");
// Use NamingContextExt which is part of the Interoperable
// Naming Service (INS) specification.
NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);

// bind the Object Reference in Naming
String name = "Hello";
NameComponent path[] = ncRef.to_name( name );
ncRef.rebind(path, href);

```

After the object has been bound in the directory, an application can look it up by using the following code.

```

// create and initialize the ORB
ORB orb = ORB.init(args, null);

// get the root naming context
org.omg.CORBA.Object objRef = orb.resolve_initial_references("NameService");

// Use NamingContextExt instead of NamingContext. This is
// part of the Interoperable naming Service.
NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);

// resolve the Object Reference in Naming
String name = "Hello";
helloImpl = HelloHelper.narrow(ncRef.resolve_str(name));

//Print reply "Hello World!!" from HelloServer
System.out.println(helloImpl.sayHello());

```

### **In Java RMI:**

In Java RMI, it uses rmiregister as basic naming service to resolve the object and object name. When a Server Object binds a remote reference to the rmiregistry, the rmiregistry tries to locate the stub in the local system classpath. If the stub class is found, then it binds the object reference as it is. To lookup an object already bound to an rmiregistry, we use the static method lookup of the Naming class. The signature of this method is RemoteObject lookup(String ObjectName). This means that we will have to type cast the RemoteObject to the type we want, which will be an object that implements our interface.

The following example first defines an interface:

```

package HelloApp.hello;

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

```

```
public interface Hello extends Remote {
    String sayHello() throws RemoteException;
}
```

It then defines an implementation of this interface, HelloImpl.

```
public class HelloImpl extends UnicastRemoteObject implements Hello {
    public HelloImpl() throws RemoteException {
        super();
    }

    public String sayHello() {
        return "\nHello world !!\n" + new java.util.Date();
    }
}
```

Next, it creates an instance of *HelloImpl* and binds it to the directory, assigning it the name "Hello".

```
// Create and install a security manager
if (System.getSecurityManager() == null) {
    System.setSecurityManager(new RMISecurityManager());
}

HelloImpl obj = new HelloImpl();

// Bind this object instance to the name "HelloServer"
Naming.rebind("//localhost:1099/Hello", obj);
```

After the object has been bound in the directory, an application can look it up by using the following code.

```
Hello helloImpl = (Hello)Naming.lookup("//localhost:1099" + "/Hello");
System.out.println(helloImpl.sayHello());
```

### **In .NET Remoting:**

All remote objects have to be registered with the Remoting Framework before clients can access them. The most important pieces of information required for registration is the type of the object, the URI where it will be deployed, the activation requirements for managing the object lifetime and the channels that can be used to connect to this object.

During this registration process, the Framework is provided with all the information required to activate and manage the lifetime of the object. Object registration is normally done by a hosting application that starts up, registers one or more channels with ChannelServices, registers one or more remote objects with RemotingServices and then waits until it is terminated.

The following example first implement a service:

```
public class helloRemoteObject : MarshalByRefObject {
    // Method sayHello
```

```

    public String sayHello() {
        return "Hello World";
    }
}

```

Next, it creates an instance of *helloRemoteObject* and binds it to the directory, assigning it the name "Hello".

```

//create and register a channel
ChannelServices.RegisterChannel(new HttpChannel(6666));

//register the helloRemoteObject for remoting:
RemotingConfiguration.RegisterWellKnownServiceType(typeof(helloRemoteObject),
"helloRemoteObject", WellKnownObjectMode.Singleton);

```

After the object has been bound in the directory, an application can look it up by using the following code.

```

// create and register a channel
ChannelServices.RegisterChannel(new HttpChannel());

//Connect to helloRemoteObject
RemotingConfiguration.RegisterWellKnownClientType(typeof(helloRemoteObject),
"http://localhost:6666/ helloRemoteObject ");

//create an instance
helloRemoteObject hello = new helloRemoteObject();
hello.sayHello();

```

### **Summary:**

Both .NET Remoting and Java RMI in object binding and locating (or called name service) is very simple and clean in terms of user interface. It is a little complicated to bind and locate a CORBA object. So in terms of simple coding, .NET Remoting and Java RMI are better than CORBA on object binding and locating.

In Both .NET Remoting and Java RMI, a remote object registers to a single naming server and the naming server can not communicate each other to share object naming and location information. In CORBA, a remote can register to multiple naming servers. The the ORBs (with naming service) can communicate and share information, and coordinate who will provide the service (object reference) when receiving request from the client. Using the ORB bridge, the ORB can communicate across multiple domains and network segments, as service routing. CORBA provides a mechanism for the stubs so that the stubs in the client either use a location-transparent IPC mechanism or directly access a location service to establish communication with the implementations if there is at least one ORB existing in the same network segment as the client. So in terms of naming information sharing and objecting auto-locating, CORBA provides better solution than .NET Remoting and Java RMI.

#### **4. Object lifetime management**

In a distributed system, just as in the local system, it is desirable to automatically delete those remote objects that are no longer referenced by any client. This frees the programmer from needing to keep track of the remote objects' clients so that it can terminate appropriately.

A reliable and light-weight remote object lifetime management mechanism is necessary to guarantee remote object referential integrity.

##### **In CORBA IIOP:**

The CORBA lifetime management model decouples the lifetime of the clients from the lifetime of the active (in-memory) representation of the persistent server object. The CORBA model allows clients to maintain references to CORBA server objects even when the clients are no longer running. Server objects can deactivate and remove themselves from memory whenever they become idle. This behavior allows resources (such as memory and networking addresses) to be released from active use for long-lived (but generally idle) services. Objects can be created and destroyed. From a client's point of view, there is no special mechanism for creating or destroying an object. Objects are created and destroyed as an outcome of issuing requests. The outcome of object creation is revealed to the client in the form of an object reference that denotes the new object.

##### **In Java RMI:**

JAVA RMI uses a reference counting garbage collection algorithm similar to Modula-3's Network Objects. To accomplish reference-counting garbage collection, the RMI runtime keeps track of all live references within each Java virtual machine. When a live reference enters a Java virtual machine, its reference count is incremented. The first reference to an object sends a "referenced" message to the server for the object. As live references are found to be unreferenced in the local virtual machine, the count is decremented. When the last reference has been discarded, an unreferenced message is sent to the server.

When a remote object is not referenced by any client, the RMI runtime refers to it using a weak reference. The weak reference allows the Java virtual machine's garbage collector to discard the object if no other local references to the object exist. The distributed garbage collection algorithm interacts with the local Java virtual machine's garbage collector in the usual ways by holding normal or weak references to objects. As long as a local reference to a remote object exists, it cannot be garbage collected and it can be passed in remote calls or returned to clients.

Note that if a network partition exists between a client and a remote server object, it is possible that premature collection of the remote object will occur (since the transport might believe that the client crashed). Because of the possibility of premature collection, remote references cannot guarantee referential integrity; in other words, it is always possible that a remote reference may in fact not refer to an existing object. An attempt to use such a reference will generate a RemoteException which must be handled by the application.

### **In .NET Remoting:**

NET Remoting provides a rich mechanism for managing the lifetime of remote objects. If the remote object does not hold any state (for example, as is the case with a `SingleCall` object), then it is no need to be concerned in any way with this process. The Remoting infrastructure is just to do what it does and remote objects will be garbage collected when needed. If remote objects are holding state, either server-activated Singletons or client-activated objects, lifetime management process, object leasing, is anticipated.

The mechanism provided by Remoting for object management is based on the principle of leasing: You never own an object, you just borrow it, and as long as you keep up the payments you get to continue using it. This process is further described below. First, though, a few words on how the COM world deals with object cleanup. DCOM uses a combination of pinging and reference counting methods to determine whether objects are still running. This is both error prone and network-bandwidth intensive. The whole principle of reference counting was at worst never fully understood and at best fragile. There were (and still are) a number of simple rules that had to be applied for reference counting to work. The `IUnknown` interface for a COM object includes `AddRef` and `Release` methods, which need to be called by the developer at the appropriate times. Sometimes programmers got this wrong, resulting in objects not being removed and in associated memory leaks.

In contrast, the Remoting leased-based lifetime management system uses a combination of leases, sponsors, and a lease manager. Each application domain contains a Lease Manager, which holds references to a lease object for each Singleton or client-activated object within its domain. Each lease can have zero or more associated sponsors that are capable of renewing the lease when the Lease Manager determines that the lease has expired. This lease functionality is provided by the Remoting infrastructure through the `ILease` interface and is acquired through the call to `InitializeLifetimeService`, which we have already seen above. The `ILease` interface defines a number of properties that are used to manage an objects lifetime:

- *InitialLeaseTime* -- Determines how long the lease is initially valid.
- *RenewOnCallTime* -- After each method call the lease is renewed for this time unit.
- *SponsorshipTimeout* -- How long Remoting will wait after sponsor lease expiry notification.
- *CurrentLeaseTime* -- How long until the lease expires (read only).

When a lease expires the Lease Manager will notify any lease sponsors asking if they wish to renew the lease. If none do, then the associated object references will be released. Sponsors are objects that can renew leases for remote objects. To become a sponsor your class must derive from `MarshalByRefObject` and implement the `ISponsor` interface. A lease can have many sponsors. A sponsor may participate in many leases.

For example:

```
public class Foo : MarshalByRefObject {
    public override Object InitializeLifetimeService()
    {
        ILease lease = (ILease)base.InitializeLifetimeService();
    }
}
```

```

    if (lease.CurrentState == LeaseState.Initial) {
        lease.InitialLeaseTime = TimeSpan.FromMinutes(1);
        lease.SponsorshipTimeout = TimeSpan.FromMinutes(2);
        lease.RenewOnCallTime = TimeSpan.FromSeconds(2);
    }
    return lease;
}
}

```

### **Summary:**

In CORBA lifetime management model, it does not require pinging or maintaining reference counts. The disadvantage is that it requires the application to explicitly decide when an object has been made obsolete and its references should become invalid. Deciding when to get rid of an object is not a trivial task

In JAVA RMI lifetime management model, a reference-counting garbage collection mechanism is implemented. To accomplish reference-counting garbage collection, the RMI runtime keeps track of all live references within each Java virtual machine. The advantage is that it does not require the application to explicitly decide when an object has been made obsolete. The disadvantage is that this is both error prone and network-bandwidth intensive.

In .NET Remoting lifetime management model, it uses combination of leases, sponsors, and a lease manager. The ILease interface defines a number of properties that are used to manage an objects lifetime. The advantages are that the referential integrity can be easily to maintain and Singleton Objects can be easily supported. The disadvantage is the Lease Manager might become single point of the failure.

## **5. Reference:**

- [1] OMG, *Common Object Request Broker Architecture (CORBA) Specification, v3.0*, <http://www.omg.org>
- [2] Sun Microsystems, *Java™ Remote Method Invocation Specification—Java™ 2 SDK, Standard Edition, v1.4*, <http://java.sun.com/j2se/1.4.2/docs/guide/rmi/index.html>
- [3] .NET Remoting, <http://msdn.microsoft.com/library/en-us/dndotnet/html/introremoting.asp>