OBJECTSPACE VOYAGER
CORE TECHNOLOGY

THE

AGENT ORB

FOR JAVA

Voyager and RMI Comparison

This paper compares Version 1.0 of the ObjectSpace Voyager™ Core Technology (Voyager) with JavaSoft's RMI product included as part of the JDK 1.1.4.

Voyager is a full-featured, intuitive object request broker (ORB) that supports mobile objects and autonomous agents. Voyager also includes services for persistence, scalable group communication, and basic directory service. This core Voyager technology is free for most commercial uses.

RMI—which stands for remote method invocation—is the first commercially available system to attempt to more closely mirror the Java language. As part of the JDK 1.1.4, RMI is also free for most commercial use.

This text presents a high-level comparison of Voyager and RMI. A table summarizing each product's features is presented first, followed by a more detailed explanation of each feature.

OBJECTSPACE

© 1997 ObjectSpace, Inc. • All rights reserved • Version 1.0 • December 1997

Contents

Overview 1
Remote-Enabling a Class 3
Constructing a Remote Object 4
Exporting a Named Remote Object 5
Connecting to a Named Object 6
Exception Handling 7
Executing a Remote Static Method 8
Object Mobility 9
Agents 10
Distributed Persistence 11
Scalability 12
Multicast Messaging 13
Distributed Events 14
Publish/Subscribe 15
Federated Directory Service 16
Message Types 17
Evolution of Remote References 18
Garbage Collection 19
Applet Connectivity 20
Network Class Loading 21
Product Size 22
Performance 23
Stubs and Skeletons 24

Overview

Because ORBs such as CORBA and DCOM are designed to work across languages, they are unable to leverage the full power of Java™. CORBA users know first-hand that mapping .idl constructs into the Java language is clumsy and unnatural at best.

JavaSoft addresses this problem with the RMI product, included as part of JDK 1.1.4. By making a free, basic ORB available, JavaSoft sets the baseline for subsequent competitive efforts.

ObjectSpace responds with the ObjectSpace Voyager Core Technology, an advanced, 100% Java ORB designed as a Java-centric distributed computing platform. We believe Voyager surpasses RMI's ORB capabilities in terms of ease of use and features without sacrificing performance.

The following table summarizes Voyager and RMI features. Each feature is described in detail after this summary.

Feature	Voyager 1.0	RMI 1.1.4
Cost	Free to most developers	Free to most developers
Constructing a remote object	Supported via regular Java syntax	Not supported
Remote-enabling a class	Requires one simple step	Requires five tedious steps
Exporting a named object	Seamlessly integrated	Requires external registry
Connecting to a named object	Seamlessly integrated	Requires external registry
Exception handling	Explicit or run-time exceptions allowed	Only explicit exceptions allowed
Executing a remote static method	Supported via regular Java syntax	Not supported
Object mobility	Fully supported, even when objects are active	Not supported
Agents	Can execute as they move and can move themselves	Not supported
Distributed persistence	▪ Supported with a transparent, default database ▪ Requires no modification of your classes ▪ Autoload and autoflush supported	Not supported
Scalability	Global-scale, fault-tolerant, persistent, distributed computing supported with innovative Space™ architecture	No similar features
Multicast messaging	▪ Fully supported ▪ 100% nonintrusive	Not supported
Distributed events	▪ Compliant with JavaBeans™ ▪ 100% nonintrusive	Not supported
Publish/subscribe	▪ Messages and events supported ▪ 100% nonintrusive	Not supported
Federated directory service	▪ Fully supported ▪ Integrated with the persistence subsystem	Not supported
Message types	Synchronous, one-way, one-way multicast, and future messages supported via smart, lightweight messenger agents	Only synchronous messages supported
Evolution of classes	Supported	Not supported
Garbage collection	Lease- and time-based garbage collection supported	Only lease-based garbage collection supported
Applet connectivity	Unrestricted	Restricted
Network class loading	Built-in	Requires Web server
Product size	About 270KB	About 180KB
Performance	See “Performance” on page 23 of this document.	
Stubs and skeletons	80% less support code generated than RMI	400% more support code generated than Voyager

Remote-Enabling a Class

Using RMI

The following steps are required to enable an existing class for remote method invocation in RMI.

1. Create an interface that extends Remote and redeclare every public nonstatic function.
2. Add RemoteException to every method signature.
3. Modify the existing class to implement the interface and extend UnicastRemoteObject.
4. Add throws RemoteException to every method implementation.
5. Modify each method that takes or returns an implementation to use a remote interface instead.

Note that constructors do not appear in the interface. The RMI documentation recommends that you do not override hashCode(), equals(), or toString() in your implementation because each is defined to work correctly in a superclass of UnicastRemoteObject.

Another restriction is that instances of classes defined as `Remote` cannot be passed by value. Therefore, if you define a RemoteVector class and pass it as an argument, a remote reference to the vector is always passed rather than a copy of the vector. This is not always the desired effect.

Using Voyager

To enable an existing class for remote method invocation in Voyager, simply run vcc on the .class or .java file to produce a virtual version of the class. The name of the virtual class is equal to the original class name preceded by the letter V. The virtual class implements the same interfaces as the original class, as well as a superset of its constructors and methods. You need not modify the original class in any way—you do not even need access to its source code. This means that you can effortlessly enable any third-party library class for remote method invocation. For example, use the following command to remote enable a JDK Vector and create VVector:

```
vcc java.util.Vector
```

Constructing a Remote Object

Using RMI

RMI does not directly support remote object construction. However, you can emulate this feature using the following steps.

1. In advance, construct a custom factory object on the server.
2. Pass the name of the class and the URL of the codebase to the factory object.

Then, to create a remote instance, program the factory object to perform the following actions.

1. Use RMIClassLoader to load the class across the network
2. Use reflection to construct a remote instance using the default constructor

This method of constructing a remote object works with the default constructor only. To construct an object using parameters is considerably more work.

Using Voyager

Voyager is designed to make remote construction effortless. Use regular Java construction syntax to construct a remote object and supply a destination address as an additional argument. For example, use the following commands to create a remote JDK `Vector` in a program on tokyo:8000 and add two elements:

```
VVector vector = new VVector( "tokyo:8000" );
vector.addElement( new Integer( 42 ) );
vector.addElement( "voyager" );
```

Exporting a Named Remote Object

Using RMI

RMI includes a registry service for binding an object to an alias, then retrieving the object via its alias. To create an object in a server and then export the object for use by remote clients, the following steps are required.

1. Start an rmiregistry process on the server.
2. Create the object on the server.
3. Bind the object to an alias.

Using Voyager

Voyager includes an integrated directory service that allows you to associate an object with an alias without the need for a separate registry process. To create an object in a program and then export the object for use by remote clients, simply construct the object in the program and supply its alias during construction. For example, to create a remote JDK Vector with alias MyVector in a program on tokyo:9000, use the following command:

```
VVector vector = new VVector( "tokyo:9000/MyVector" );
```

Voyager's integrated directory service supports most simple use cases. Voyager also includes a federated directory service, described on page 16.

Connecting to a Named Object

Using RMI

To connect to an object using its alias in RMI, first perform a lookup using the object's name, and then cast the returned remote reference to the correct type.

Using Voyager

To connect to an object using its alias in Voyager, use the static method `VObject.forObjectAt()`; then, cast the returned virtual reference to the correct type. For example, use the following commands to connect to an existing remote JDK `Vector` with alias `MyVector` in a program on `tokyo:9000`:

```
VVector vector = (VVector) VObject.forObjectAt( "tokyo:9000/MyVector" );
```

Exception Handling

Using RMI

RMI approaches exception handling by forcing developers to program safely. Every remote method invocation in a try..catch block must be explicitly wrapped, even if the communicating objects are on the same machine or on the same virtual machine.

Using Voyager

Voyager supports a superset of the RMI exception handling strategy.

By default, every function in a virtual class throws a VoyagerException that must be wrapped with an explicit try..catch block. This exception does not have to be declared in the original class.

To process a class that implements an interface whose methods do not explicitly throw a VoyagerException (such as java.util.EventListener or java.applet), you can use vcc with the -r option. This causes each VoyagerException to be wrapped and rethrown as a VoyagerRuntimeException.

Unlike RMI, all remote Voyager exceptions are automatically annotated with useful trace information to assist in remote debugging.

Executing a Remote Static Method

Using RMI

Because RMI deals exclusively with interfaces, RMI has no support for executing a remote static method.

Using Voyager

To execute a remote static method in Voyager, invoke the method and supply the destination address as an additional argument. For example, use the following command to execute the static function Account.getNumberOfAccounts() in a program on tokyo:9000:

```
int count = VAccount.getNumberOfAccounts( "tokyo:9000" );
```

Object Mobility

Using RMI

RMI does not support object mobility. Once created, an object remains on the same machine for its lifetime.

Using Voyager

Voyager allows any serializable object to move to a new program, even while the object is receiving remote messages. By default, messages sent to the object's old location are automatically forwarded to the new location. The new location is attached to the return value so that subsequent messages are delivered directly to the object at its new location.

Use the following commands to connect to the `Vector` with alias MyVector in a program on tokyo:9000, and then move MyVector to a program on dallas:8000:

```
VVector vector = (VVector) VObject.forObjectAt( "tokyo:9000/MyVector" );
vector.moveTo( "dallas:8000" );
```

Agents

Using RMI

RMI does not support mobile agents.

Using Voyager

Voyager allows a developer to create—in minutes—an agent that continues to execute as it moves between programs. An agent can independently move to a remote object and get a local reference to the object to communicate using high-speed, local messaging. An agent can move to an object even if the object is moving.

Distributed Persistence

Using RMI

RMI does not include seamless integration for persistence of objects.

Using Voyager

Voyager includes seamless support for object persistence. In many cases, you can persist an object without modifying its source in any way.

Every Voyager program can be associated with a database. The type of database can vary from program to program and is transparent to a Voyager programmer. Voyager includes a high-performance object storage system called VoyagerDb and will soon include bindings that work with most popular relational and object databases as well.

To save an object to its program's database, send saveNow() to the object. This method causes a copy of the object to be written to the database, overwriting the previous copy if one exists. If the program is shut down and then restarted, the persistent object remains in the database. An attempt to communicate with the persistent object causes the object to be immediately reloaded from the database.

If a persistent object is moved from one program to another, the persistent copy of the object is automatically removed from the source program's database and added to the

To conserve memory, you can use one of the flush() family of methods to flush a persistent object from memory to a database. A subsequent attempt to communicate with a flushed persistent object causes the object to be immediately reloaded from the database.

By default, Voyager's database system persists Java classes that are loaded into a program across a network so they need not be reloaded when the program is restarted.

Scalability

Using RMI

RMI does not include a scalable architecture for multicast messaging, distributed events, or publish/subscribe.

Using Voyager

Many distributed systems like those listed below require features for communicating with groups of objects.

- Stock quote systems use a distributed event feature to send stock price events to customers around the world.
- A voting system uses a distributed messaging feature (multicast messaging) to send messages to voters around the world and ask them for their views on a particular matter.
- News services use a distributed publish/subscribe feature to ensure each broadcast is received only by readers interested in the topic of the broadcast.

Most traditional systems use a single repeater object to replicate a message or event to each object in the target group. This approach works fine if few objects reside in the target group, but does not scale well when large numbers of objects are involved.

Voyager uses an innovative architecture for message and event replication called Space™ that can scale to global proportions. Clusters of objects in the target group are stored in local groups called subspaces. The subspaces are linked together across a network to form a larger logical group, or Space. When a message or event is sent into one of the subspaces in a Space, the message or event is cloned to each of the other subspaces in the Space before being delivered to every object in every subspace. This results in a very rapid, parallel fanout of the message or event to every object in the Space. A special mechanism in each subspace ensures that no message or event is accidentally processed more than once, regardless of how the subspaces are linked together.

Voyager's multicast messaging, distributed events, and publish/subscribe features all use and benefit from the same underlying Space architecture.

Multicast Messaging

Using RMI

RMI does not support multicast messaging, although the RMI user documentation suggests a technique that might be used to implement it. The documentation also mentions that a future version of RMI will include MulticastRemoteObject, an alternative to UnicastRemoteObject. That is, if you want to create a class that you can multicast messages to, you must extend MulticastRemoteObject instead of UnicastRemoteObject. We believe this technique is an example of poor object-oriented design—the implementation of an object is coupled with how the object is used.

Using Voyager

Voyager includes seamless support for large-scale multicast messaging that does not require modifying your classes in any way. To perform multicast messaging, add objects to a Space, establish a virtual reference to the Space, and send the Space a message as if you were sending it to a single object. The message is propagated in a fault-tolerant and parallel fashion to every object in the Space.

Use the commands below to create two sports fans, and then add them to the sports Space:

```
VSportsFan fan1 = new VSportsFan( "localhost" );
VSportsFan fan2 = new VSportsFan( "localhost" );
sports.add( fan1 );
sports.add( fan2 );
```

To send every sports fan in the sports Space a one-way score() message, use the following commands:

```
VSportsFan fans = new VSportsFan( space ); //attach to space
fans.score( "bulls", 40, "lakers", 50 ); //multicast
```

Distributed Events

Using RMI

RMI does not support distributed events.

Using Voyager

Voyager includes seamless support for large-scale, distributed JavaBeans™ events. To send an event to a group of objects, first process the event listener class using vcc. Then add the group of objects to a Space and attach the appropriate virtual event listener to the Space. Finally, add the virtual event listener to the event source. When an event is sent to the virtual event listener, the event is sent to every object in the Space that implements the appropriate event listener interface. The Voyager events system allows you to send any JavaBeans event to a network of distributed listeners *without modifying the bean in any way*.

For example, assume that the SportsFan class implements a NewsEventListener interface that accepts a NewsEvent via the newsFlash() method. Use the following commands to create two sports fans and then add them to the sports Space:

```
VSportsFan fan1 = new VSportsFan( "localhost" );
VSportsFan fan2 = new VSportsFan( "localhost" );
sports.add( fan1 );
sports.add( fan2 );
```

Use the following commands to send every sports fan in the sports Space a NewsEvent:

```
VNewsEventListener fans = new VNewsEventListener( space );
NewsEvent event = new NewsEvent( "the cowboys win!" );
fans.newsFlash( event ); //send event to every fan in space
```

Publish/Subscribe

Using RMI

RMI does not support publish/subscribe capabilities.

Using Voyager

Voyager includes seamless support for large-scale, distributed publish/subscribe of messages and events. To send a message or event to all objects in a Space that are interested in a particular subject, use a OneWayMulticast message with a selector. All objects in the Space that are registered subscribers of the selected subject receive the message or event. To register an object with a particular subject, use Voyager's built-in property mechanism. The Voyager publish/subscribe feature is 100% nonintrusive, supports wildcards, and does not require modifying the communicating objects in any way.

For example, to create a sports fan and register its interest in the Bulls and Mavericks scores being broadcast in the sports Space, use the following commands:

```
VSportsFan fan = new VSportsFan( "localhost" );
fan.addProperty( Subscription.SUBSCRIBE, "scores.bulls" );
fan.addProperty( Subscription.SUBSCRIBE, "scores.mavericks" );
sports.add( fan );
```

Use the commands below to publish Bulls and Lakers scores in the sports Space:

```
VSportsFan fans = new VSportsFan( space ); //attach to space
Subscription subscription = new Subscription();
subscription.addSubject( "scores.bulls" );
subscription.addSubject( "scores.lakers" );
Messenger m = new OneWayMulticast( subscription );
fans.score( "bulls", 40, "lakers", 50, m ); //publish
```

Federated Directory Service

Using RMI

RMI does not support federated directories.

Using Voyager

Voyager includes a directory service that allows you to create and connect network directories to form a large, federated directory service. You can associate an object with a hierarchical name, such as sports/basketball/lakers or chemistry/symbols/calcium. The federated directory service is fully integrated with Voyager's persistence subsystem.

Message Types

Using RMI

RMI supports synchronous messages only.

Using Voyager

Voyager supports four different message types: synchronous, one-way, one-way multicast, and future messages.

By default, Voyager messages are synchronous (the sender blocks until the message completes and the return value is received). Voyager also supports one-way messages (the sender discards the result), future messages (the sender returns immediately with a placeholder that allows the result to be retrieved later), and one-way multicast messages (the message is sent to all objects in a Space). You can also send a one-way multicast message to only certain objects in a group by using a selector.

Voyager method invocations are performed by smart messengers, which are lightweight, active objects rather than passive data structures. Smart messengers can route themselves, resend themselves, take actions on failures, and so on. Source code licensees can create customized messengers without modifying the core system.

Voyager also supports dynamic message creation. You can set a messenger's signature and arguments at run time before a message is sent. This is a powerful feature that can be used for many purposes; for example, you can easily create a scheduler that sends a user-supplied message to any kind of object at a particular point in time.

Evolution of Remote References

Using RMI

When the RMI rmic compiler generates client and server stubs, it associates hardcoded numbers with each function; that is, foo() might be associated with 1, bar() might be associated with 2, and so on. These numbers are used by the client stub to remotely activate a function via the server stub. Thus, if you deploy a class remotely and then modify its class, you might be unable to use the new client stub to communicate with older instances of the class. The only recourse at this point is to shut down and restart the entire system. RMI, therefore, does not support evolution of remote references in a network environment.

Using Voyager

When used to generate client stubs, the Voyager vcc utility embeds method signatures in the virtual class instead of hardcoded numbers. These signatures are used by the reflection mechanism on the server to find and execute a remote method. Therefore, if you deploy a class remotely and then modify its class, you can continue to execute methods on older instances of the class using the new virtual class. Voyager, therefore, supports smooth evolution of remote references in a network environment.

Garbage Collection

Using RMI

RMI has a lease-based garbage collection system. When a client obtains a reference to a remote object, the client is granted a lease that must be renewed periodically to prevent the remote object from being garbage-collected. A remote object is garbage-collected when all its leases expire.

Using Voyager

Voyager offers a superset of the RMI garbage collection system that supports both lease-based and time-based garbage collection. An object's life span can be defined based on a specific length of time or a particular point in time. The object is garbage-collected at the end of its life span. Time-based garbage collection is often used to create roaming agents that automatically self-destruct in a few days. Voyager's garbage collection system is fully integrated with its support for persistence thus correctly garbage-collects persistent objects.

Applet Connectivity

Using RMI

Most browsers allow an applet to open a socket connection only to its server. This means that in the absence of any higher-level routing mechanism, an applet can only communicate with objects located on the same server. RMI has no additional routing mechanism thus is subject to this limitation.

Using Voyager

Voyager includes a lightweight software router that allows both applet-to-applet and applet-to-program connectivity. An object inside an applet can communicate with another object no matter where each object resides. That is, an object can communicate with objects in another applet, whether the applet is on the same server or on a different server, firewalls permitting.

Network Class Loading

Using RMI

For an RMI program to act as a source for classes that can be loaded across the network, the program must be running an HTTP server. Although reasonable when the client is an applet accessed from a Web site, this requirement is clumsy in other cases. For example, to build an intranet system in which the server program is running an internal Windows NT system, you would have to install an HTTP server on the Windows NT machine, even though the machine does not actually service the Web. JavaSoft supplies developers with a mini-Web server to overcome this difficulty.

Using Voyager

Voyager programs can transmit classes to other Voyager programs using regular socket connections. Therefore, you need not install any additional software to write programs that make full use of Java's code mobility, which results in much simpler deployment of Voyager programs.

Product Size

Using RMI

RMI's total class file size is approximately 180KB. This includes all primary RMI .class files, the registry, and the distributed garbage collection system, but excludes the HTTP firewall support.

Using Voyager

Voyager's total class file size is approximately 270KB. This includes the entire Voyager system—the integrated directory service, the distributed garbage collection system, smart messengers, mobility, and agent support.

Performance

The table below lists a few benchmarks that compare RMI and Voyager performance on remote method calls between objects on the same virtual machine and between objects on different virtual machines. Each function was defined to take a specific kind of argument and to perform no operation. The benchmarks were performed on a 150Mhz Tecra laptop with 80MB of RAM. Times are in milliseconds per function call. The following interface definition was used.

```
package benchmarks;

import java.util.Vector;
import java.rmi.*;

public interface IServer extends Remote
  {
  public void noArguments() throws RemoteException;
  public int twoInts( int a, int b ) throws RemoteException;
  public int vectorIntegers( Vector integers ) throws RemoteException;
  public int vectorStrings( Vector strings ) throws RemoteException;
  }
```

	No Arguments	Two Integers	Vector of 100 Integers	Vector of 100 Strings
Same Virtual Machine				
RMI 1.1.4	2.1	2.3	437.83	193.48
Voyager 1.0	0.2	0.5	0.3	0.4
Different Virtual Machines				
RMI 1.1.4	2.1	3.01	436.02	190.28
Voyager 1.0	3.0	3.21	117.87	193.98

Stubs and Skeletons

RMI and Voyager each use a utility to generate custom code for remote method invocation. RMI generates both client code and server code, whereas Voyager generates client code only. This section shows the code that each product generates for the twoInts() method of the following interface.

```
public interface IServer extends Remote
 {
 public void noArguments() throws RemoteException;
 public int twoInts( int a, int b ) throws RemoteException;
 public int vectorIntegers( Vector integers ) throws RemoteException;
 public int vectorStrings( Vector strings ) throws RemoteException;
 }
```

As the following code examples demonstrate, Voyager typically emits 80 percent less support code than RMI.

Using RMI

Stub code generated by twoInts():

```
//Implementation of twoInts
  public int twoInts(int $_int_1, int $_int_2) throws java.rmi.RemoteException {
    int opnum = 1;
    java.rmi.server.RemoteRef sub = ref;
    java.rmi.server.RemoteCall call =
    sub.newCall((java.rmi.server.RemoteObject)this, operations, opnum, interfaceHash);
    try {
      java.io.ObjectOutput out = call.getOutputStream();
      out.writeInt($_int_1);
      out.writeInt($_int_2);
    } catch (java.io.IOException ex) {
      throw new java.rmi.MarshalException("Error marshaling arguments", ex);
    };
    try {
      sub.invoke(call);
    } catch (java.rmi.RemoteException ex) {
      throw ex;
    } catch (java.lang.Exception ex) {
      throw new java.rmi.UnexpectedException("Unexpected exception", ex);
    };
    int $result;
    try {
      java.io.ObjectInput in = call.getInputStream();
      $result = in.readInt();
    } catch (java.io.IOException ex) {
      throw new java.rmi.UnmarshalException("Error unmarshaling return", ex);
    } catch (Exception ex) {
      throw new java.rmi.UnexpectedException("Unexpected exception", ex);
    } finally {
      sub.done(call);
    }
    return $result;
  }
```

Skeleton code generated by twoInts():

```
case 1: { // twoInts
        int $_int_1;
        int $_int_2;
        try {
          java.io.ObjectInput in = call.getInputStream();
          $_int_1 = in.readInt();
          $_int_2 = in.readInt();
        } catch (java.io.IOException ex) {
          throw new java.rmi.UnmarshalException("Error unmarshaling arguments", ex);
        } finally {
          call.releaseInputStream();
        };
        int $result = server.twoInts($_int_1, $_int_2);
        try {
          java.io.ObjectOutput out = call.getResultStream(true);
          out.writeInt($result);
        } catch (java.io.IOException ex) {
          throw new java.rmi.MarshalException("Error marshaling return", ex);
        };
        break;
        }
```

Using Voyager

Stub code generated by twoInts(), default version:

```
public Result twoInts( int arg1, int arg2, Messenger __messenger )
 {
 __messenger.writeInt( arg1 );
 __messenger.writeInt( arg2 );
 return __instanceMethod( __messenger, instance1 );
 }
```

Stub code generated by twoInts(), smart messenger version:

```
public int twoInts( int arg1, int arg2 )
 {
 return twoInts( arg1, arg2, new Sync() ).readIntRuntime();
 }
```

There is no skeleton code for twoInts() because Voyager does not require skeleton code on the server side.

For additional technical information on ObjectSpace products and programs or for information on how to order and evaluate ObjectSpace technology, contact us today!

14850 Quorum Drive, Suite 500
Dallas TX 75240

972.726.4100
1.800.OBJECT.1
Fax: 972.715.9099

E-mail: sales@objectspace.com
Web: www.objectspace.com

Dallas • Austin • Chicago •
San Francisco • Washington DC

Java is a trademark of Sun Microsystems.
ObjectSpace Voyager and Space are trademarks of ObjectSpace, Inc.
All other trademarks are the property of their respective companies.