OBJECTSPACE
VOYAGER

The Agent ORB for Java

Core Technology User Guide

Version 1.0.0

Part No. DOC-3000-00

Software Version 1.0.0

First Edition

Printed in the United States of America

Table of Contents

Preface

Purpose and Audience

This user guide is designed to educate users on the basic operation of the ObjectSpace Voyager™ Core Technology (Voyager). This user guide is intended for those with a basic working knowledge of Java™. It explains the power and simplicity of Voyager and benefits anyone currently developing network Java applications.

How to Use This Document

This user guide is organized into three parts. Part 1, “ObjectSpace Voyager Overview,” presents a high-level explanation of Voyager, using a mix of text, diagrams, and sample code. Part 2, “ObjectSpace Voyager ORB,” describes Voyager’s ORB functionality in detail, including a description of each Voyager class. Part 3, “ObjectSpace Voyager Services,” describes the extra services that come with Voyager.

The chapters of this user guide contain several sections, each of which describe a Voyager feature using a comprehensive set of examples. We recommend that you read each chapter sequentially, because examples presented later in a chapter often build on previous examples in the chapter.

Notational Conventions

The following conventions are used in this guide:

Italic text	Used for document titles and user-supplied variables in command examples
Arial Narrow font	Used for keyboard key names and for field names and user-supplied text in GUI windows
`Courier New font`	Used to identify source code, file names, and directory names
`Bold Courier New font`	Used to identify code output
>	Used to indicate a user prompt

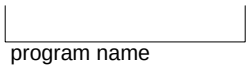
Commands that the user types in a screen and resulting output displayed to the screen are presented in a window, as shown below.

```
>Command typed in screen
Resulting output displayed to screen
```

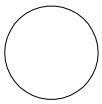
Sometimes, all output from a command does not display to the screen at once. When additional output is presented in a window, the original command and output text are shaded gray, and new output is presented in bold. For example:

```
>Command typed in screen
Resulting output displayed to screen
More output displayed to screen
>
```

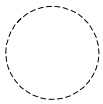
The following key can be used for the diagrams in this manual.



Voyager-enabled program



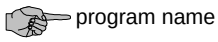
Object



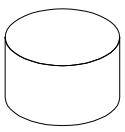
Virtual object

Agent

Virtual agent



Forwarder object



Database

Message

Forwarded message

Related Documents

Voyager comes complete with online API documentation. Access this material by opening a browser on `\voyager1.0.0\doc\index.html`.

Information about how to install Voyager is provided in the online *Voyager Installation Guide*.

For the most up-to-date information about Voyager and other ObjectSpace products, visit the ObjectSpace Web site at www.objectspace.com.

Directory Layout

Following is a high-level map of the Voyager directory structure:

`\voyager1.0.0`

`\bin`	Voyager `vcc` and `voyager` scripts
`\doc`	Documentation files
`\examples`	Example files
`\lib`	`voyager1.0.0.jar` file (Voyager `.class` files)

Example Programs

Voyager examples shown in this user guide are located in the `\voyager1.0.0\examples` directory, which also contains `vmake.bat` and `vmake` script files for most of the examples. This user guide presents each example in the following order:

1. First, the commands used to prepare the example program for execution are presented. Commands that generate virtual classes and compile Java source code belong in this category.
2. Next, the commands used to run the example program are presented, followed by the program output.
3. Finally, the files associated with the example program are shown.

Each file associated with an example program is preceded by an example heading. As shown below, the example heading contains the type of file and the path that describes the file's location in the Voyager directory structure. Applets and applications are the main example programs. Classes and interfaces are the class and interface definitions associated with the main programs. HTML listings are the HTML source files for the applets.

Applet voyager1.0.0\examples\applets\chat\ChatApplet.java

Application voyager1.0.0\examples\shopper\Build.java

Class voyager1.0.0\examples\orb\Customer.java

HTML voyager1.0.0\examples\applets\calculator\Calculator.html

Interface voyager1.0.0\examples\applets\shopper\IMall.java

Technical Support

ObjectSpace offers its customers the following support options.

Free Support

Our Web site contains useful information about ObjectSpace products. Visit www.objectspace.com and select the appropriate product to view answers to frequently asked questions (FAQ), read about known problems, review technical white papers, or download new versions of software. Several forums, such as discussion lists, newsgroups, and notification mailing lists, are also available for select products.

Annual Technical Support Contracts

ObjectSpace offers commercial quality technical support for the Voyager Core Technology download. This support is sold as an annual contract on a per-product, per-user basis. An annual support contract offers several important benefits.

- Full-service problem resolution, via telephone (in North America only), fax, or e-mail, for one year from the date of purchase.
- All major and minor product upgrades for one year from the date of purchase. Major upgrades include significant product enhancements and complete new documentation.
- Access to members-only sections and downloads on the ObjectSpace Web site.
- Participation in selective restricted programs for early code access. These programs are available only to supported customers.

Customers that purchase in volume receive discounts on technical support. Support contracts are valid for one year from date of purchase.

Our technical support staff are available from 9:00 a.m. to 5:00 p.m. central time and can be contacted as follows.

Internet mail:	support@objectspace.com
Phone:	972.726.4500
Fax:	972.715.9099
Web site:	www.objectspace.com

When you contact our support staff, please have the following items of information available.

- Your support ID number. This number is assigned to your group when you purchase or renew your support contract. This number is on the support certificate sent to you at the time you purchase support.
- The version number of the product you are using.
- The name and version of the operating system and compiler you are using.
- A small example program that demonstrates the problem, if appropriate.
- The preferred time and method of reaching you, in case we cannot solve the problem right away.

Problem Reports and Suggestions

ObjectSpace welcomes problem reports and suggestions for improving Voyager. All significant contributions to Voyager will be acknowledged in future releases. Please send your valuable feedback to voyager@objectspace.com. ObjectSpace does not guarantee responses to customers without support contracts.

Product Updates

To be notified of new Voyager releases and other Voyager-related news items, do one of the following.

- Join the automatic e-mail notification service when you download Voyager. This happens automatically when you download Voyager from the ObjectSpace Web site unless you explicitly deny automatic notification or enter an incorrect e-mail address.
- Request this service now by sending e-mail to voyager@objectspace.com.

Acknowledgments

Many people contributed to the success of the Voyager product. Thanks to everyone who contributed to the Voyager Interest Group. With the awareness that someone important might be left out, and a sincere apology if that is the case, special thanks go to the following people:

Gad Barnea	Scott Ganyo
David Brown	Johan Gilliusson
Tilo Christ	Jay Gindin
Randy Darling	Mike Jenkins
Julio Cesar de Almeida Maia	Carey Jung
Richard Deadman	Dmitri Kondratiev
Court Demas	Steve MacDonald
Erik Eilerts	Jaco van der Merwe
Steven Farley	Matthias Oelmann

Additionally, Voyager could not have succeeded without the cooperation and help from all ObjectSpace personnel. Thank you, SpacePeople!

Part 1

ObjectSpace Voyager Overview

1

Introduction

Part 1 is comprised of three chapters that present a high-level overview of the ObjectSpace Voyager™ Core Technology (Voyager).

- Read this chapter for a summary of the Part 1 chapters, a quick overview of Voyager and Voyager's future integration with CORBA, and a Voyager class hierarchy listing.
- Read Chapter 2, "Concepts," for a description of Voyager's primary concepts related to traditional and agent-enhanced distributed computing.
- Read Chapter 3, "Guided Tour," for an example project that quickly demonstrates the power and simplicity of Voyager. Chapter 3 contains all steps necessary to build an agent-enhanced system, complete with full, annotated source code.

What Is Voyager?

ObjectSpace Voyager is the ObjectSpace product line designed to help developers produce high-impact distributed systems quickly. Voyager is 100% Java™ and is designed to use the Java language object model. Voyager allows you to use regular message syntax to construct remote objects, send them messages, and move them between programs. This reduces learning curves, minimizes maintenance, and, most importantly, speeds your time to market for new advanced systems. Voyager's architecture is designed to provide developers full flexibility and powerful expansion paths.

The root of the Voyager product line is the ObjectSpace Voyager Core Technology. This product contains the core features and architecture of the platform, including a full-featured, intuitive object request broker (ORB) with support for mobile objects and autonomous agents. Also in the core package are services for persistence, scalable group communication, and basic directory services. The ObjectSpace Voyager Core Technology is everything you need to get started building high-impact systems in Java today.

As the industry evolves, other companies providing distributed technologies struggle as they try to adapt to the new Java language. These companies are required to adapt earlier object models to fit Java. This results in a series of compromises that together have a dramatic impact on time to market and development costs. Voyager, on the other hand, is developed to use the Java language as its fundamental interface.

One of Java's primary differentiations is the ability to load classes into a virtual machine at run time. This capability enables infrastructures to use mobile objects and autonomous agents as

another tool for building distributed systems. Adding this capability to older distributed technologies is often impractical and results in difficult-to-use infrastructures. Voyager provides seamless support for mobile objects and autonomous agents.

Future CORBA Integration

Complete bidirectional CORBA integration is scheduled for release as part of the Voyager Core Technology 1.1.0. This additional Java package allows Voyager to be used as a CORBA 2 client or server. You will be able to generate a Voyager remote interface from any IDL file. You will be able to use this interface to communicate with any Voyager or CORBA server. Without modifying the code, you will be able to export any Java class as a CORBA server in seconds, automatically generating IDL for use by CORBA implementations.

As part of the Voyager Core Technology, the CORBA integration will also be free for most commercial use.

Voyager Class Hierarchy

The following lists outline the hierarchy of Voyager's public interfaces, classes, and exceptions. A brief description is provided for each class.

Interfaces

`Db` ..... Database interface.

`Messenger` ..... Smart messenger interface.

`ObjectListener` ..... Interface that allows object event notification.

`Result` ..... Result interface.

`ResultListener` ..... Interface that allows result event notification.

`Selector` ..... Unary predicate for applying a boolean pass/fail test to an object.

`SystemListener` ..... Interface that allows system event notification.

`VoyagerClassLoader` ..... Interface that allows developers to plug in custom class loading semantics.

`VoyagerServerSocket` ..... Interface that allows developers to define custom server socket semantics.

`VoyagerSocket` ..... Interface that allows developers to define custom client socket semantics.

`VoyagerSocketFactory` ..... Interface that allows developers to plug in custom socket creation semantics.

Classes

`Agent` ..... Root of all agents.

- `Subspace` ..... Building block of a Space.

`DbInfo` ..... Information about a persistent object.

`Directory` ..... Building block of the federated naming service.

`Inverter` ..... Selector that returns `true` if an object fails a set of criteria.

`Monitor` ..... Listens to all `ObjectEvent` events.

`OneWayResult` ..... Result of `OneWay` or `OneWayMulticast` message.

`SmartMessenger` ..... Root of all smart messengers.

- `Future` ..... Returns immediately. Delivers return value later.
- `Sync` ..... Returns when return value is received.
- `OneWay` ..... Returns immediately. Does not deliver return value.
- `OneWayMulticast` ..... Returns immediately. Broadcasts to many objects.

`Subscription` ..... Selector for `OneWayMulticast`. Used in publish/subscribe.

`Token` ..... Holds a string or GUID.

`UnicastResult` ..... Result of synchronous or future messages.

`VObject` ..... Root of all virtual references.

- `VAgent` ..... Root of all virtual references to agents.

`Voyager` ..... Represents a Voyager program.

`VoyagerDb` ..... Lightweight object storage engine.

`java.util.EventObject` ..... Root of all Java events.

- `ObjectEvent` ..... Generated by objects when object-level activities occur.
  - `SubspaceEvent` ..... Generated by subspaces when objects are added or removed and when neighbors are connected or disconnected.
- `ResultEvent` ..... Generated by a `Result` when reply is received.
- `SystemEvent` ..... Generated by Voyager when system-level activities occur.

`java.lang.SecurityManager` ..... Default security manager.

- `VoyagerSecurityManager` ..... Voyager-specific security manager.

Exceptions

`java.lang.Exception` ..... Root of all Java exceptions.

- `VoyagerException` ..... Root of all Voyager exceptions.
  - `AmbiguousAliasException` .... More than one object had the same alias.
  - `ClassLoadingException` ..... Voyager could not load a class.
  - `DbException` ..... Database input/output error occurred.
  - `DeadlockException` ..... Condition that causes deadlock occurred.
  - `DirectoryException` ..... Illegal directory path was supplied.
  - `InvalidAddressException` .... Illegal address was supplied.
  - `MethodNotFoundException` .... Method was not found.
  - `ObjectNotFoundException` .... Object was not found.
  - `ResultException` ..... Attempted to read from a `OneWayResult`.
  - `StartupException` ..... Program startup error occurred.
  - `TimeoutException` ..... Activity could not be completed in specified time.
  - `TransportException` ..... Object or message could not be transported.
  - `UserException` ..... Wraps a user exception as a `VoyagerException`.

`java.lang.RuntimeException` ..... Root of all Java run-time exceptions.

- `VoyagerRuntimeException` ..... Wraps a `VoyagerException` as a run-time exception.

2

Concepts

This section describes the concepts behind the ObjectSpace Voyager™ Core Technology (Voyager) architecture, using a mix of text, example code, and drawings.

Objects

Objects are the building blocks of all Voyager programs. An object is a software component that has a well-defined set of public functions and encapsulates data. The following object is an instance of the class `Store` with a public function to accept new stock.

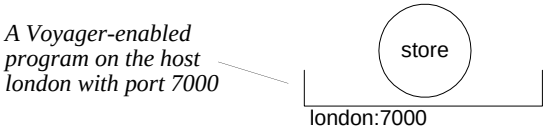
```
Store store = new Store();
store.stock( "widget", 43 );
```

Voyager-Enabled Programs

When a Voyager-enabled program starts, it automatically spawns threads that provide timing services, perform distributed garbage collection, and accept network traffic. Each Voyager-enabled program has a network address consisting of its host name and a communications port number, which is an integer unique to the host.

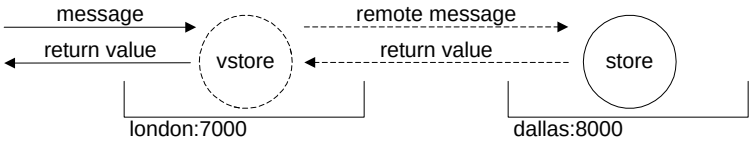
Port numbers are usually randomly allocated to programs. This is sufficient for clients communicating with remote objects and for creating and launching agents into a network. However, if a program will be addressed by other programs, you can assign a well-known port number to the program at startup.

```
Voyager.startup( 7000 ); // assign port number 7000 to this program
Store store = new Store();
```



Remote-Enabled Classes and Virtual References

A class is remote-enabled if its instances can be created outside the local address space of a program and if these instances can receive messages as if they were local. Voyager allows an object to communicate with an instance of a remote-enabled class via a special object called a *virtual reference*. When messages are sent to a virtual reference, the virtual reference forwards the messages to the instance of the remote-enabled class. If a message has a return value, the target object sends the return value to the virtual reference, which returns this message to the sender.



After remote-enabling a class, you can:

- Construct instances remotely, even if the class code does not exist on the remote machine.
- Send messages to remote instances using regular Java™ syntax.
- Connect to existing remote instances in other programs.
- Move objects to other programs, even if the class code is not already in the destination program.
- Persist the object.

Generating a Remote-Enabled Class

Use Voyager's `vcc` utility to generate a remote-enabled class from an existing class. The `vcc` utility reads a `.class` or `.java` file and generates a new *virtual class*. The virtual class contains a superset of the original class functions and allows function calls to occur even when objects are remote or moving.

The virtual class name is `V` plus the original class name. For example, if the file `Store.java` contains the source code for class `Store`, the compiled class file is `Store.class`. You can remote-enable the `Store` class by running `vcc` on either `Store.java` or `Store.class` to create a new, virtual class named `VStore`.

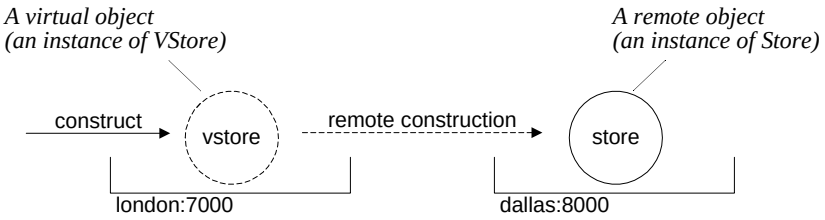
For more detailed information about remote enabling, refer to "Remote-Enabling a Class" on page 46.

Constructing a Remote Object

After remote-enabling a class, you can use the class constructors of the resulting virtual class to create a remote instance of the original class. The remote instance can reside in your current program or a different program, and a virtual reference to the remote instance is created in your current program.

To construct a remote instance of a class, give the virtual class constructor the address of the destination program where the remote instance will reside. If the original class code for the remote instance does not exist in the destination program, the Voyager network class loader automatically loads the original class code into the destination program.

```
Voyager.startup( 7000 );
VStore vstore = new VStore( "dallas:8000/Acme" ); // alias is Acme
```

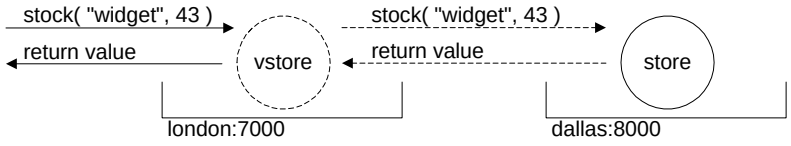


When a remote object is constructed, it is automatically assigned a 16-byte globally unique identifier (GUID), which uniquely identifies the object across all programs worldwide. Optionally, you can assign an alias to an object during construction. The GUID or the optional alias can be used to locate or connect to the object at a later point in time. This directory service is a basic Voyager feature. Voyager also includes an advanced federated directory service for more complex directory requirements. Refer to Chapter 16, “Federated Directory Service,” for information.

Sending a Message to a Remote Object

When a message is sent to a virtual reference, the virtual reference forwards the message to its associated remote object. If the message requires a return value, the remote object passes the return value to the virtual reference, which forwards it to the sender. Similarly, if the remote object throws an exception, the exception is caught and passed back to the virtual reference, which throws it to the caller.

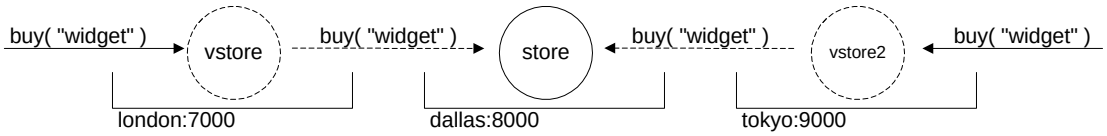
```
vstore.stock( "widget", 43 );
```



Connecting to an Existing Remote Object

A remote object can be referenced by any number of virtual references. To create a new virtual reference and associate it with an existing remote object, supply the address of the program where the existing remote object currently resides and the alias of the remote object to the static `VObject.forObjectAt()` method.

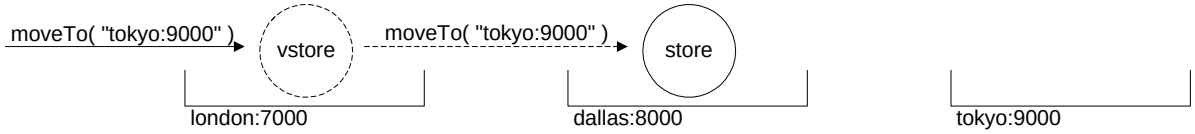
```
// connect using alias
Voyager.startup( 9000 );
VStore vstore2 = (VStore) VObject.forObjectAt( "dallas:8000/Acme" );
int price = vstore2.buy( "widget" );
```



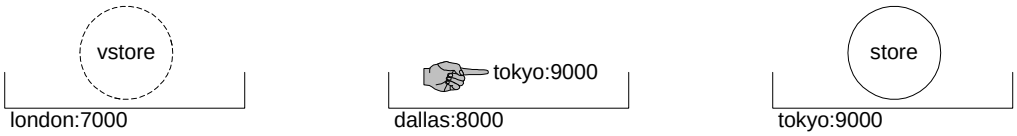
Mobility

You can move an object from one program to another by sending the `moveTo()` message to the object via its virtual reference. Supply the address of the destination program as a parameter.

```
vstore.moveTo( "tokyo:9000" );
```

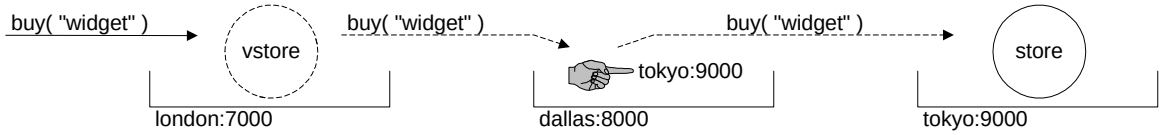


The object waits until all pending messages are processed and then moves to the specified program, leaving behind a forwarder to forward messages and future connection requests.

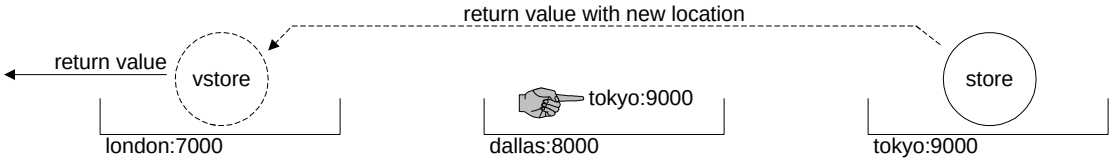


You can send a message to an object even if the object has moved from one program to another. Simply send the message to the object at its last known address. When the message cannot locate its target object, the message searches for a forwarder. If the message locates a forwarder representing the object, the forwarder sends the message to the object's new location.

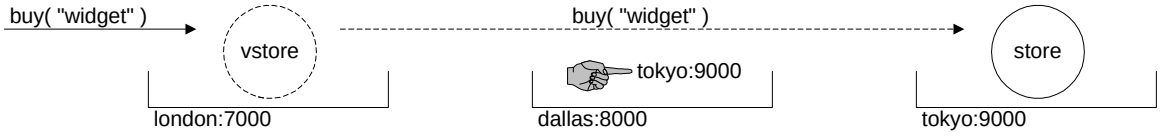
```
int price = vstore.buy( "widget" );
```



The return value is tagged with the remote object's new location, so the virtual reference can update its knowledge of the remote object's location.



Subsequent messages are sent directly to the remote object at its new location, bypassing the forwarder.



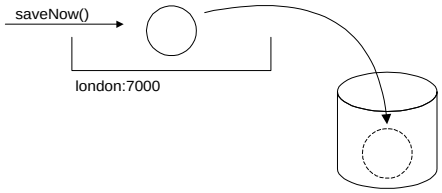
Persistence

A persistent object has a backup copy in a database. A persistent object is automatically recovered if its program is unexpectedly terminated or if it is flushed from memory to the database to make room for other objects. Voyager includes seamless support for object persistence. In many cases, you can persist an object without modifying its source.

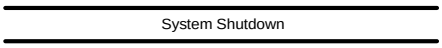
Each Voyager program can be associated with a database. The type of database can vary from program to program and is transparent to a Voyager programmer. Voyager includes a high-performance object storage system called `VoyagerDb`, but Voyager works with most popular relational and object databases.

To save an object to the program's database, send `saveNow()` to the object. This method writes a copy of the object to the database, overwriting any previous copy. If the program is shut down and then restarted, the persistent objects are left in the database. Any attempt to communicate with a persistent object causes the object to be reloaded from the database.

See Chapter 14, “Database-Independent Persistence,” for more details about Voyager persistence.

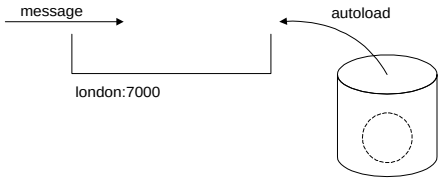


The saveNow() message writes a copy of the persistent object to the database.

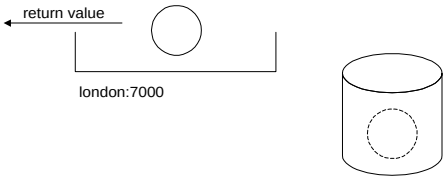


System is shut down temporarily.

When the system restarts, the copy of the object remains in the database, but the actual object is not immediately restored in its original location.

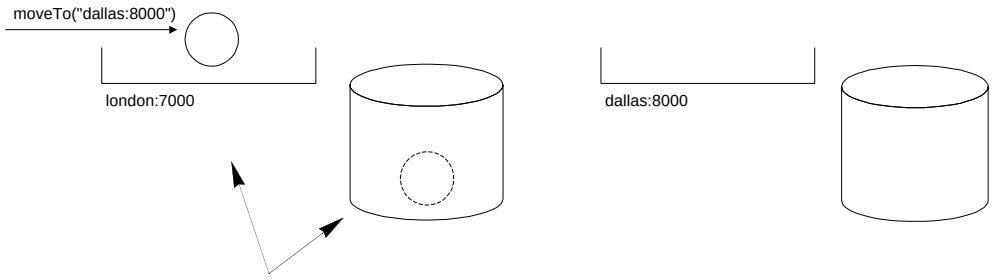


When a message arrives, a copy of the persistent object is autoloaded into memory.

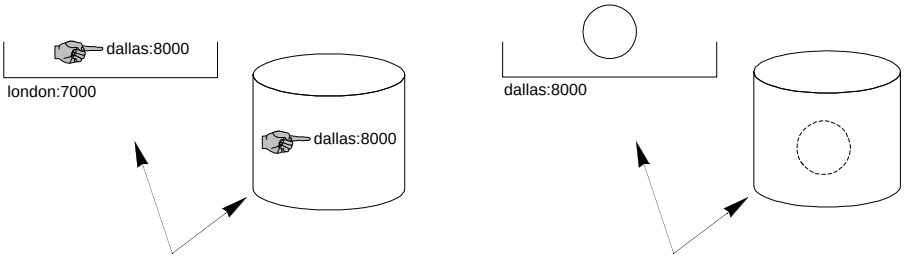


The object is restored, the message is delivered to the object, and a return value is sent.

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



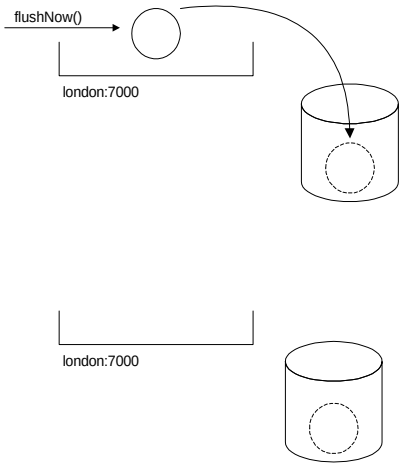
A persistent object in london:7000 and its copy in the database.



Forwarders are left behind in london:7000 and in the london:7000 database.

The persistent object is moved to dallas:8000, and a copy of the persistent object is entered into the dallas:8000 database.

You can conserve memory by using one of the `flush()` family of methods to remove a persistent object from memory and store it in a database. Any subsequent attempt to communicate with a flushed persistent object reloads the object from the database.



The flushNow() message writes a copy of the persistent object to the database and causes the actual object to be garbage-collected from its original location. The object is restored the first time a message is sent to it.

By default, Voyager's database system automatically persists Java classes loaded into a program across a network, thus avoiding a reload of these classes when the program is restarted.

Agents

An agent is a special object type. Although there is no single definition of an agent, all definitions agree that an agent has autonomy. An autonomous object can be programmed to satisfy one or more goals, even if the object moves and loses contact with its creator.

Some definitions state that an agent has mobility as well as autonomy. Mobility is the ability to move independently from one device to another on a network. Voyager agents are both autonomous and mobile. They have all the same features as simple objects—they can be assigned aliases, have virtual references, communicate with remote objects, and so on.

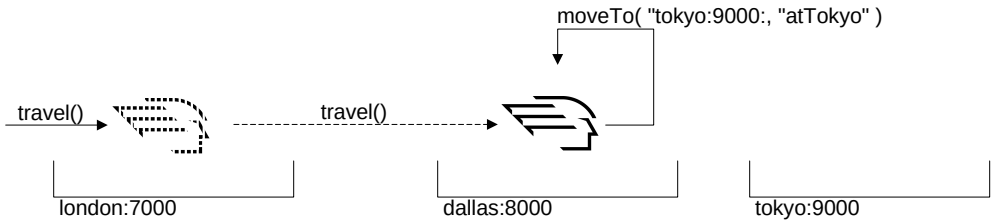
To create an agent, extend the base class `COM.objectspace.voyager.Agent`, and then use Voyager's `vcc` utility to remote-enable the agent's class. Use the resulting virtual class to instantiate an agent object and use virtual references to communicate with this object even if it moves.

Like all objects, an agent can be moved from one program to another. However, unlike simple objects, an agent can move itself autonomously. An agent can move to other programs, allowing the execution of distributed itineraries, or an agent can move to other objects, allowing communication using high-speed, local messaging.

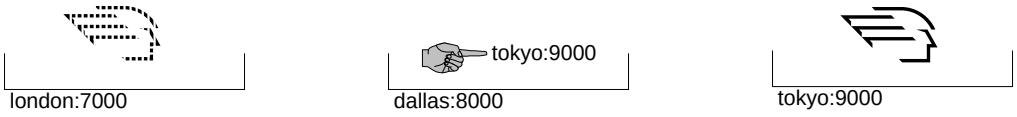
An agent can move to another program and continue to execute when it arrives by sending itself `moveTo()` with the address of the destination program and the name of the member function that should be executed on arrival.

For example, an agent in `dallas:8000` is told to travel. The agent sends itself a `moveTo()` message with two parameters: `dallas:9000`, the destination address, and `atTokyo`, the name of the callback function.

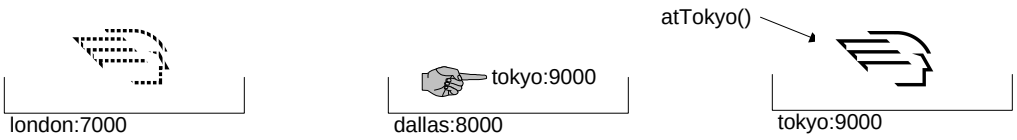
```
public void travel() // defined in Shopper
  {
  moveTo( "tokyo:9000", "atTokyo" );
  }
```



The agent then moves to `tokyo:9000`, leaving behind a forwarder to forward messages.



After arriving at its new location, the agent automatically receives the `atTokyo()` message.



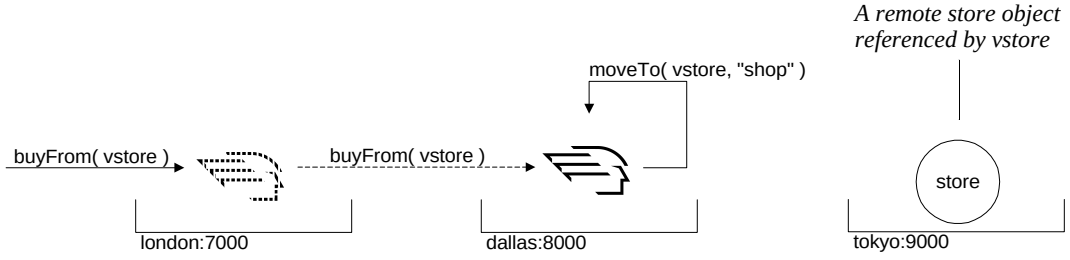
The following code in the agent is then executed.

```
public void atTokyo() // defined in Shopper
  {
  // this code is executed when I move successfully to tokyo:9000.
  }
```

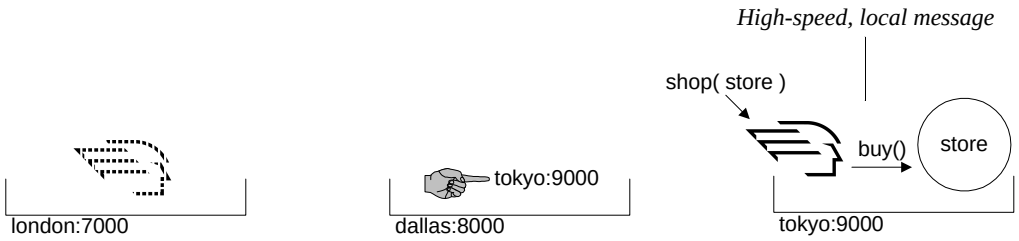
If an agent wants to have a high-speed conversation with a remote object, the agent can move to the object and then send it local Java messages. The easiest way for an agent to move to an object is by sending itself a variation of `moveTo()` that specifies both a virtual reference to the destination object and a callback parameter.

For example, an agent in `dallas:8000` is told to buy from a store object. The agent sends itself a `moveTo()` message with two parameters: `vstore`, a virtual reference to the remote store object, and `shop`, the name of a callback function.

```
public void buyFrom( VStore vstore ) // defined in Shopper
  {
  moveTo( vstore, "shop" );
  }
```



After leaving behind a forwarder and moving to `tokyo:9000`, the agent receives the callback message `shop()` with a local native Java reference to the object `store`.



The following code in the agent is then executed.

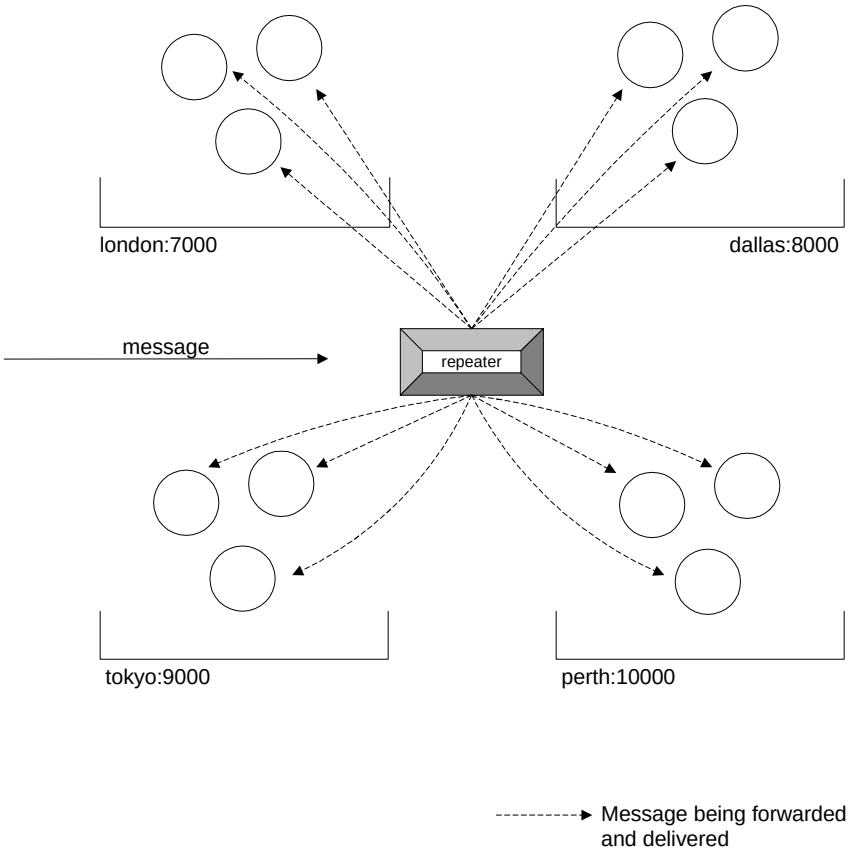
```
public void shop( Store store ) // defined in Shopper
  {
  // this code is executed when I successfully move to the store
  // note that store is a regular Java reference to the store
  int price = store.buy( "widget" );
  }
```

Space

Many distributed systems require features for communicating with groups of objects. For example:

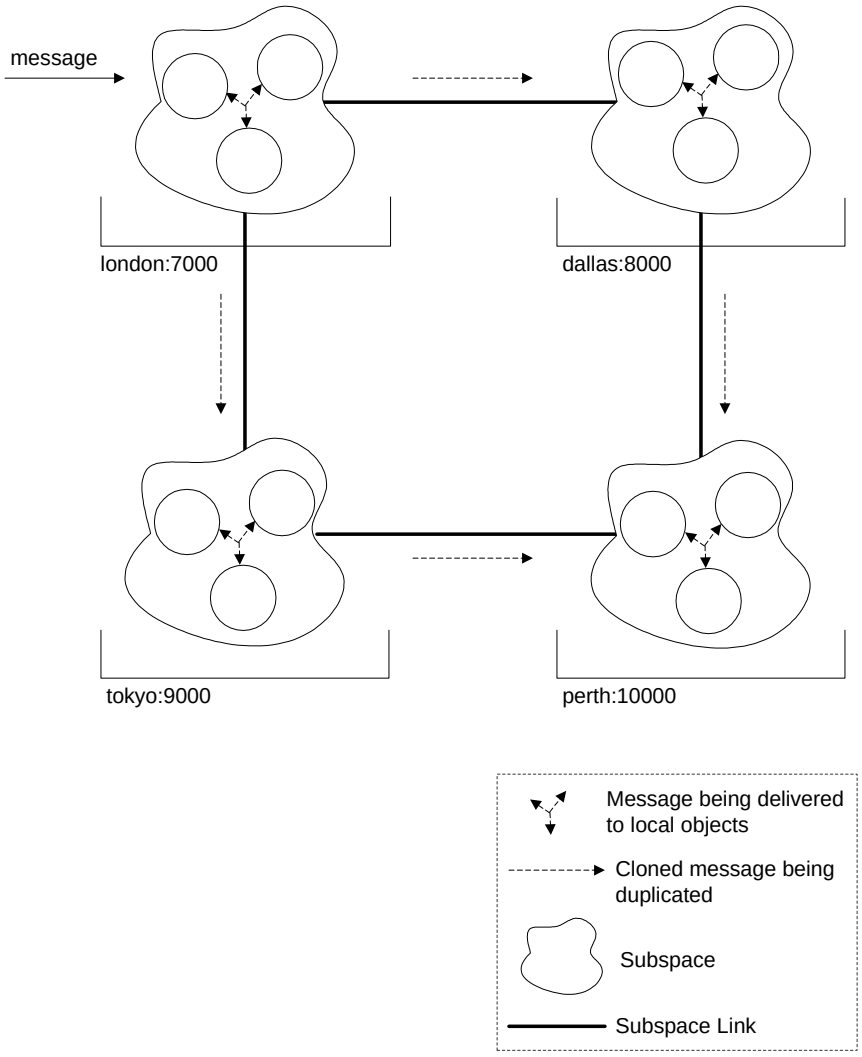
- Stock quote systems use a distributed event feature to send stock price events to customers around the world.
- A voting system uses a distributed messaging feature (multicast) to send messages around the world to voters, asking their views on a particular matter.
- News services use a distributed publish/subscribe feature so that broadcasts are received only by readers interested the broadcast topic.

Most traditional systems use a single *repeater* object to replicate the message or event to each object in the target group.



This traditional approach works well if the number of objects in the target group is small, but does not scale well when large numbers of objects are involved.

Voyager uses a different and innovative architecture for message/event replication called Space™ that can scale to global proportions. Clusters of objects in the target group are stored in local groups called *subspaces*. Subspaces are linked to form a larger logical group called a *Space*. When a message or event is sent into one of the subspaces, the message or event is cloned to each of the neighboring subspaces before being delivered to every object in the local subspace. This process results in a 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.



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

Message Types

Unlike traditional object request brokers, which use a simple, on-the-wire message protocol, Voyager messages are delivered by lightweight agents called *messengers*. Voyager has four predefined message types.

Synchronous Messages

By default, Voyager messages are synchronous. When a caller sends a synchronous message, the caller blocks until the message completes and the return value is received. You can use regular Java syntax to send a synchronous message to an object. Arguments are automatically encoded on the sender side and decoded on the receiver side.

```
int price = vstore.buy( "Killer Rabbits" );
```

One-Way Messages

Although messages are synchronous by default, Voyager supports one-way messages as well. One-way messages do not return a value. When a caller sends a one-way message, the caller does not block while the message completes.

```
vstore.buy( "Killer Rabbits", new OneWay() ); // no return
  ...
```

Future Messages

Voyager also supports future messages. When a caller sends a future message, the caller does not block while the message completes. The caller receives a placeholder that can be used to retrieve the return value later by polling, blocking, or waiting for a callback.

```
Result result = vstore.buy( "Killer Rabbits", new Future() );
  ...
int price = result.readInt(); // Block for price.
```

One-Way Multicast Messages

One-way multicast messages can be used to send one-way messages to all objects in a Space using a single operation.

```
VStore stores = new VStore( space ); // gateway into space
stores.stock( "video", 25 ); // send stock() to all stores in space
```

To send a one-way message to only certain objects in a Space, use a one-way multicast message with a selector.

Dynamic Messaging

Voyager supports dynamic message construction at run time. The following code creates a synchronous message at run time using the Java virtual machine syntax for signature definition.

```
// Dynamically create and execute a synchronous message.
Sync sync = new Sync();
sync.setSignature( "buy(Ljava.lang.String;)I" );
sync.writeObject( "Killer Rabbits" );
Result result = vstore.send( sync );
int price = result.readInt(); // price
```

Life Spans and Garbage Collection

Each instance of a remote-enabled class has a life span. When an object reaches the end of its life span, the object dies and is garbage-collected. Garbage collection destroys an object, freeing the object's memory for reclamation by the Java virtual machine.

Voyager includes a distributed garbage collector that supports a variety of life spans.

- An object can live forever.
- An object can live until there are no more local or virtual references to it. By default, an instance of any class that does not extend `Agent` has this kind of reference-based life span.
- An object can live for a specified amount of time. By default, an instance of any class that extends `Agent` lives for one day.
- An object can live until a particular point in time.

You can change an object's life span at any time.

3

Guided Tour

This chapter guides you through an example project to demonstrate the power and simplicity of the ObjectSpace Voyager™ Core Technology (Voyager). All steps necessary to build an agent-enhanced, persistent electronic shopping system are presented, complete with full, annotated source code from the directory `\voyager1.0.0\examples\shopper`.

This chapter is not a technical reference. For information about a particular aspect of Voyager, consult Part 2, “ObjectSpace Voyager ORB.”

Introduction

One of the hot areas of computer technology is electronic commerce. As companies begin to allow customers to purchase goods and services electronically, an interesting opportunity arises for the consumer. Rather than scan the yellow pages for stores that sell the product you want, why not use a personal shopping agent that can do this for you automatically? Such an agent could learn your tastes and requirements over time and tirelessly scour the network to find you the best possible deal. Another advantage of such agents is their proactive abilities—they could be smart enough to locate items similar to those you purchased in the past and suggest them to you, rather than relying on you to continually prompt them into action.

This chapter demonstrates how to build a simple version of an agent-based shopping system in the following phases.

Phase 1: Building Stores. In the first phase, a Java™ class is defined to represent a store. Two persistent stores are constructed in different Voyager servers, and each store is added to a well-known registry. The Voyager servers are then shut down.

Phase 2: Launching a Shopping Agent. In the second phase, a Java class is defined to represent a shopping agent. The Voyager servers are restarted, and a persistent shopping agent named `Alfred` is constructed in one of the servers and told the location of the store registry and the name of the desired product. `Alfred` sets his itinerary to the contents of the registry, visits each store in turn, and then moves to the store with the best price to await further instructions.

Phase 3: Buying an Item. In the third and final phase, a program is written that contacts `Alfred` to request the store location offering the best price. The program then tells `Alfred` to die and contacts the recommended store to purchase the item directly.

The remainder of this chapter describes how to complete each phase of the shopping system using Voyager. The program in this section is text based. For an applet-based version of the same program, consult Chapter 10, “Applets.”

Phase 1: Building Stores

The first phase of the shopping system project constructs two persistent stores and adds virtual references to each store into a registry by performing the following steps:

1. Define a Java class named `Store` that represents a store and generate a virtual version of `Store` so it can be constructed remotely.
2. Choose a class for the registry and generate a virtual version of this class.
3. Write a program named `Build.java` that creates two persistent stores and populates the remote registry.
4. Compile the Phase 1 programs.
5. Start Voyager servers to hold the persistent remote stores.
6. Run `Build.class` to create the stores and registry.
7. Shut down the Voyager servers.

The rest of this section discusses these steps in detail.

Step 1. Define a Java class named Store that represents a store and generate a virtual version of Store so it can be constructed remotely.

For the purposes of this example, a store has limited behavior. The `Store.java` program below gives the `Store` class a name and a hash table that maps the name of a product to its price. `Store` defines functions for adding, pricing, and purchasing a product. Several functions contain print statements used to track transactions as they occur. `Store` is defined to be serializable so it can be stored persistently in the default Voyager database.

Class voyager1.0.0\examples\shopper\Store.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

import java.util.Hashtable; // utilize a JDK Hashtable

public class Store implements java.io.Serializable
  {
  String name;
  Hashtable products = new Hashtable(); // contains product->price pairs

  public Store( String name )
    {
    this.name = name;
    System.out.println( "Build " + this );
    }

  public String toString()
    {
    return "Store( " + name + " )";
    }

  public void stock( String product, int price )
    {
    System.out.println( "stock " + product + " @ $" + price );
    products.put( product, new Integer( price ) ); // add product to stock
    }

  public int getPrice( String product )
    {
    Integer integer = (Integer) products.get( product ); // get price
    return integer == null ? 0 : integer.intValue(); // zero if not in stock
    }

  public int buy( String product ) throws IllegalArgumentException
    {
    int price = getPrice( product );

    if( price == 0 )
      throw new IllegalArgumentException( "no " + product + " found" );

    System.out.println( "purchase " + product + " @ $" + price );
    return price;
    }
  }
```

Run the Voyager `vcc` utility on `Store`. The `vcc` utility uses the most recently modified version of the `Store.java` or `Store.class` file to create a virtual class named `VStore`.

For the rest of this guided tour, assume commands are typed from a command line in the directory `voyager1.0.0\examples\shopper`.

```
>vcc Store
vcc 1.0.0, copyright objectspace 1997
note: VoyagerException not thrown by java.lang.Object:java.lang.String
toString()
>dir Vstore.*
VSTORE~1 JAV         4,658  08-22-97 10:17a VStore.java
>
```

Like all virtual classes, `VStore` directly or indirectly extends `VObject`, which contains the functionality common to all virtual objects.

Step 2. Choose a class for the registry and generate a virtual version of this class.

This guided tour uses the JDK class `java.util.Vector` as the registry class. Run `vcc` to create a virtual version of `Vector`. The following code creates the virtual class `VVector` and places it in the current directory.

```
>vcc java.util.Vector
vcc 1.0.0, copyright objectspace 1997
note: java.* virtual classes are not placed in a package
note: VoyagerException not thrown by java.lang.Object:java.lang.Object
clone()
note: VoyagerException not thrown by java.lang.Object:java.lang.String
toString()
>dir VVector.*
VVECTO~1 JAV        21,434  08-22-97 11:04a VVector.java
>
```

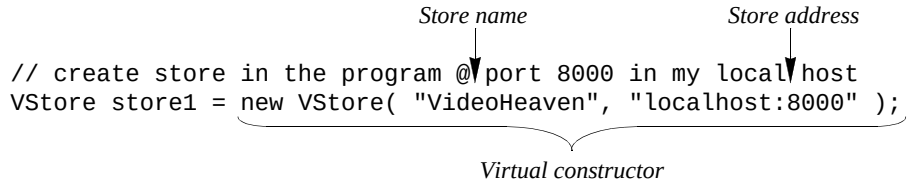
Step 3. Write a program named Build.java that creates two persistent stores and populates the remote registry.

The `Build.java` program below constructs two persistent instances of `Store` in local Voyager servers and populates the persistent registry.

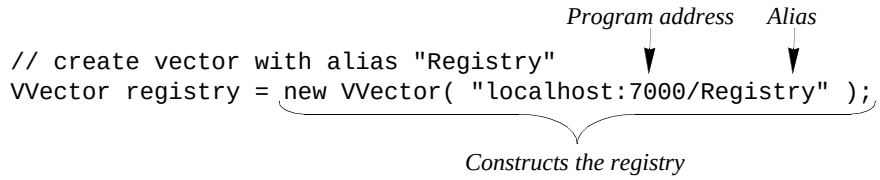
Application voyager1.0.0\examples\shopper\Build.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

import COM.objectspace.voyager.*;
import VVector;

public class Build
  {
  public static void main( String args[] )
    {
    try
      {
      // create store in local server @ port 8000
      VStore store1 = new VStore( "VideoHeaven", "localhost:8000" );
      store1.liveForever(); // prevent garbage collection
      store1.stock( "Killer Rabbits", 25 ); // stock item
      store1.stock( "Jaws XXIII", 29 ); // stock item
      store1.saveNow(); // become persistent, save copy to database

      // create store in local server @ port 7000
      VStore store2 = new VStore( "MegaHits", "localhost:7000" );
      store2.liveForever(); // prevent garbage collection
      store2.stock( "Killer Rabbits", 35 ); // stock item
      store2.stock( "Jaws XXIII", 30 ); // stock item
      store2.saveNow(); // become persistent, save copy to database

      // create vector with alias "Registry" in local server @ port 8000
      VVector registry = new VVector( "localhost:8000/Registry" );
      registry.liveForever();
      registry.addElement( store1 );
      registry.addElement( store2 );
      registry.saveNow(); // store in database

      System.out.println( "Registry is " + registry );
      Voyager.shutdown(); // shutdown program
      }
    catch( VoyagerException exception )
      {
      System.err.println( exception );
      }
    }
  }
```

The `Build.java` program uses the `VStore` constructor to instantiate a `Store`. Virtual class constructors have the same arguments as their original classes plus an additional string that specifies the address of the destination program. In other words, the virtual counterpart of the constructor `Store( String` *name* `)` is `VStore( String` *name*, `String` *address* `)`. The format of an object's address resembles a URL and usually includes the host name and port number of the program in which the object is to be created. You can supply a specific host name or use the built-in host name `localhost` to denote your current local host.



The `Build.java` program also creates a remote `Vector` and assigns it the alias `Registry` in a single step. Voyager allows you to assign an alias to a new object using standard URL syntax. A separate name-binding step is not required.



The following excerpt from `Build.java` demonstrates how to prevent an object from being garbage-collected. By default, a simple (non-agent) remote object is garbage-collected when there are no more local or virtual references to it. Sending the `liveForever()` message to an object prevents its garbage collection; that is, the object lives forever unless you explicitly send it the `dieNow()` message. Because the stores and the registry must survive beyond the lifetime of the program, they are made immortal.

```
store1.liveForever(); // prevent garbage collection
```

The `saveNow()` method instructs an object to become persistent and save itself into its program's database.

```
store1.saveNow(); // become persistent, save copy to database
```

Step 4. Compile the Phase 1 programs.

Use the `javac` command to compile the Phase 1 source code.

```
javac Store.java Build.java VStore.java VVector.java
```

Step 5. Start Voyager servers to hold the persistent remote stores.

Start a Voyager server on each of ports 7000 and 8000 by running the `voyager` command in two separate windows. As shown below, this command accepts the required port number as an argument. The `-d` option instructs Voyager to use the named database file for its persistent storage. The `-c` option clears the database file if one already exists. Note that a Voyager server runs until it is explicitly terminated, and two Voyager programs cannot share the same port.

Window 1

```
>voyager 7000 -cd 7000.db
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:7000
database = 0 objects, 0 classes
```

Window 2

```
>voyager 8000 -cd 8000.db
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:8000
database = 0 objects, 0 classes
```

Step 6. Run Build.class to create the stores and registry.

Run `Build.class` in a third window. The following output is initially displayed.

Window 3

```
>java Build
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1187
```

Window 3 remains inactive while `Build.java` constructs the persistent stores and populates the registry. As `Build.java` executes, additional output displays in the first two windows:

Window 1

```
>voyager 7000 -cd 7000.db
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:7000
database = 0 objects, 0 classes
Build Store( MegaHits )
stock Killer Rabbits @ $35
stock Jaws XXIII @ $30
```

Window 2

```
>voyager 8000 -cd 8000.db
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:8000
database = 0 objects, 0 classes
Build Store( VideoHeaven )
stock Killer Rabbits @ $25
stock Jaws XXIII @ $29
```

After the stores are constructed and the registry is populated, the `Build.java` program displays an additional line of output in Window 3, then terminates.

Window 3

```
>java Build
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1187
Registry is [Store( VideoHeaven ), Store( MegaHits )]
>
```

Step 7. Shut down the Voyager servers.

Phase 1 is now complete. Shut down the two Voyager servers by pressing Ctrl+C in Windows 1 and 2.

Phase 2: Launching a Shopping Agent

In this phase, a persistent shopping agent named `Alfred` is used to find the best price for a product. When launched, `Alfred` sets his itinerary to the contents of the registry, visits each store in the itinerary to find the best price, and then parks at the server that contains the best store to await further instructions. Phase 2 is comprised of the following steps:

1. Define a class named `Shopper` that represents a shopping agent.
2. Write a program named `Shop.java` to instantiate and launch an instance of `Shopper`.
3. Create a virtual version of `Shopper`.
4. Compile the Phase 2 programs.
5. Restart the Voyager servers.
6. Run `Shop.class` to launch the shopper.

The rest of this section discusses these steps in detail.

Step 1. Define a class named Shopper that represents a shopping agent.

The `Shopper` class defined below extends the `Agent` class and adds behavior specific for a shopping agent. The program does not need to override any special functions for the agent to operate correctly. Because `Agent` implements `Serializable`, all the nontransient, nonstatic fields in the agent are automatically maintained as the agent moves.

Class voyager1.0.0\examples\shopper\Shopper.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

import COM.objectspace.voyager.*;
import java.util.Vector;
import VVector;

public class Shopper extends Agent
  {
  String product; // the product to locate
  Vector itinerary; // list of stores to visit
  int index; // index into itinerary
  VStore bestStore = null; // store with cheapest price
  int bestPrice = Integer.MAX_VALUE; // current best price
  boolean parked = false; // have I finished?

  public void findBestPriceFor( String product, VVector registry )
    {
    this.product = product;

    try
      {
      moveTo( registry, "atRegistry" );
      }
    catch( VoyagerException exception )
      {
      System.err.println( exception );
```

```
    }
  }

public void atRegistry( Vector registry )
  {
  itinerary = (Vector) registry.clone(); // get local copy of registry
  System.out.println( "shopping using itinerary: " + itinerary );

  try
    {
    moveTo( (VStore) itinerary.elementAt( index ), "shop" );
    }
  catch( VoyagerException exception )
    {
    System.err.println( exception );
    }
  }

public void shop( Store store )
  {
  int price = store.getPrice( product );

  if( price == 0 )
    {
    System.out.println( "at " + store + ", " + product + " not sold" );
    }
  else
    {
    System.out.println( "at " + store + ", " + product + " is $" + price );

    if( price < bestPrice ) // best store so far
      {
      // obtain virtual reference to store
      try
        {
        bestStore = (VStore) VObject.forObject( store );
        }
      catch( VoyagerException exception )
        {
        System.err.println( exception );
        }

      bestPrice = price;
      }
    }

  // delay to make execution easier to follow
  try{ Thread.sleep( 5000 ); } catch( InterruptedException exception ) {}

  try
    {
    if( ++index < itinerary.size() )
     moveTo( (VStore) itinerary.elementAt( index ), "shop" ); // next store
    else
      moveTo( bestStore.getProgramAddress(), "park" ); // best store
```

```
    }
  catch( VoyagerException exception )
    {
    System.err.println( exception );
    }
  }

public void park()
  {
  parked = true;
  System.out.println( "at " + bestStore + ",  best price $" + bestPrice );
  System.out.println( "shopper parks at " + Voyager.getAddress() );

  if( getPersistent() ) // if i'm persistent save my final state
    {
    try
      {
      flushNow(); // save copy to database and flush from memory
      }
    catch( VoyagerException exception )
      {
      System.err.println( exception );
      }
    }
  }

public VStore getBestStore()
  {
  if( !parked )
    throw new IllegalStateException( "not parked yet" );

  return bestStore;
  }
}
```

A shopper uses two variations of `moveTo()` to move and continue execution on arrival:

- When `moveTo()` is passed a program address and a function name, the agent is moved to the specified program and is then resumed by executing the callback function with no arguments.
- When `moveTo()` is passed a virtual reference and a function name, the agent is moved to the referenced object and is then resumed by executing the callback function with a local reference to the target object as the single argument. The agent can then communicate with the target object using high-speed, local Java method calls.

This callback style of programming, familiar to any programmer who has created a graphical user interface, neatly avoids Java's inability to maintain an execution stack across virtual machine boundaries.

If an error occurs at any time during a move, an exception is thrown when the agent calls `moveTo()`. Because an agent is deactivated conceptually when it executes `moveTo()`, a programming error occurs if any code other than exception-handling code follows these methods.

Step 2. Write a program named Shop.java to instantiate and launch an instance of Shopper.

The `Shop.java` program below creates a persistent instance of `Shopper` with alias `Alfred` and tells him to find the best price for a video named *Killer Rabbits*.

Application voyager1.0.0\examples\shopper\Shop.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

import COM.objectspace.voyager.*;
import VVector;

public class Shop
  {
  public static void main( String[] args )
    {
    try
      {
      // connect to vector with alias "Registry" in local server @ port 8000
      VVector registry =
        (VVector) VObject.forObjectAt( "localhost:8000/Registry" );

      // create a shopper with alias "Alfred" in local server @ port 7000
      VShopper shopper = new VShopper( "localhost:7000/Alfred" );
      shopper.saveNow(); // become persistent, save copy to database

     // ask the shopper to use the registry to find the best price of product
      shopper.findBestPriceFor( "Killer Rabbits", registry );

      // shutdown program
      Voyager.shutdown();
      }
    catch( VoyagerException exception )
      {
      System.err.println( exception );
      }
    }
  }
```

The first line of `Shop.java` uses the static method `VObject.forObjectAt()` to obtain a virtual reference to the existing remote `Vector` named `Registry`.

The second and third lines of `Shop.java` create a persistent `Shopper` with alias `Alfred`. Unlike a simple (non-agent) object, an agent lives for one day by default. This allows an agent to roam a network and perform duties without requiring any local or virtual references. Chapter 4, “Introduction,” explains how you can specify an agent’s life span so that the agent can either live forever or die when it has no local or virtual references.

The fourth line of `Shop.java` instructs `Alfred` to find the best price for the *Killer Rabbits* video. When `Alfred` receives the message `findBestPriceFor()`, he saves the name of the product for future use and then executes the following code:

```
    moveTo( registry, "atRegistry" );
```

This function deactivates `Alfred`, moves him from server 8000 to the registry in server 7000, and then reactivates him by sending him the message `atRegistry()` with the registry as its single argument. Because `Alfred` is persistent, his database backup copy is automatically moved between servers when `Alfred` moves. When `Alfred` arrives at server 7000, the original call to `findBestPriceFor()` returns, and the `Shop.java` program terminates. `Alfred`, however, continues to execute in the Voyager server. He stores a clone of the registry and then executes the following code:

```
moveTo( (VStore) registry.elementAt( index ), "shop" );
```

This causes `Alfred` to move to the first store (located in server 8000) and execute `shop()` with the store as its single argument. This function gets the price of the product from the store, updates the variable `bestStore` if appropriate, and then moves `Alfred` to the next store. This sequence continues until the itinerary is exhausted, at which point `Alfred` executes the following code:

```
moveTo( bestStore.getProgramAddress(), "park" );
```

This variation of `moveTo()` causes `Alfred` to move into the program that holds the best store and then execute `park()` with no arguments. The `park()` method displays a status message and flushes the final state of `Alfred` to the local database. When `park()` completes, `Alfred`'s thread of execution finishes, but `Alfred` does not die.

Step 3. Create a virtual version of Shopper.

Use `vcc` to create a virtual version of `Shopper`.

```
>vcc Shopper
vcc 1.0.0, copyright objectspace 1997
>dir VShopper.*
VSHOPP~1 JAV         5,926  08-25-97  9:37a VShopper.java
>
```

Step 4. Compile the Phase 2 programs.

Use the `javac` command to compile the Phase 2 source files.

```
javac Shopper.java VShopper.java Shop.java
```

Step 5. Restart the Voyager servers.

Restart a Voyager server on each of ports 7000 and 8000 in two different windows. As in Phase 1, use the `-d` option to load the handles of all persistent objects in a database.

Window 1

```
>voyager 7000 -d 7000.db
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:7000
database = 1 object, 0 classes
```

Window 2

```
>voyager 8000 -d 8000.db
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:8000
database = 2 objects, 0 classes
```

Step 6. Run Shop.class to launch the shopper.

Run `Shop.class` in a third window. This program launches `Alfred` and then immediately terminates. The following output is displayed.

Window 3

```
>java Shop
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1191
>
```

As `Alfred` moves from server to server to find the best price, additional output displays in the first two windows.

Window 1

```
>voyager 7000 -d 7000.db
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:7000
database = 1 object, 0 classes
at Store( MegaHits ), Killer Rabbits is $35
```

Window 2

```
>voyager 8000 -d 8000.db
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:8000
database = 2 objects, 0 classes
shopping using itinerary: [Store( VideoHeaven ), Store( MegaHits )]
at Store( VideoHeaven ), Killer Rabbits is $25
at Store( VideoHeaven ),  best price $25
shopper parks at 208.6.239.200:8000
```

After `Alfred` finds the best price for *Killer Rabbits*, he parks in server 8000 to await further instructions. Phase 2 is now complete. Because the stores, registry, and shopper are all persistent, the Voyager servers could be shut down and restarted at this point without causing problems.

Phase 3: Buying an Item

In the final phase of the shopping system project, a program is created that contacts `Alfred`, asks for the best store, tells `Alfred` to die, and then makes a remote purchase from the store. When `Alfred` dies, his resources are reclaimed by the local Java virtual machine. Phase 3 is comprised of the following steps:

1. Write a program called `Buy.java` that uses `Alfred`'s recommendation to purchase a product.
2. Compile `Buy.java`.
3. Run `Buy.class`.

Step 1. Write a program called Buy.java that uses Alfred's recommendation to purchase a product.

The following is the `Buy.java` source code:

Application voyager1.0.0\examples\shopper\Buy.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

import COM.objectspace.voyager.*;

public class Buy
  {
  public static void main( String[] args )
    {
    try
      {
      // connect to Alfred, whose last known location was server @ port 7000
      VShopper shopper =
        (VShopper) VObject.forObjectAt( "localhost:7000/Alfred" );

      // ask the shopper for the best store, waiting if not ready yet
      VStore bestStore = getBestStore( shopper );

      // tell the shopper to die
      System.out.println( "sorry Alfred, but i have to kill you now" );
      shopper.dieNow(); // kill and remove from database

      // buy the product
      int price = bestStore.buy( "Killer Rabbits" );
      System.out.println( "bought video for $" + price + " @ " + bestStore );

      // shutdown program
      Voyager.shutdown();
      }
    catch( VoyagerException exception )
      {
      System.err.println( exception );
      }
    }

  /**
   * Get the best store from the agent, waiting until the store is available
   */
  static VStore getBestStore( VShopper shopper ) throws VoyagerException
    {
    while( true )
      {
      try
        {
        return shopper.getBestStore();
        }
      catch( IllegalStateException exception )
        {
        System.out.println( "Shopper not parked yet" );
        try { Thread.sleep( 2000 ); } catch( InterruptedException ie ) {}
        }
      }
    }
  }
```

Step 2. Compile Buy.java.

Use the `javac` command to compile the Phase 3 program.

```
javac Buy.java
```

Step 3. Run Buy.class.

The `Buy.java` program connects to `Alfred`, using his last known location at server 7000, even though by this time he has moved to the best store in server 8000. This is possible due to the trail of forwarders `Alfred` leaves behind as he moves. `Buy.java` then attempts to obtain a virtual reference to the best store. If `Alfred` has not yet parked, an exception is thrown, which the `Buy.java` program catches. `Buy.java` continues to attempt to get a virtual reference to the best store. When successful, the program kills `Alfred` (removing him from the local database) and makes the purchase.

Running `Buy.java` generates the output below.

```
>java Buy
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1195
sorry Alfred, but i have to kill you now
bought video for $25 @ Store( VideoHeaven )
>
```

This concludes the guided tour.

Part 2

ObjectSpace Voyager ORB

4

Introduction

Part 2 is comprised of nine chapters that describe the various object request broker (ORB) features of the ObjectSpace Voyager™ Core Technology (Voyager).

- Read this chapter for a summary of the Part 2 chapters.
- Read Chapter 5, “Fundamental ORB Features,” for a description of the features commonly associated with ORBs, such as the creation of virtual classes, remote instantiation, remote messaging, connecting to existing objects, and object life spans. Two of the main Voyager classes introduced are `Voyager` and `VObject`.
- Read Chapter 6, “Advanced Messaging,” for an explanation of Voyager’s messaging capabilities, including lightweight messenger agents. Some of the main Voyager classes introduced are `Messenger`, `SmartMessenger`, `Future`, `OneWay`, `Sync`, `Result`, `OneWayResult`, `UnicastResult`, `ResultEvent`, and `ResultListener`.
- Read Chapter 7, “Events, Listeners, and Assistants,” for an explanation of how to monitor programs and mobile objects using Voyager’s events, listeners, and assistants. Some of the main Voyager classes introduced are `ObjectEvent`, `ObjectListener`, `SystemEvent`, `SystemListener`, and `Monitor`.
- Read Chapter 8, “Mobility,” for an explanation of how to move objects between programs at run time, even as the objects continue to receive remote messages. Also included are figures that illustrate the performance benefits of mobility.
- Read Chapter 9, “Agents,” for details about the creation and deployment of mobile, autonomous agents. Some of the main Voyager classes introduced are `Agent` and `VAgent`.
- Read Chapter 10, “Applets,” for step-by-step instructions on how to use Voyager to create network-enabled applets that take full advantage of ORB and agent technology.
- Read Chapter 11, “Security,” for information about restricting the operations that can be performed by foreign objects. Voyager’s security manager allows you to control the features available to roaming agents. One of the main Voyager classes introduced is `VoyagerSecurityManager`.
- Read Chapter 12, “Customizing Voyager Applications,” to learn how to customize your Voyager applications to your specific needs. You can use custom sockets, extend class loading behavior, and configure your computer with more than one domain name.

5

Fundamental ORB Features

This chapter describes fundamental distributed computing using the ObjectSpace Voyager™ Core Technology (Voyager) features, including remote-enabling classes, exception handling, virtual references, garbage collection, and life spans. The source code for the examples in this chapter is located in `\voyager1.0.0\examples\orb`. Execute all commands for compiling and running the examples from within this directory. Adding `\voyager1.0.0` to your `CLASSPATH` is required to run the Voyager example programs. Usually, this path is set automatically at startup.

Starting Voyager Within a Program

Each Voyager program has a single communications port with a number that is unique to the program's local host. Additionally, each Voyager program has a Web root used for remote class loading, described in detail in Chapter 10, "Applets." These features enable you to create objects in remote programs and move the objects between programs.

By default, when you call a Voyager method, Voyager starts on a random, unique port number and the Web root is set to a backslash (\). To start Voyager on a specific port number, call `Voyager.startup()` with the desired port number before calling any other Voyager methods.

A Voyager program displays a copyright notice, the program address, and the program root during startup. The address of a program consists of the name of the host and its port number, separated by a colon. A host name can be either a numeric IP address, like `208.6.239.200`, or a symbolic host name, like `homer`.

Use the `javac` command from the `\voyager1.0.0\examples\orb` directory to compile the `Program1.java` example program:

```
javac Program1.java
```

Now run `Program1.class`.

```
>java examples.orb.Program1
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1072
program address is 208.6.239.200:1072
program running...
```

Application voyager1.0.0\examples\orb\Program1.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.orb;

import COM.objectspace.voyager.*;

public class Program1
  {
  public static void main( String args[] )
    {
    // display address of program
    System.out.println( "program address is " + Voyager.getAddress() );

    // program code goes here...
    System.out.println( "program running..." );
    }
  }
```

Note: When executed, this example program does not automatically terminate when the last line of the program is reached. This default behavior allows objects to move in and out of a program even after the main method of the program has completed.

To terminate a Voyager program, execute `Voyager.shutdown()`, which shuts down the program gracefully by allowing sockets to flush pending data to the network and by cleaning up all threads. To manually set a program's port number, execute `Voyager.startup()` in the first line of the main program.

Use the `javac` command from the `\voyager1.0.0\examples\orb` directory to compile the `Program2.java` example program:

```
javac Program2.java
```

Now run `Program2.class`.

```
>java examples.orb.Program2
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:8000
program address is 208.6.239.200:8000
program running...
>
```

Application voyager1.0.0\examples\orb\Program2.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.orb;

import COM.objectspace.voyager.*;

public class Program2
  {
  public static void main( String args[] )
    {
    try
      {
      // initialize program to use port 8000
      Voyager.startup( 8000 );

      // display address of program
      System.out.println( "program address is " + Voyager.getAddress() );

      // program code goes here...
      System.out.println( "program running..." );

      // shutdown program gracefully
      Voyager.shutdown();
      }
    catch( VoyagerException exception )
      {
      System.err.println( exception );
      }
    }
  }
```

Starting a Voyager Server from a Command Line

A Voyager server is a Voyager program that initially contains no user objects. When you start a Voyager server, the server continues to execute until explicitly terminated. Use the `voyager` utility to start a Voyager server on a specified port. You can start a Voyager server from any directory, regardless of the location of the program source code or object code.

Executing `voyager` with no arguments displays a list of the command line options:

```
voyager(tm) 1.0.0, copyright objectspace 1997

Usage: voyager <argument list>

This utility starts a voyager server on a specified port.
If the server will serve applets from a web server, you
must also supply the root of pages on the web server.

Valid arguments:
  port              the number of the port to serve
  -c                clear voyager database
  -d <filename>     use voyager database with specified name
  -e                extended stack trace
  -i <interpreter>  use this interpreter instead of java
  -p <int>          the maximum thread pool size
  -q                quiet mode, do not display status on startup
  -r <root>         the root of applet codebases
  -s                install voyager security manager
  -t[ohc]           trace mode
                      o = objects
                      h = housekeeping
                      c = communications
  -x                pass remaining parameters to the java interpreter
```

The `-c` (clear) and `-d` (database) options are related to Voyager's database support and are described in detail in Chapter 14, "Database-Independent Persistence."

The `-e` (extended) option causes remote exceptions to include Voyager source information in annotated exceptions. To get this extended stack trace information from within your Voyager program, use `Voyager.setExtendedStackTrace()`.

The `-i` (interpreter) option allows you to specify a different interpreter than the default, `java`.

The `-p` (pool size) option sets the size of the thread pool used for caching of inactive threads.

The `-q` (quiet) option inhibits the display of startup information. To disable startup information from within your Voyager program, use `Voyager.quiet()`.

The `-r` (root) option allows you to override the default location for performing remote class loading for applets. This option is described in detail in Chapter 10, "Applets."

The `-s` (security) option installs a Voyager security manager. See Chapter 11, "Security," for more information.

The `-t[ohc]` (trace) option causes the server to display information about the events generated in the Voyager program, such as incoming and outgoing messages, class loading, and object life spans. The `o` (object) switch causes object events to be traced,

the `c` (communication) switch causes communication system events to be traced, and the `h` (housekeeping) switch causes housekeeping system events to be traced. See the "System Tracing with the Monitor" section of Chapter 7, "Events, Listeners, and Assistants," for information about how to trace your system with Voyager's `Monitor` class.

The `-x` (extra) option passes all remaining parameters to the Java™ interpreter. Each element following the `-x` option must be in the format specified by the Java interpreter or the interpreter specified earlier in the command line with the `-i` option.

Remote-Enabling a Class

Remote-enabling a class allows you to create remote instances of the class and send messages to the remote instances. In order to remote-enable a class in Voyager, simply create a virtual version of the class using the `vcc` (virtual class creator) utility, as discussed in detail on page 47. You do not need to modify the class in any way, nor do you need access to the class source code.

The following rules describe the default relationship between an original class and its virtual version:

- The virtual class is placed into the same package as the original class, except for `java.*` classes, which are placed in no package.
- The virtual class name is `V` plus the original class name; for example, the virtual version of class `Person` is named `VPerson`.
- The virtual class implements the same interfaces as the original class.
- If the original class extends the class `MySuperClass`, then the virtual class extends the class `VMySuperClass`. If the original class extends `java.lang.Object`, the virtual class extends `COM.objectspace.voyager.VObject`.
- Every public constructor *ClassName*`(` <*args*> `)` `throws` <*exceptions*> generates two public constructors with the signatures:
  - *ClassName*`(` <*args*>`, String` *address* `)` `throws` <*exceptions*>†
  - *ClassName*`(` <*args*>`, String` *address*`, long` *timeout* `)` `throws` <*exceptions*>†

  where *address* specifies the program in which the object should be constructed and *timeout* specifies the construction timeout in milliseconds.
- Every public instance method <*returntype*> `foo(` <*args*> `)` `throws` <*exceptions*> generates two public instance methods with the signatures:
  - <*returntype*> `foo(` <*args*> `)` `throws` <*exceptions*>†
  - *Result* `foo(` <*args*>`, Messenger` *messenger* `)`

  where *Result* is a Voyager interface to a class that acts as a placeholder for a method result and *messenger* is the type of Voyager messenger used to deliver the message.
- Every public static method <*returntype*> `foo(` <*args*> `)` `throws` <*exceptions*> generates four public static methods with the following signatures:
  - <*returntype*> `foo(` <*args*>`, String` *address* `)` `throws` <*exceptions*>†
  - <*returntype*> `foo(` <*args*>`, String` *address*`, long` *timeout* `)` `throws` <*exceptions*>†

† By default, `vcc` adds `VoyagerException` to the list of exceptions that a virtual version of a method can throw. If a class implements an interface and there are methods in that interface that do not declare `VoyagerException`, the virtual class method throws Voyager-related exceptions as run-time exceptions for those methods. Use the `-r` (run-time) option to specify that all exceptions be thrown as run-time exceptions.

- *Result* `foo(` *<args>*`, Messenger` *messenger*`, String` *address* `)`
- *Result* `foo(` *<args>*`, Messenger` *messenger*`, String` *address*`, long` *timeout* `)`

where *address* specifies the program in which the static class resides, *timeout* specifies the message timeout value in milliseconds, *Result* is a Voyager interface to a class that acts as a placeholder for a method result, and *messenger* is the type of Voyager smart messenger used to deliver the message.

- The private methods, protected methods, package methods, and data fields of the original class have no counterparts in the virtual class.

The vcc Utility

To create a virtual version of one or more classes, execute the Voyager `vcc` utility. Specify the names of the classes, not including extensions such as `.class` and `.java`. The `vcc` utility searches the directories, `.zip` files, and `.jar` files in your `CLASSPATH` environment variable to find the first `.class` or `.java` file that corresponds to each specified file. If the `.java` file is the only file found, or if the `.java` file was modified more recently than the `.class` file, `vcc` parses the `.java` file; otherwise, `vcc` parses the `.class` file. The `vcc` utility creates, but does not compile, a virtual class for each class specified. Each virtual class has the same package and name as its original class, except the virtual class name begins with a `V`; for example, the virtual version of class `Person` is `VPerson`.

You can run `vcc` on an interface as well as a class. The virtual version of an interface implements the original interface and extends `COM.objectspace.voyager.VObject`. If a method in the original interface does not throw `VoyagerException`, `vcc` guarantees that the virtual class implements the interface by throwing Voyager-related exceptions as run-time exceptions for that method.

You can also run `vcc` on an abstract or nonpublic class. In each case, however, the resulting virtual class cannot be used to construct instances. A constructor that takes a `VSubspace` is provided so that instances of the virtual class can serve as gateways into a Space. See Chapter 15, “Space: Scalable Group Communication,” for details.

Setting Your CLASSPATH

When executed, `vcc` searches for both the class source code and the class object code via `CLASSPATH`. The search is automatically successful if the class source files and object files reside in the same directory. If the class source files and object files reside in different directories, then, for the search to be successful, the directory structure that leads to the source files must mirror the directory structure that leads to the object files, and the root of each path must be added to `CLASSPATH`.

For example, the ObjectSpace convention is to place source code and object files in different directories. Because ObjectSpace packages begin with the prefix `COM.objectspace`, the following directory structure is used for Voyager:

```
\voyager1.0.0
  \COM
    \objectspace
      \voyager        Voyager .class files (object code)
  \src
    \COM
      \objectspace
         \voyager     Voyager .java files (source code)
```

Note: The object code directory structure mirrors the package structure.

So that `vcc` searches are successful, the `CLASSPATH` for each ObjectSpace developer includes both `\voyager1.0.0` and `\voyager1.0.0\src`.

Command Line Options

To list all `vcc` options from the command line, execute `vcc` with no arguments. The following output displays:

```
vcc 1.0.0, copyright objectspace 1997

usage: vcc <argument list>

vcc creates the source of virtual classes from .class
and .java files. By default, non-java.* virtual classes are
placed into the same package as the original class and
java.* virtual classes are not placed into a package.

valid arguments:
  file                  name of class to be processed, without extension
  -d <path>             store packages relative to <path>
  -e <class>            pretend subsequent classes extend <class>
  -i <interpreter>      use <interpreter> instead of java
  -m <class> <package>  virtual version of <class> was placed into <package>
  -o                    output interface version of classes
  -p <package>          put virtuals of subsequent classes into <package>
  -q                    quiet mode, do not display copyright notice
  -r                    throw voyager-related errors as runtime exceptions
  -v                    verbose mode, display status while processing
```

-d (directory) Option

By default, `vcc` places the `.java` and `.class` files of a virtual class into the current working directory. The `-d` option allows you to specify a different root directory to store the virtual classes in. This option is analogous to the `-d` option of `javac`.

For example, to create a virtual version of `examples.orb.Adder` (assuming Voyager is installed in `\voyager1.0.0`), execute the `vcc` utility as shown:

```
>vcc examples.orb.Adder -d \voyager1.0.0
vcc 1.0.0, copyright objectspace 1997
>dir \voyager1.0.0\examples\orb\VAdder.java
VADDER~1 JAV         6,255  08-25-97 12:09p VAdder.java
>
```

If your current directory is the same as the output directory, you can omit the `-d` argument.

```
>cd \voyager1.0.0\examples\orb
>vcc examples.orb.Adder
vcc 1.0.0, copyright objectspace 1997
>dir VAdder.java
VADDER~1 JAV         6,255  08-25-97 12:09p VAdder.java
>
```

Similarly, if your current directory is the same as the one containing the source code and object code of the original class, you do not have to supply the full class name.

```
>cd \voyager1.0.0\examples\orb
>vcc Adder
vcc 1.0.0, copyright objectspace 1997
>dir VAdder.java
VADDER~1 JAV         6,255  08-25-97 12:09p VAdder.java
>
```

-i (interpreter) Option

By default, `vcc` is executed by `java`. To override this default, use the `-i` option followed by the name of the required interpreter. For example, if you are using the Microsoft development system, you can specify `vcc -i jview`.

-p (package) Option

By default, `vcc` does not place `java.*` virtual classes into any package, but it places all other virtual classes into the same package as the original class. To override this default, use the `-p` option followed by the name of a package. The virtual versions of all subsequent classes on the command line are placed into the specified package.

For example, to create a virtual version of `java.util.Vector` and place it into the package `test` relative to `\voyager1.0.0`, use the following command:

```
vcc -p test java.util.Vector -d \voyager1.0.0
```

-m (map) Option

The `-m` option is used to instruct `vcc` about previous uses of the `-p` option. For example, suppose you create a virtual version of `java.util.Dictionary` as follows:

```
vcc -p test java.util.Dictionary -d \voyager1.0.0
```

Now, suppose you want to create a virtual version of `java.util.Hashtable` that inherits from `java.util.Dictionary`. For `vcc` to know where the virtual version of `java.util.Dictionary` resides, you must use the `-m` option followed by the name of the original class and the original class package.

```
vcc -p test java.util.Hashtable -d \voyager1.0.0 -m java.util.Dictionary test
```

The `-m` option is unnecessary if you can place all of the classes to be repackaged on the same command line.

```
vcc -p test java.util.Dictionary java.util.Hashtable -d \voyager1.0.0
```

-e (extended) Option

By default, if subclass `Alpha` inherits from superclass `Beta`, then the virtual class `VAlpha` inherits from `VBeta`. The `-e` option forces `vcc` to treat all subsequent classes on the command line as if each extends a different specified class. For example, if you create an applet class `MyApplet` that extends `java.applet.Applet`, you can force `vcc` to treat `MyApplet` as if it extended `java.lang.Object` as follows:

```
vcc -e java.lang.Object MyApplet
```

The resultant virtual class `VMyApplet` extends `COM.objectspace.voyager.VObject`. The `-e` option is particularly useful if you want to avoid sending an object messages that are defined in the object's superclasses and if you want to avoid processing every class in the object's superclass chain.

-r (run-time) Option

Voyager supports two exception handling policies: Voyager can throw a Voyager-related exception as a checked exception (`VoyagerException`) or as a run-time exception (`VoyagerRuntimeException`).

By default, `vcc` generates methods that throw the checked exception `VoyagerException`. This exception is thrown if any Voyager-related exception occurs during the method call, including network errors and class-loading problems. Because `VoyagerException` is not a run-time exception, a user of the virtual class must explicitly catch `VoyagerException` when calling a remote method. However, if one of the interfaces implemented by the class declares a method of the same signature, but the interface method does not throw `VoyagerException`, then `vcc` creates the virtual class so that each Voyager-related exception caused by the remote method is thrown as a `VoyagerRuntimeException`. This approach ensures that the generated virtual class implements the interface. In other words, Voyager-related exceptions are thrown as checked exceptions when possible, as run-time exceptions when necessary.

The `-r` option causes `vcc` to take the opposite approach. That is, `vcc -r` generates the virtual class so that Voyager-related exceptions are thrown as run-time exceptions when possible, as

checked exceptions when necessary. If the target class implements an interface that defines a method with the same signature as the remote method, but the interface method declares `VoyagerException` in its throws clause, then `vcc` generates the remote method to throw `VoyagerException`.

To promote good programming practices, most of the examples in this document are processed without the `-r` option.

-q (quiet) Option

By default, `vcc` displays a copyright notice. Use the `-q` option to disable this notice.

-v (verbose) Option

By default, no status is printed as `vcc` operates. Use the `-v` option to view status output.

-o (output interface) Option

The `-o` option instructs `vcc` to generate an interface definition from the original class. For example, to generate an interface for a class `Person`, use the following command.

```
vcc -o Person
```

This generates the interface `IPerson` that includes all public methods defined by `Person`. Each method declares `VoyagerException` in its `throws` clauses. `Person` should then be modified to implement `IPerson` and can be remote-enabled with `vcc` by using the following command.

```
vcc Person
```

Both `Person` and `VPerson` then implement `IPerson`, and users can use interface programming to hide the underlying implementation. See “Interfaces” on page 82 for more information.

vcc Notes

Periodically, `vcc` prints notes to the console, each preceded by the word *note*. If `vcc` is run on a class that implements an interface and some of the interface methods do not declare `VoyagerException` in their `throws` clauses, Voyager prints a note to the console to notify the developer that run-time exceptions will be thrown. To see a `vcc` note, run `vcc` on `examples.orb.Account` (assuming Voyager is installed in `\voyager1.0.0`).

```
>vcc examples.orb.Account -d \voyager1.0.0
vcc 1.0.0, copyright objectspace 1997
note: VoyagerException not thrown by java.lang.Object:java.lang.String
toString()
>
```

By default, `vcc` does not place `java.*` classes into a package. The developer is reminded about this default behavior via a note each time `vcc` is run on a `java.*` class, as shown in the following example.

```
>vcc java.util.Vector
vcc 1.0.0, copyright objectspace 1997
note: java.* virtual classes are not placed in a package
note: VoyagerException not thrown by java.lang.Object:java.lang.Object
clone()
note: VoyagerException not thrown by java.lang.Object:java.lang.String
toString()
>
```

Cyclic References

If two or more classes contain cyclic virtual references to each other, all of the classes involved in the circular reference must be processed by `vcc` before any of their virtual classes are compiled. For example, if a `Man` class contains a `VWoman`, and a `Woman` class contains a `VMan`, use the following commands to create and compile the virtual classes:

```
vcc Man Woman
javac VMan.java VWoman.java
```

For an example that uses these cyclic references, read the “Cyclic References” section on page 74.

All classes containing cyclic references also must be found explicitly in the `CLASSPATH` directories, not only the current directory.

VObject Methods

Once processed with `vcc`, a class can be sent messages and remotely constructed. However, this is just the beginning of what a remote-enabled object can do. Virtual references to objects are instances of `VObject` subclasses and, as such, contain a wide variety of methods that transparently add significant functionality to domain objects. A brief synopsis of a few of the commonly used methods and features are described below. For details on these and many more methods and features, read about the class `COM.objectspace.voyager.VObject` in the API documentation.

Movement

The `moveTo()` family of methods allows a remote-enabled object to be moved around the network or moved to other objects and agents. These methods also allow a moved object to be sent callbacks with optional parameters upon arrival at its destination. When `moveTo()` is sent to an object with a remote object as the destination, the moving object performs the following actions.

1. Locates and moves to the remote object, even if the remote object is moving around the network
2. Locks the remote object
3. Exchanges high-speed, raw Java™ messages with the now-local object

The `moveTo()` methods are often used for load balancing, disconnected operation, and message performance enhancement. See Chapter 8, “Mobility,” for more information.

Persistence

The `save()` and `flush()` methods allow objects to be transparently saved and flushed from memory to persistent storage. These methods can be used with Voyager’s built-in object-storage system or with a variety of third-party databases. See Chapter 14, “Database-Independent Persistence,” for more information.

Life Spans

The `die()` family of methods is used to customize the life span of remote-enabled objects. A comprehensive toolkit of methods is provided for tailoring Voyager’s distributed garbage-collection mechanism on an object-by-object basis. Refer to “Life Spans and Garbage Collection” on page 86 of this chapter for more information.

Properties

The `property()` family of methods provides a way to attach arbitrary key-value properties to a remote-enabled object. This mechanism is leveraged by the publish/subscribe feature, in which one or more subscribe properties are attached to an object. Refer to “Properties” on page 94 of this chapter and to Chapter 15, “Space: Scalable Group Communication,” for more information about properties.

Assistants and Listeners

The `addAssistant()` and `addListener()` family of methods allows you to attach an `ObjectListener` to any remote-enabled object. Voyager automatically generates an `ObjectEvent` if a remote-enabled object has an `ObjectListener` and something of specified interest happens to the object. Therefore, an `ObjectListener` can receive events for almost every object behavior, including movement, dying, saving, and messaging. `ObjectListener` objects can be added as assistants if the listeners need to move with or be persisted with the event source. Consult Chapter 7, "Events, Listeners, and Assistants," for details.

Methods That Override Object Methods

Of particular interest are the methods that override those of the `java.lang.Object` class. The realities of distributed computing sometimes force these methods to be implemented on the virtual reference itself, rather than being delivered to the remote object. The semantics of each of the methods that override the methods on class `Object` are described below.

equals()

When invoked on two virtual references, the `equals()` method returns `true` if both virtual references point to the same remote object, that is, if they have the same GUID. This comparison does not generate extra message traffic because no messages are sent to the remote object.

hashCode()

When invoked on the virtual object, the `hashCode()` method returns a hash code based on the remote object's GUID. Like `equals()`, the `hashCode()` method does not send a message to the remote object, so no extra message traffic occurs.

toString()

The `toString()` method is invoked on the remote object. To maintain the signature of `Object.toString()`, `vcc` cannot add `VoyagerException` to the `throws` clause of `toString()`. As a result, each Voyager-related exception is thrown as a `VoyagerRuntimeException`. Because calls invoked across the network can fail in many ways, every invocation of `toString()` should be accompanied by a try-catch statement to prevent a `VoyagerRuntimeException` from terminating the invoking thread.

clone()

If a class implements `clonable` and exposes `clone()` as a public method, the virtual class `clone()` method is generated. This method is executed on the remote object, and the result of the clone is serialized across the network and returned to the caller. As with `toString()`, callers should accompany an invocation of `clone()` with a try-catch statement to prevent a `VoyagerRuntimeException`. To create a clone of the virtual reference, use `VObject.cloneReference()`. To clone the remote object but return a new virtual reference to the clone, invoke `setVirtual()` before invoking `clone()`.

Other Methods

All other methods of `Object` are implemented on the virtual reference itself.

Remote Construction and Messaging

As discussed on page 46, you can remote-enable a class by creating a virtual version of the class. Remote-enabling a class allows you to construct a remote instance of the original class in any program and send messages to the remote instance.

To construct a remote instance of a class, supply the virtual class constructor the address of the destination program where the remote instance is to reside. If the original class code for the remote instance does not exist in the destination program, the Voyager network class loader automatically loads the original class code into the destination program. If the original class is abstract, the virtual class still can be used to construct instances, but only as gateways into a Space. See Chapter 15, “Space: Scalable Group Communication,” for details.

To send a message to a remote instance, simply send a message to its virtual reference; the message is automatically forwarded to the remote instance. Return values and exceptions are returned to the virtual reference, which passes them to the sender.

By default, a simple (non-agent) remote object is destroyed when it no longer has local or virtual references. If you want a remote object to live whether it has references or not, send `liveForever()` to the object. To destroy an object explicitly, send it the `dieNow()` message. Object life spans are discussed in more detail on page 86.

A virtual class has methods for each public constructor and each public static method in the original class. However, each virtual class method has an additional argument: the destination address, that is, the program where the function should be executed. The built-in address `localhost` refers to your current local host. If you specify `localhost` without a port number, the local program port number is used.

Each remote object is assigned a random, 16-byte globally unique id (GUID). When displayed, a GUID is formatted like a numeric IP address. The address of an object is the object’s program address followed by a slash (/) and its GUID. To get an object’s program address and GUID, use `getObjectAddress()` and `getGUID()`, respectively. To get a remote object’s program address, use `getProgramAddress()`.

The `Account1.java` example program creates an instance of `Account` in the local program, sends the instance messages, and then executes a static method.

Use the following commands from the `\voyager1.0.0\examples\orb` directory to compile `Account1.java`:

```
vcc Account
javac Account.java VAccount.java Account1.java
```

Now run `Account1.class`.

```
>java examples.orb.Account1
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1110
Construct with initial balance 0
program address = 208.6.239.200:1110
Account address =
208.6.239.200:1110/221-49-244-233-168-93-38-63-231-205-72-134-
111-72-132-214
Account id = 221-49-244-233-168-93-38-63-231-205-72-134-111-72-132-214
application = 208.6.239.200:1110
initial balance is 0
deposit 1000
deposit 1000
balance is 1000
accounts opened in localhost = 1
>
```

Class voyager1.0.0\examples\orb\Account.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.orb;

public class Account
  {
  static int opened; // number of opened accounts in program
  int balance; // current balance

  public Account()
    {
    System.out.println( "Construct with initial balance 0" );
    ++opened;
    }

  public Account( int amount )
    {
    System.out.println( "Construct with initial balance " + amount );

    if( amount < 0 )
      throw new IllegalArgumentException( "cannot initialize with "+amount );

    balance = amount;
    ++opened;
    }

  public String toString()
    {
    return "Account( " + balance + " )";
    }

  public synchronized int deposit( int amount )
    {
```

```
  System.out.println( "deposit " + amount );

  if( amount < 0 )
    throw new IllegalArgumentException( "cannot deposit " + amount );

  balance += amount;
  return balance;
  }

public synchronized int withdraw( int amount )
    throws NotEnoughMoneyException
  {
  System.out.println( "withdraw " + amount );

  if( balance < amount )
    throw new NotEnoughMoneyException( amount + " > " + balance );

  balance -= amount;
  return balance;
  }

public synchronized int getBalance()
  {
  return balance;
  }

static public int getOpened()
  {
  return opened;
  }
}
```

Application voyager1.0.0\exampels\orb\Account1.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.orb;

import COM.objectspace.voyager.*;

public class Account1
  {
  public static void main( String args[] )
    {
    try
      {
      // create account in my local program
      VAccount account = new VAccount( "localhost" );

      // display address and id information
      System.out.println( "program address = " + Voyager.getAddress() );
     System.out.println( "Account address = " + account.getObjectAddress() );
      System.out.println( "Account id = " + account.getGUID() );
      System.out.println( "application = " + account.getProgramAddress() );

      // execute some instance methods
      System.out.println( "initial balance is " + account.getBalance() );
      System.out.println( "deposit 1000" );
      account.deposit( 1000 );
      System.out.println( "balance is "+account.getBalance() );

      // execute a static method
      int count = VAccount.getOpened( "localhost" );
      System.out.println( "accounts opened in localhost = " + count );

      Voyager.shutdown();
      }
    catch( VoyagerException exception )
      {
      System.err.println( exception );
      }
    }
  }
```

The `Account2.java` example program is the same as `Account1.java` except `Account2.java` remotely constructs the account in server 8000 using the remote variation of the constructor that allows an initial balance to be specified. Use the following command from the `\voyager1.0.0\examples\orb` directory to compile `Account2.java`:

```
javac Account2.java
```

Start a server on port 8000 in one window, and then run `Account2.class` in a second window.

Window 1

```
>voyager 8000
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:8000
Construct with initial balance 1000
deposit 1000
```

Window 2

```
>java examples.orb.Account2
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1111
application address = 208.6.239.200:1111
Account address =
208.6.239.200:8000/122-197-2-32-202-20-102-84-131-66-72-134-11
1-72-159-192
Account id = 122-197-2-32-202-20-102-84-131-66-72-134-111-72-159-192
application = 208.6.239.200:8000
initial balance is 1000
deposit 1000
balance is 2000
accounts opened in localhost:8000 = 1
>
```

Application voyager1.0.0\examples\orb\Account2.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.orb;

import COM.objectspace.voyager.*;

public class Account2
  {
  public static void main( String args[] )
    {
    try
      {
      // create account with initial balance of 1000 in a remote application
      VAccount account = new VAccount( 1000, "localhost:8000" );

      // display address and id information
      System.out.println( "application address = " + Voyager.getAddress() );
     System.out.println( "Account address = " + account.getObjectAddress() );
      System.out.println( "Account id = " + account.getGUID() );
      System.out.println( "application = " + account.getProgramAddress() );

      // execute some remote instance methods
      System.out.println( "initial balance is " + account.getBalance() );
      System.out.println( "deposit 1000" );
      account.deposit( 1000 );
      System.out.println( "balance is " + account.getBalance() );

      // execute a remote static method
      int count = VAccount.getOpened( "localhost:8000" );
      System.out.println( "accounts opened in localhost:8000 = " + count );

      Voyager.shutdown();
      }
    catch( VoyagerException exception )
      {
      System.err.println( exception );
      }
    }
  }
```

Remote Exceptions

The `Account3.java` example program demonstrates how remote exceptions are handled. When a remote exception occurs, it is shipped to the virtual reference and then rethrown. Use the following command from the `\voyager1.0.0\examples\orb` directory to compile `Account3.java`:

```
javac Account3.java
```

Start a server on port 8000 in one window, and then run `Account3.class` in a second window.

Window 1

```
>voyager 8000
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:8000
Construct with initial balance 500
withdraw 1000
deposit -1
```

Window 2

```
>java examples.orb.Account3
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1113
withdraw 1000
examples.orb.NotEnoughMoneyException: 1000 > 500
        at examples.orb.Account.withdraw(Account.java:48)
        thrown remotely from 208.6.239.200:8000
deposit -1
java.lang.IllegalArgumentException: cannot deposit -1
        at examples.orb.Account.deposit(Account.java:37)
        thrown remotely from 208.6.239.200:8000
>
```

Application voyager1.0.0\examples\orb\Account3.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.orb;

import COM.objectspace.voyager.*;

public class Account3
  {
  public static void main( String args[] )
    {
    try
      {
      // create a remote account with initial balance of 500
      VAccount account = new VAccount( 500, "localhost:8000" );

      try
        {
        System.out.println( "withdraw 1000" );
        account.withdraw( 1000 ); // withdraw too much
        System.out.println( "balance is " + account.getBalance() );
        }
      catch( NotEnoughMoneyException exception )
        {
        System.err.println( exception );
        }

      try
        {
        System.out.println( "deposit -1" );
        account.deposit( -1 ); // deposit too little
        System.out.println( "balance is " + account.getBalance() );
        }
      catch( IllegalArgumentException exception )
        {
        System.err.println( exception );
        }

      Voyager.shutdown();
      }
    catch ( VoyagerException exception )
      {
      System.err.println( exception );
      }
    }
  }
```

Storing and Passing Virtual References

A virtual reference can be stored and passed around like any other Java object. Because a virtual reference contains only the address of a remote object plus a small amount of bookkeeping information, storing and passing are efficient.

Use the static method `VObject.forObject( Object object )` to obtain a virtual reference to a simple local object. Although this method always returns the virtual instance as a `VObject`, Voyager looks for the most specific virtual class available for the given object's class. This class is instantiated and the instance is returned. Cast the instance to the appropriate class or interface.

The `Bank1.java` example program demonstrates the storing and passing of virtual references. The program creates a remote bank and two customers, allocates each customer a remote account, and then arranges for each customer to make a deposit into his account.

Use the following commands from the `\voyager1.0.0\examples\orb` directory to prepare and compile the `Bank1.java` program:

```
vcc Bank Customer
javac Bank.java VBank.java Customer.java VCustomer.java Bank1.java
```

Start a server on port 8000 in one window, and then run `Bank1.class` in a second window.

Window 1

```
>voyager 8000
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:8000
Construct with initial balance 0
Construct customer David
Construct with initial balance 0
deposit 1000
deposit 2000
balance of David is 2000
accounts are
  Account( 1000 )
  Account( 2000 )
```

Window 2

```
>java examples.orb.Bank1
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1115
Construct customer Graham
balance of Graham is 1000
>
```

Class voyager1.0.0\examples\orb\Bank.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.orb;

import java.util.Vector;
import COM.objectspace.voyager.*;

public class Bank
  {
  Vector accounts = new Vector(); // vector of bank accounts

  public VAccount newAccount() throws VoyagerException
    {
    Account account = new Account();
    accounts.addElement( account );

    // return a virtual reference to the account
    return (VAccount) VObject.forObject( account );
    }

  public void printAccounts()
    {
    System.out.println();
    System.out.println( "accounts are" );

    for( int i = 0; i < accounts.size(); i++ )
      System.out.println( "  " + accounts.elementAt( i ) );

    System.out.println();
    }
  }
```

Class voyager1.0.0\examples\orb\Customer.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.orb;

import COM.objectspace.voyager.*;

public class Customer
  {
  String name; // name of customer
  VAccount account; // reference to remote account

  public Customer( String name )
    {
    this.name = name;
    System.out.println( "Construct customer " + name );
    }

  public void setAccount( VAccount account )
    {
    this.account = account;
    }

  public void deposit( int amount ) throws VoyagerException
    {
    account.deposit( amount );
    System.out.println(
      "balance of " + name + " is " + account.getBalance() );
    }
  }
```

Application voyager1.0.0\examples\orb\Bank1.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.orb;

import COM.objectspace.voyager.*;

public class Bank1
  {
  public static void main( String args[] )
    {
    try
      {
      // create a bank in a remote program
      VBank bank = new VBank( "localhost:8000" );

      // create customer in local program and allocate remote account
      VCustomer customer1 = new VCustomer( "Graham", "localhost" );
      customer1.setAccount( bank.newAccount() );

      // create customer in remote program and allocate remote account
      VCustomer customer2 = new VCustomer( "David", "localhost:8000" );
      customer2.setAccount( bank.newAccount() );

      // make some deposits
      customer1.deposit( 1000 );
      customer2.deposit( 2000 );

      // display both accounts
      bank.printAccounts();

      Voyager.shutdown();
      }
    catch( VoyagerException exception )
      {
      System.err.println( exception );
      }
    }
  }
```

Connecting to an Existing Object

Distributed programs often create an object for another program to use. Voyager allows you to connect to such an object using its alias or GUID.

Connecting via an Alias

The simplest way to connect to an object is through its alias. Voyager provides the option of assigning an alias to any object during construction of the object or by using `setAlias()` to assign an alias after construction. To assign an alias to an object during construction, follow the program address with a slash (/) and the desired alias. Numeric aliases cannot contain a dash; for example, `123-45` is an invalid alias. More than one object in the same program can have the same alias.

To obtain a virtual reference to an object via its alias, pass the object's address with a slash (/) and the desired alias to the static method `VObject.forObjectAt( String` *address* `)`. If more than one object in the target program has the same alias, an `AmbiguousAliasException` is thrown.

Use the following command from the `\voyager1.0.0\examples\orb` directory to compile the `Alias1A.java` and `Alias1B.java` example programs:

```
javac Alias1A.java Alias1B.java
```

Start a server on port 8000 in one window and run `Alias1A.class` in a second window. An object with the alias `MyAccount` is created in server 8000. Then run `Alias1B.class` in the second window.

Window 1

```
>voyager 8000
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:8000
Construct with initial balance 0
deposit 1000
deposit 500
```

Window 2

```
>java examples.orb.Alias1A
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1117
The account alias is MyAccount
deposit 1000
account = Account( 1000 )
>java examples.orb.Alias1B
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1119
deposit 500
account = Account( 1500 )
>
```

Application voyager1.0.0\examples\orb\Alias1A.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.orb;

import COM.objectspace.voyager.*;

public class Alias1A
  {
  public static void main( String args[] )
    {
    try
      {
      // create a remote account with the alias "MyAccount"
      VAccount account = new VAccount( "localhost:8000/MyAccount" );
      System.out.println( "The account alias is " + account.getAlias() );
      account.liveForever(); // don't garbage collect this account
      System.out.println( "deposit 1000" );
      account.deposit( 1000 );
      System.out.println( "account = " + account );
      Voyager.shutdown();
      }
    catch( VoyagerException exception )
      {
      System.err.println(exception);
      }
    }
  }
```

Application voyager1.0.0\examples\orb\Alias1B.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.orb;

import COM.objectspace.voyager.*;

public class Alias1B
  {
  public static void main( String args[] )
    {
    try
      {
      // connect to the aliased account
      VAccount account =
        (VAccount) VObject.forObjectAt( "localhost:8000/MyAccount" );
      System.out.println( "deposit 500" );
      account.deposit( 500 );
      System.out.println( "account = " + account );
      account.dieNow(); // destroy the account
      Voyager.shutdown();
      }
    catch( VoyagerException exception )
      {
      System.err.println( exception );
      }
    }
  }
```

Connecting via a GUID

Connecting to an existing object using its GUID-based address is possible, although not as convenient as using an alias.

Use the following command from the `\voyager1.0.0\examples\orb` directory to compile the `Connect1.java` example program:

```
javac Connect1.java
```

Start a server on port 8000 in one window, and then run `Connect1.class` in a second window.

Window 1

```
>voyager 8000
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:8000
Construct with initial balance 0
deposit 1000
deposit 500
```

Window 2

```
>java examples.orb.Connect1
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1121
deposit 1000
account1 = Account( 1000 )
connected to Account( 1000 )
deposit 500
account1 = Account( 1500 )
>
```

Application voyager1.0.0\examples\orb\Connect1.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.orb;

import COM.objectspace.voyager.*;

public class Connect1
  {
  public static void main( String args[] )
    {
    try
      {
      // create account in remote application
      VAccount account1 = new VAccount( "localhost:8000" );
      System.out.println( "deposit 1000" );
      account1.deposit( 1000 );
      System.out.println( "account1 = " + account1 );
      useAccount( account1.getObjectAddress() );
      System.out.println( "account1 = " + account1 );
      Voyager.shutdown();
      }
    catch( VoyagerException exception )
      {
      System.err.println( exception );
      }
    }

  static void useAccount( String address ) throws VoyagerException
    {
    // connect to existing account using its address
    VAccount account = (VAccount) VObject.forObjectAt( address );
    System.out.println( "connected to " + account );
    System.out.println( "deposit 500" );
    account.deposit( 500 );
    }
  }
```

Remote Arguments, Serialization, and Morphology

Arguments of a remote message are copied using the Java serialization† mechanism before they are sent to a different program. If an argument is not serializable, a `TransportException` is thrown.

Modifications made to a remote argument do not affect the original argument. If an argument is a virtual reference, all messages sent to that argument are automatically forwarded to the argument's associated remote object.

Voyager maintains morphology across remote method calls. An object that is an argument or part of an argument is copied exactly once to the remote program. An argument or part of an argument that shares an object in the local program also shares a copy of the object in the remote program.

Use the following commands from the `\voyager1.0.0\examples\orb` directory to prepare and compile the `Morphology1.java` example program:

```
vcc Comparator
javac Comparator.java VComparator.java Morphology1.java
```

Start a server on port 8000 in one window, and then run `Morphology1.class` in a second window.

Window 1

```
>voyager 8000
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:8000
a = 3, b = 3
a.equals( b ) = true
( a == b ) = false
a = 3, b = 3
a.equals( b ) = true
( a == b ) = true
```

Window 2

```
>java examples.orb.Morphology1
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1123
>
```

† Consult any Java language reference guide for information about serialization.

Class voyager1.0.0\examples\orb\Comparator.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.orb;

public class Comparator
  {
  public void compare( Object a, Object b )
    {
    System.out.println( "a = " + a + ", b = " + b );

    // compare the objects using equals()
    System.out.println( "a.equals( b ) = " + a.equals( b ) );

    // compare the objects using ==
    System.out.println( "( a == b ) = " + ( a == b ) );
    }
  }
```

Application voyager1.0.0\examples\orb\Morphology1.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.orb;

import COM.objectspace.voyager.*;

public class Morphology1
  {
  public static void main( String args[] )
    {
    try
      {
      VComparator comparator = new VComparator( "localhost:8000" );

      // compare two different Integers
      comparator.compare( new Integer( 3 ), new Integer( 3 ) );

      // compare an Integer against itself
      Integer three = new Integer( 3 );
      comparator.compare( three, three );
      Voyager.shutdown();
      }
    catch( VoyagerException exception )
      {
      System.err.println( exception );
      }
    }
  }
```

Cyclic References

Two or more classes often contain cyclic virtual references to each other. For example, a `Man` object can contain a virtual reference to his wife, and a `Woman` object can contain a virtual reference to her husband. In this kind of circular situation, `vcc` requires that all classes involved are processed together, on the same command line.

Use the following commands from the `\voyager1.0.0\examples\orb` directory to prepare and compile the `Marriage1.java` example program:

```
vcc Man Woman
javac Man.java Woman.java VMan.java VWoman.java Marriage1.java
```

Start a server on port 8000 in one window, and then run `Marriage1.class` in a second window.

Window 1

```
>voyager 8000
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:8000
Construct woman with name Bertha
set husband of Bertha to Fred
```

Window 2

```
>java examples.orb.Marriage1
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1147
Construct man with name Fred
set wife of Fred to Bertha
Man( Fred married to Bertha )
Woman( Bertha married to Fred )
>
```

Class voyager1.0.0\examples\orb\Man.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.orb;

import COM.objectspace.voyager.*;

public class Man
  {
  String name;
  VWoman wife; // circular reference

  public Man( String name )
    {
    this.name = name;
    System.out.println( "Construct man with name " + name );
    }

  public String toString()
    {
    String s = new String();

    try
      {
      s = "Man( " + name + " married to " + wife.getName() + " )";
      }
    catch( VoyagerException exception)
      {
      s = "Exception";
      }

    return s;
    }

  public String getName()
    {
    return name;
    }

  public void setWife( VWoman woman ) throws VoyagerException
    {
    System.out.println( "set wife of " + name + " to " + woman.getName() );
    wife = woman;
    }
  }
```

Class voyager1.0.0\examples\orb\Woman.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.orb;

import COM.objectspace.voyager.*;

public class Woman
  {
  String name;
  VMan husband; // circular reference

  public Woman( String name )
    {
    this.name = name;
    System.out.println( "Construct woman with name " + name );
    }

  public String toString()
    {
    String s = new String();

    try
      {
      s = "Woman( " + name + " married to " + husband.getName() + " )";
      }
    catch( VoyagerException exception )
      {
      s = "Exception";
      }
    return s;
    }

  public String getName()
    {
    return name;
    }

  public void setHusband( VMan man ) throws VoyagerException
    {
    System.out.println( "set husband of " + name + " to " + man.getName() );
    husband = man;
    }
  }
```

Application voyager1.0.0\examples\orb\Marriage1.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.orb;

import COM.objectspace.voyager.*;

public class Marriage1
  {
  public static void main( String args[] )
    {
    try
      {
      VMan man = new VMan( "Fred", "localhost" );
      VWoman woman = new VWoman( "Bertha", "localhost:8000" );
      man.setWife( woman );
      woman.setHusband( man );
      System.out.println( man );
      System.out.println( woman );
      Voyager.shutdown();
      }
    catch( VoyagerException exception )
      {
      System.err.println( exception );
      }
    }
  }
```

Inheritance and Polymorphism

When you send a message to an object via a virtual reference to one of the object's base classes, the message is executed correctly according to the standard rules of polymorphism. For example, consider a hierarchy of employees in which the method `workHarder()` is defined in `Employee` and overridden in its subclasses. Sending `workHarder()` to a `VEmployee` causes the correct version of `workHarder()` to be executed on the remote employee, regardless of whether the employee is a `Programmer` or a `Manager`.

Use the following commands from the `\voyager1.0.0\examples\orb` directory to prepare and compile the `Inheritance1.java` example program:

```
vcc Employee Programmer Manager
javac Employee.java Programmer.java Manager.java
javac VEmployee.java VProgrammer.java VManager.java Inheritance1.java
```

Start a destination program on port 8000 in one window, and then run `Inheritance1.class` in a second window.

Window 1

```
>voyager 8000
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:8000
construct class examples.orb.Programmer Steve
construct class examples.orb.Manager Mary
construct class examples.orb.Manager Doug
Doug: pass me the whip
Doug tells examples.orb.Manager( Mary ) to work harder
Mary: pass me the whip
Mary tells examples.orb.Programmer( Steve ) to work harder
Steve: pass me the Jolt and I'll hack harder
Mary tells examples.orb.Programmer( Dave ) to work harder
```

Window 2

```
>java examples.orb.Inheritance1
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1150
construct class examples.orb.Programmer Dave
Dave: pass me the Jolt and I'll hack harder
>
```

Class voyager1.0.0\examples\orb\Employee.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.orb;

public abstract class Employee
  {
  String name;

  public Employee( String name )
    {
    this.name = name;
    System.out.println( "construct " + getClass() + " " + name );
    }

  public String toString()
    {
    return getClass().getName() + "( " + name + " )";
    }

  public abstract void workHarder();
  }
```

Class voyager1.0.0\examples\orb\Programmer.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.orb;

public class Programmer extends Employee
  {
  public Programmer( String name )
    {
    super( name );
    }

  public void workHarder()
    {
    System.out.println( name + ": pass me the Jolt and I'll hack harder" );
    }
  }
```

Class voyager1.0.0\examples\orb\Manager.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.orb;

import java.util.Vector;
import COM.objectspace.voyager.*;

public class Manager extends Employee
  {
  Vector employees = new Vector();

  public Manager( String name )
    {
    super( name );
    }

  public void addEmployee( VEmployee employee )
    {
    employees.addElement( employee );
    }

  public void workHarder()
    {
    System.out.println( name + ": pass me the whip" );

    for( int i = 0; i < employees.size(); i++ )
      {
      try
        {
        VEmployee subordinate = (VEmployee) employees.elementAt( i );
        System.out.println( name + " tells " + subordinate
                            + " to work harder" );
        subordinate.workHarder();
        }
      catch( VoyagerException exception )
        {
        System.err.println( exception );
        }
      }
    }
  }
```

Application voyager1.0.0\examples\orb\Inheritance1.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.orb;

import COM.objectspace.voyager.*;

public class Inheritance1
  {
  public static void main( String args[] )
    {
    try
      {
      // create two programmers
      VProgrammer programmer1 = new VProgrammer( "Steve", "localhost:8000" );
      VProgrammer programmer2 = new VProgrammer( "Dave", "localhost" );

      // create a manager to manage the programmers
      VManager manager1 = new VManager( "Mary", "localhost:8000" );
      manager1.addEmployee( programmer1 );
      manager1.addEmployee( programmer2 );

      // create a manager to manage the manager
      VManager manager2 = new VManager( "Doug", "localhost:8000" );
      manager2.addEmployee( manager1 );

      // tell the top manager to work harder
      manager2.workHarder();
      Voyager.shutdown();
      }
    catch( VoyagerException exception )
      {
      System.err.println( exception );
      }
    }
  }
```

Interfaces

The need often arises for a function that can operate on either a local object or a remote object. Voyager allows you to create such a function with Java interfaces. When you process a class using `vcc`, all interfaces implemented by the original class are also implemented by the virtual class, enabling variables having those interface types to accept an instance of the original class or its virtual class.

The `Interface1.java` example program defines an interface `Calculator` and a class `CalculatorImpl` that implements `Calculator`. Because `VCalculatorImpl` also implements `Calculator`, the function `calculate()` can be written to accept either a local reference or a remote reference.

Use the following commands from the `\voyager1.0.0\examples\orb` directory to prepare and compile the `Interface1.java` example program:

```
vcc CalculatorImpl
javac CalculatorImpl.java VCalculatorImpl.java Interface1.java
```

Start a server on port 8000 in one window, and then run `Interface1.class` in a second window.

Window 1

```
>voyager 8000
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:8000
construct calculator
```

Window 2

```
>java examples.orb.Interface1
construct calculator
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1154
use local calculator...
2 + 3 = 5
2 - 3 = -1
use remote calculator...
2 + 3 = 5
2 - 3 = -1
>
```

Interface voyager1.0.0\examples\orb\Calculator.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.orb;

public interface Calculator
  {
  int add( int x, int y );
  int subtract( int x, int y );
  }
```

Class voyager1.0.0\examples\orb\CalculatorImpl.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.orb;

public class CalculatorImpl implements Calculator
  {
  public CalculatorImpl()
    {
    System.out.println( "construct calculator" );
    }

  public int add( int x, int y )
    {
    return x + y;
    }

  public int subtract( int x, int y )
    {
    return x - y;
    }
  }
```

Application voyager1.0.0\examples\orb\Interface1.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.orb;

import COM.objectspace.voyager.*;

public class Interface1
  {
  public static void main( String args[] )
    {
    try
      {
      Calculator calculator1 = new CalculatorImpl();
      Calculator calculator2 = new VCalculatorImpl( "localhost:8000" );
      System.out.println( "use local calculator..." ) ;
      calculate( calculator1 ); // calculate using local calculator
      System.out.println( "use remote calculator..." ) ;
      calculate( calculator2 ); // calculate using remote calculator
      Voyager.shutdown();
      }
    catch( Exception exception )
      {
      System.out.println( exception );
      }
    }

  static void calculate( Calculator calculator )
    {
    System.out.println( "2 + 3 = " + calculator.add( 2, 3 ) );
    System.out.println( "2 - 3 = " + calculator.subtract( 2, 3 ) );
    }
  }
```

Exceptions with Interfaces

The `vcc` utility must automatically detect when a class implements an interface with methods that do not declare `VoyagerException` in their `throws` clauses. For the virtual class to implement the interface, Voyager cannot add `VoyagerException` to the `throws` clause of these methods. Instead, `vcc` generates the class in such a way that this type of method throws each Voyager-related exception wrapped in a Voyager `RuntimeException`. If desired, use the `-r` option of `vcc` to make all methods throw run-time exceptions.

The rationale behind this approach is that many times an interface, like the `Runnable` interface, declares only one or two methods. The `vcc` utility makes all methods use the safe (not the run-time) exception policy except for those methods that must be used to implement the interface. Even these methods are safe if developers catch Voyager run-time exceptions when invoking them.

Sometimes an interface is written with distributed computing in mind, and the interface methods declare `VoyagerException` in their `throws` clauses. Voyager facilitates doing so with the `vcc -o` option, which instructs `vcc` to generate an interface containing the public methods of a class. These methods declare `VoyagerException` in their `throws` clauses. As a result, developers can program to both local and remote implementations of the interface without sacrificing the safety of compile-time exceptions. See “The vcc Utility” on page 47 for more details on the `-o` option.

Life Spans and Garbage Collection

Voyager object life spans come in two varieties: life spans that depend on the existence of references and those that do not. By default, a simple remote object has a reference-based life span—it is garbage-collected when all its local and virtual references have been destroyed—and an agent has a fixed life span—it lives for one day and is then garbage-collected. However, agent-enhanced distributed computing requires flexibility. You might want to create an agent that lives forever or a simple object that lives until it is explicitly told to die. Voyager provides a rich set of functions for controlling the life span of an object and implements these functions in a manner well-suited for mobile, distributed systems.

Reference-Based Life Spans

A simple object has a reference-based life span by default because, by default, it calls `dieWhenNoReferences()`. Reference-based life spans use a keep-alive strategy that is well-suited for mobile, distributed computing. References to a remote object are responsible for keeping it alive in one of two ways. The first way is by sending messages. Messages alone (without extra heartbeats or pings) are intelligent enough to keep the object from dying because Voyager uses `SmartMessenger` objects to deliver messages. The second way of keeping an object alive is by sending heartbeats. A virtual reference automatically pulses one-way heartbeat messages to its remote object every 60 seconds, but only if the reference is not being used to send messages to the object. Thus, in practice, heartbeat traffic is minimized. When a virtual reference goes out of scope, it ceases to keep its remote object alive.

While a virtual reference sends messages or heartbeats to its remote object, the Voyager program containing the remote object verifies that its objects should still be alive. Periodically, each remote object is asked to check if it has recently received a message—either a normal program message or a heartbeat message. If not, the object is sent `dieNow()`, which deregisters it from the naming service and allows it to be reclaimed by the Java virtual machine. Normally, an object that has not received a message or heartbeat within 90 seconds is sent `dieNow()`.

To make an object die after a specific period of inactivity, send `dieWhenInactiveFor()` to the object, specifying the period in milliseconds. The `dieNow()` message is sent to the object if the object does not receive a message within the specified period of time.

The `dieNow()` method generates an `ObjectEvent.DYING` event before completing. If an object needs to perform a cleanup operation, it can listen for this event and take the necessary action. See Chapter 7, “Events, Listeners, and Assistants,” for more information.

Send `becomeWeak()` to a virtual reference to disable its heartbeat pulse. This method turns the virtual reference into a weak reference; that is, the virtual reference can access its remote object as long as it is alive, but the reference does not prevent the remote object from being garbage-collected.

Fixed Life Spans

Although reference-based garbage collection is important for programs that create and share transient remote objects, many applications require remote objects to live regardless of associated virtual references. For example, reference-based garbage collection usually works poorly for service applications, such as weather services, streaming video services, and electronic stores. For these kinds of applications, Voyager allows you to fix an object's life span.

An agent has a fixed, one-day life span by default because, by default, it calls `dieAfter()`. To prevent a remote object from ever being garbage-collected, send it `liveForever()`. References to such an object generate no heartbeat traffic. To make such an object die, send the `dieNow()` message.

To make an object die after a specific time period has elapsed, send the `dieAfter()` message to the object, specifying the time period in milliseconds. The `dieNow()` message is sent to the object after the specified period of time passes regardless of whether or not the object has been receiving messages, so its virtual references do not generate heartbeat traffic.

To make an object die at a particular point in time, send the object the `dieAt()` message, specifying the date the object should die, that is, an instance of `java.util.Date`. References do not send heartbeats to objects with fixed die times.

Dynamic Reference Updating

When an object's life span changes, the new value is propagated to all of the object's virtual references via an efficient update-on-demand technique. Every time a virtual reference sends a message to its remote object, the message is tagged with the reference's current understanding of the remote object's life span. When the message arrives, the life span information is compared to the remote object's actual life span information. If the information differs, the new information is either piggy backed onto the message return value (for synchronous and future messages) or returned to the virtual reference by an explicit message (for one-way messages, like heartbeats).

Scalability of the Garbage Collection Model

Voyager's hybrid distributed garbage collection mechanism is well-suited for distributed computing, not only because of its flexibility, but also because of its scalability. Pure lease-based or heartbeat-based systems generate more network traffic than this hybrid approach because, with Voyager, a remote object generally falls into one of two categories:

- A remote object is created to perform relatively few tasks and is never used again. In this scenario, the object's virtual reference is accessed fairly frequently and then destroyed, so no heartbeat traffic is generated.
- A remote object is created to service other virtual references that connect to the object via a well-known alias or trading service. Such an object is sent `liveForever()`. In this scenario, the object's virtual references know they are connected to an immortal object and do not generate heartbeat traffic.

Therefore, heartbeats are generated only when a remote object is created with a short or medium life span, and the object is not sent frequent messages. In practice, this scenario is rare, so the amount of heartbeat traffic is relatively small.

Use the following commands from the `\voyager1.0.0\examples\orb` directory to prepare and compile the `Lifespan1A.java` and `Lifespan1B.java` example programs:

```
vcc Submarine
javac Submarine.java VSubmarine.java Lifespan1A.java Lifespan1B.java
```

Start a server on port 8000 in one window and run `Lifespan1A.class` in a second window. A few seconds later, run `Lifespan1B.class` in a third window. `Lifespan1B` waits a few moments, and then kills the submarine being pinged by `Lifespan1A`. When `Lifespan1A` attempts to send a message to the dead submarine, an `ObjectNotFoundException` is thrown.

Window 1

```
>voyager 8000
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:8000
ping!
```

Window 2

```
>java examples.orb.Lifespan1A
create the red october
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1156
tell the red october to live forever...
sleep for 10 seconds...
ping red october...
sleep for 100 seconds...
ping red october...
COM.objectspace.voyager.ObjectNotFoundException:
237-131-214-136-204-109-241-21-28-248-72-134-111-72-159-192
        thrown remotely from 208.6.239.200:8000
>
```

Window 3

```
>java examples.orb.Lifespan1B
hunt for the red october...
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1158
sleep for 20 seconds...
kill the red october...
>
```

Class voyager1.0.0\examples\orb\Submarine.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.orb;

public class Submarine
  {
  public void ping()
    {
    System.out.println( "ping!" );
    }
  }
```

Class voyager1.0.0\examples\orb\Lifespan1A.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.orb;

import COM.objectspace.voyager.*;

public class Lifespan1A
  {
  public static void main( String args[] )
    {
    try
      {
      System.out.println( "create the red october" );
      VSubmarine submarine = new VSubmarine( "localhost:8000/Red" );
      System.out.println( "tell the red october to live forever..." );
      submarine.liveForever();
      System.out.println( "sleep for 10 seconds..." );

      try
        {
        Thread.sleep( 10000 );
        }
      catch( InterruptedException exception )
        {
        }

      System.out.println( "ping red october..." );
      submarine.ping();
      System.out.println( "sleep for 100 seconds..." );

      try
        {
        Thread.sleep( 100000 );
        }
      catch( InterruptedException exception )
        {
        }

      System.out.println( "ping red october..." );
      submarine.ping();
      Voyager.shutdown();
      }
    catch( VoyagerException exception )
      {
      System.out.println( exception );
      }
    }
  }
```

Class voyager1.0.0\examples\orb\Lifespan1B.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.orb;

import COM.objectspace.voyager.*;

public class Lifespan1B
  {
  public static void main( String args[] )
    {
    try
      {
      System.out.println( "hunt for the red october..." );
      VSubmarine submarine =
        (VSubmarine) VObject.forObjectAt( "localhost:8000/Red" );
      System.out.println( "sleep for 20 seconds..." );

     try{ Thread.sleep( 15000 ); } catch( InterruptedException exception ) {}

      System.out.println( "kill the red october..." );
      submarine.dieNow();
      Voyager.shutdown();
      }
    catch( VoyagerException exception )
      {
      System.out.println( exception );
      }
    }
  }
```

The `Lifespan2.java` example program demonstrates the `dieAfter()` and `dieAt()` messages. From the `\voyager1.0.0\examples\orb` directory, use the following commands to compile `Lifespan2.java`:

```
javac Lifespan2.java
```

Start a server on port 8000 in one window, and then run `Lifespan2.class` in a second window.

Window 1

```
>voyager 8000
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:8000
ping!
ping!
ping!
ping!
ping!
```

Window 2

```
>java examples.orb.Lifespan2
create the red october
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1160
tell the red october to die after 10 seconds...
ping red october...
sleep for 3 seconds...
ping red october...
sleep for 3 seconds...
ping red october...
sleep for 3 seconds...
ping red october...
sleep for 3 seconds...
ping red october...
sleep for 3 seconds...
ping red october...
COM.objectspace.voyager.ObjectNotFoundException:
126-140-221-153-165-196-35-131-
162-205-72-134-111-72-159-192
        thrown remotely from 208.6.239.200:8000
tell blue to die on Mar 12, 1998, at 7:00am...
>
```

Application voyager1.0.0\examples\orb\Lifespan2.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.orb;

import java.util.Calendar;
import COM.objectspace.voyager.*;

public class Lifespan2
  {
  public static void main( String args[] )
    {
    try
      {
      System.out.println( "create the red october" );
      VSubmarine submarine1 = new VSubmarine( "localhost:8000/Red" );
     System.out.println( "tell the red october to die after 10 seconds..." );
      submarine1.dieAfter( 10000 ); // die in 10 seconds time.

      try
        {
        while( true ) // loop until the submarine dies
          {
          System.out.println( "ping red october..." );
          submarine1.ping();
          System.out.println( "sleep for 3 seconds..." );

          try{Thread.sleep( 3000 );} catch(InterruptedException exception) {}
          }
        }
      catch( ObjectNotFoundException exception )
        {
        System.out.println( exception );
        }

      VSubmarine submarine2 = new VSubmarine( "localhost:8000/Blue" );
      Calendar calendar = Calendar.getInstance();
      calendar.set( 1998, Calendar.MARCH, 12, 7, 0 );
      System.out.println( "tell blue to die on Mar 12, 1998, at 7:00am..." );
      submarine2.dieAt( calendar.getTime() );
      Voyager.shutdown();
      }
    catch( VoyagerException exception )
      {
      System.out.println( exception );
      }
    }
  }
```

Properties

With Voyager, you can add and remove arbitrary key/value pairs, called *properties*, to and from an object via a virtual reference. Voyager's publish/subscribe mechanism, described in Chapter 15, "Space: Scalable Group Communication," demonstrates one use of this feature.

The `VObject` class includes the following methods for manipulating an object's properties:

- `void addProperty( Object` *key*`, Object` *value* `)`

  Add the specified key/value pair to the object's property map.

- `void putProperty( Object` *key*`, Object` *value* `)`

  Add the specified key/value pair to the object's property map, replacing the first key/value with a matching key if one already exists.

- `void removeProperty( Object` *key* `)`

  Remove every matching key/value pair.

- `void removeProperty( Object` *key*`, Object` *value* `)`

  Remove every matching key/value pair. Both the key and the value must match.

- `Object getProperty( Object` *key* `)`

  Return the value of the first matching key/value pair.

- `Vector getProperties( Object` *key* `)`

  Return the values of all matching key/value pairs.

- `Vector getPropertyPairs()`

  Return clones of the key/value pairs. Every element of the vector is an `Object[]`, the first of which is the key and the second of which is the value.

The `Properties.java` example program demonstrates the use of these methods. Use the following commands from the `\voyager1.0.0\examples\orb` directory to prepare and compile `Properties.java`.

```
vcc Consumer
javac VConsumer.java Consumer.java Properties.java
```

Now run `Properties.class`:

Window 1

```
>java examples.orb.Properties
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1162
initial, property list =
put subscribe, property list =
  subscribe -> sports.bulls
put subscribe, property list =
  subscribe -> sports.lakers
add age and subscribe, property list =
  subscribe -> sports.lakers
  age -> 34
  subscribe -> sports.bulls
  subscribe -> sports.mavericks
first subscribe -> sports.lakers
all subscribe -> [sports.lakers, sports.bulls, sports.mavericks]
remove sports.bulls, property list =
  subscribe -> sports.lakers
  age -> 34
  subscribe -> sports.mavericks
remove subscribe, property list =
  age -> 34
>
```

Class voyager1.0.0\examples\orb\Consumer.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.orb;

public class Consumer
  {
  }
```

Application voyager1.0.0\examples\orb\Properties.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.orb;

import java.util.Vector;
import COM.objectspace.voyager.*;

public class Properties
  {
  public static void main( String args[] )
    {
    try
      {
      VConsumer consumer = new VConsumer( "localhost" );
```

```
    display( "initial", consumer.getPropertyPairs() );

    consumer.putProperty( "subscribe", "sports.bulls" );
    display( "put subscribe", consumer.getPropertyPairs() );

    consumer.putProperty( "subscribe", "sports.lakers" );
    display( "put subscribe", consumer.getPropertyPairs() );

    consumer.addProperty( "age", new Integer( 34 ) );
    consumer.addProperty( "subscribe", "sports.bulls" );
    consumer.addProperty( "subscribe", "sports.mavericks" );
    display( "add age and subscribe", consumer.getPropertyPairs() );

    System.out.println(
      "first subscribe -> " + consumer.getProperty( "subscribe" ) );
    System.out.println(
      "all subscribe -> " + consumer.getProperties( "subscribe" ) );

    consumer.removeProperty( "subscribe", "sports.bulls" );
    display( "remove sports.bulls", consumer.getPropertyPairs() );

    consumer.removeProperty( "subscribe" );
    display( "remove subscribe", consumer.getPropertyPairs() );

    Voyager.shutdown();
    }
  catch( VoyagerException exception )
    {
    System.err.println( exception );
    }
  }

public static void display( String prompt, Vector pairs )
  {
  System.out.println( prompt + ", property list = " );

  for( int i = 0; i < pairs.size(); i++ )
    {
    Object[] pair = (Object[]) pairs.elementAt( i );
    System.out.println( "  " + pair[ 0 ] + " -> " + pair[ 1 ] );
    }
  }
}
```

6

Advanced Messaging

This chapter describes the advanced messaging features of the ObjectSpace Voyager™ Core Technology (Voyager). By default, Voyager constructors and messages are synchronous and never time out. Greater flexibility is often needed in many applications, however, so Voyager provides complete control over timeout values and message characteristics.

Timeouts

Though most programs should wait as long as necessary for a command to complete, some programs require remote invocations to fail if not completed within a given time period. To change the timeout value for messages sent via a virtual reference, send the virtual reference the `setMessageTimeout()` message. Subsequent messages sent using the virtual reference use the new timeout value. Operations that take longer to complete than the new timeout period cause a `TimeoutException` to be thrown.

To get the current timeout value for a message sent through a given reference, invoke `getMessageTimeout()` on the reference. A value of zero, the default value, indicates messages sent via that reference never time out.

The `Timeout1.java` example program uses a `Timer` object to produce a predictable message delay and demonstrate dynamic timeouts. Use the following commands from the `\voyager1.0.0\examples\orb` directory to prepare and compile `Timeout1.java`:

```
vcc Timer
javac Timer.java VTimer.java Timeout1.java
```

Start a server on port 8000 in one window, and then run `Timeout1.class` in a second window.

Window 1

```
>voyager 8000
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:8000
Construct Timer
start countdown( 65000 )
start countdown( 65000 )
finish countdown( 65000 )
finish countdown( 65000 )
```

Window 2

```
>java examples.orb.Timeout1
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1163
default timeout is 0ms
changing to 60000ms
invoke timer.countdown( 65000 )
COM.objectspace.voyager.TimeoutException: 60000ms elapsed
change timeout to 80s
new timeout is 80000ms
invoke timer.countdown( 65000 )
Done
>
```

Class voyager1.0.0\examples\orb\Timer.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.orb;

public class Timer
  {
  int x;
  int y;

  public Timer()
    {
    System.out.println( "Construct Timer" );
    }

  public Timer( int delay )
    {
    System.out.println( "construct( " + delay + " )" );
    try{ Thread.sleep( delay ); } catch( InterruptedException exception ) {}
    System.out.println( "finish construct" );
    }

  public String countdown( int delay )
    {
    System.out.println( "start countdown( " + delay + " )" );

    if( delay < 0 )
      throw new IllegalArgumentException( "delay cannot be less than 0ms" );

    try{ Thread.sleep( delay ); } catch( InterruptedException exception ) {}

    System.out.println( "finish countdown( " + delay + " )" );
    return "Done";
    }

  static public String waitFor( int delay )
    {
    System.out.println( "start waitFor( " + delay + " )" );

    try{ Thread.sleep( delay ); } catch( InterruptedException exception ) {}

    System.out.println( "finish waitFor( " + delay + " )" );
    return "Done";
    }
  }
```

Application voyager1.0.0\examples\orb\Timeout1.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.orb;

import COM.objectspace.voyager.*;

public class Timeout1
  {
  public static void main( String args[] )
    {
    try
      {
      VTimer timer = new VTimer( "localhost:8000" );
      System.out.println(
        "default timeout is " + timer.getMessageTimeout() + "ms" );

      System.out.println( "changing to 60000ms" );
      timer.setMessageTimeout( 60000 );
      try
        {
        // force a timeout to occur
        System.out.println( "invoke timer.countdown( 65000 )" );
        String string = timer.countdown( 65000 );
        System.out.println( string );
        }
      catch( Exception exception )
        {
        System.out.println( exception );
        }

      System.out.println( "change timeout to 80s" );
      timer.setMessageTimeout( 80000 );
      System.out.println( "new timeout is "+timer.getMessageTimeout()+"ms" );

      try
        {
        // a timeout will not occur now
        System.out.println( "invoke timer.countdown( 65000 )" );
        String string = timer.countdown( 65000 );
        System.out.println( string );
        }
      catch( Exception exception )
        {
        System.out.println( exception );
        }

      Voyager.shutdown();
      }
    catch( VoyagerException exception )
      {
      System.err.println( exception );
      }
    }
  }
```

As demonstrated in the `Timeout2.java` example program, you can override the default timeout for a constructor by adding the required timeout value after the address parameter. The virtual reference created inherits the timeout value from the constructor.

Use the following command from the `\voyager1.0.0\examples\orb` directory to compile `Timeout2.java`:

```
javac Timeout2.java
```

Start a server on port 8000 in one window, and then run `Timeout2.class` in a second window.

Window 1

```
>voyager 8000
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:8000
construct( 65000 )
construct( 65000 )
finish construct
finish construct
start countdown( 65000 )
finish countdown( 65000 )
```

Window 2

```
>java examples.orb.Timeout2
construct with 60s timeout
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1165
COM.objectspace.voyager.TimeoutException: 60000ms elapsed
construct with timeout of 80s
ok
inherited timeout is 80000
invoke countdown( 65000 )
countdown( 65000 ) = Done
>
```

Application voyager1.0.0\examples\orb\Timeout2.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.orb;

import COM.objectspace.voyager.*;

public class Timeout2
  {
  public static void main( String args[] )
    {
    try
      {
      // force a timeout to occur
      System.out.println( "construct with 60s timeout" );
      VTimer timer = new VTimer( 65000, "localhost:8000", 60000 );
      System.out.println( "ok" );
      }
    catch( Exception exception )
      {
      System.out.println( exception );
      }

    try
      {
      System.out.println( "construct with timeout of 80s" );
      VTimer timer = new VTimer( 65000, "localhost:8000", 80000 );
      System.out.println( "ok" );
      System.out.println( "inherited timeout is "+timer.getMessageTimeout()
);
      System.out.println( "invoke countdown( 65000 )" );
      String string = timer.countdown( 65000 );
      System.out.println( "countdown( 65000 ) = " + string );
      }
    catch( Exception exception )
      {
      System.out.println( exception );
      }

    Voyager.shutdown();
    }
  }
```

The `Timeout3.java` example program demonstrates that you can override the default timeout for a remote static method call by adding the required timeout value after the address parameter. Use the following command from the `\voyager1.0.0\examples\orb` directory to compile the `Timeout3.java` example program:

```
javac Timeout3.java
```

Start a server on port 8000 in one window, and then run `Timeout3.class` in a second window.

Window 1

```
>voyager 8000
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:8000
start waitFor( 65000 )
finish waitFor( 65000 )
start waitFor( 65000 )
finish waitFor( 65000 )
```

Window 2

```
>java examples.orb.Timeout3
invoke waitFor( 65000 ) with no timeout
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1167
Done
invoke waitFor( 65000 ) with timeout of 20s
COM.objectspace.voyager.TimeoutException: 20000ms elapsed
>
```

Application voyager1.0.0\examples\orb\Timeout3.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.orb;

import COM.objectspace.voyager.*;

public class Timeout3
  {
  public static void main( String args[] )
    {
    try
      {
      System.out.println( "invoke waitFor( 65000 ) with no timeout" );
      String string = VTimer.waitFor( 65000, "localhost:8000" );
      System.out.println( string );
      }
    catch( Exception exception )
      {
      System.out.println( exception );
      }

    try
      {
      System.out.println( "invoke waitFor( 65000 ) with timeout of 20s" );
      String string = VTimer.waitFor( 65000, "localhost:8000", 20000 );
      System.out.println( string );
      }
    catch( Exception exception )
      {
      System.out.println( exception );
      }

    Voyager.shutdown();
    }
  }
```

Thread Management

A Voyager messenger often requires a thread of its own. Because creating a new Java thread is expensive, Voyager reuses threads when possible. When Voyager needs a thread to perform an operation, it attempts to reuse a thread from its internal pool. If the pool is not empty, Voyager takes a thread from the pool, uses it to perform the operation, and then, if the pool is not full, returns it to the pool. If the pool is empty, Voyager creates a new thread, uses it to perform the operation, and, if the pool is not full, places the thread in the pool for future reuse.

Usually, the size of the thread pool stabilizes after a short time, after which Voyager rarely needs to create new threads. The maximum size of the thread pool can be set with the `setPoolSize()` method to optimize Voyager for a given platform. More powerful servers might need a large thread pool for rapid allocation of threads, and less powerful servers might need a smaller thread pool for memory conservation. The default thread pool size is 30.

A user can leverage Voyager's thread management functionality in code by invoking one of the `spawnThreadFor()` methods on Voyager and providing a `Runnable`. In this way, Voyager can track user threads and users can invoke new threads efficiently.

Smart Messengers

By default, Voyager messages are synchronous, but Voyager supports other message types. You can initiate remote operations and immediately continue without waiting for return results. You can send a future message if you want to get the result of the operation at a later time, or you can send a one-way message if you do not need the result at all. You can use a one-way multicast message to send a one-way message to a group of objects instead of a single object, and you can use a one-way multicast message with selectors to send a one-way message to a set of objects that satisfy a selection criterion.

In traditional distributed computing architectures, a remote message is typically sent in on-the-wire format to the destination program, at which point a preinstalled message router decodes the message and performs the upcall to the target object.

Voyager, on the other hand, delivers messages using smart messengers: autonomous mini-agents that encode messages and intelligently deliver them to remote objects. Smart messengers can navigate multiprotocol networks, resend periodically when network connections are down, and take special actions if messages cannot be delivered successfully. Features traditionally supplied as add-ons to object request brokers, such as store and forward, can be embedded directly in smart messengers.

Two important consequences of the smart messenger architecture are listed below:

- Messengers carry built-in capabilities as they move around a network, thus you need not install special software at each node to get features like store and forward and fault-tolerant messaging.
- Users can create and use custom messengers without modifying the underlying distributed computing architecture.†

Voyager includes a hierarchy of four messenger classes that support future, one-way, one-way multicast, and synchronous messaging.

`SmartMessenger` (implements `Messenger`)

```
Future
OneWay
OneWayMulticast
Sync
```

The `vcc` utility generates two versions of every remote method—one that uses a `Sync` messenger and one that allows developers to append a default-constructed instance of the required messenger to the message argument list. Messengers can be used only once; after delivering a message, a messenger is irrevocably modified such that it cannot deliver another message.

The next few sections describe how to use one-way, future, and synchronous messengers. Information about using one-way multicast messengers is presented in Chapter 15, “Space: Scalable Group Communication.”

† Voyager source is currently restricted to source licensees. Contact info@objectspace.com for more information.

One-Way Messengers

A message sent using a `OneWay` messenger immediately returns. A `OneWay` messenger performs “fire-and-forget” messaging—it allocates itself a thread from the Voyager thread pool, delivers the message to the remote object, and discards the reply.

Use the following command from the `\voyager1.0.0\examples\orb` directory to compile the `OneWay1.java` example program:

```
javac OneWay1.java
```

Start a server on port 8000 in one window, and then run `OneWay1.class` in a second window.

Window 1

```
>voyager 8000
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:8000
Construct Timer
start countdown( 10000 )
start countdown( 10000 )
start countdown( 10000 )
finish countdown( 10000 )
finish countdown( 10000 )
finish countdown( 10000 )
```

Window 2

```
>java examples.orb.OneWay1
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1170
>
```

Application voyager1.0.0\examples\orb\OneWay1.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.orb;

import COM.objectspace.voyager.*;

public class OneWay1
  {
  public static void main( String args[] )
    {
    try
      {
      VTimer Timer = new VTimer( "localhost:8000" );
      Timer.countdown( 10000, new OneWay() );
      Timer.countdown( 10000, new OneWay() );
      Timer.countdown( 10000, new OneWay() );

      // allow two seconds for the messages to leave this program
      try{ Thread.sleep( 2000 ); } catch( InterruptedException exception ) {}

      Voyager.shutdown();
      }
    catch( VoyagerException exception )
      {
      System.out.println( exception );
      }
    }
  }
```

Future Messengers

A message sent using a `Future` messenger immediately returns a `Result` object, which is a placeholder for the subsequent return value. A `Future` messenger allocates itself a thread from the Voyager thread pool, delivers the message to the remote object, and then sends the return value to the waiting `Result` object.

Use `isAvailable()` to determine whether a `Result` has received its return value.

To read the return value from a `Result`, use the appropriate version of `read()`. The message blocks until either the return value is received or the timeout period is exceeded. If the result is not received within the message's timeout period, a `TimeoutException` is thrown. The timeout countdown starts when `read()` is called, not when the message is actually sent.

If a remote exception occurs and you attempt to read the return value using `read()`, the exception is automatically rethrown. If the exception is a `RuntimeException` or `VoyagerException`, it is rethrown directly; otherwise, it is wrapped inside a `UserException` and rethrown. You can use `isException()` to determine whether a `Result` contains an exception and, if so, you can use `getException()` to get the exception.

If the return value of a `Future` messenger is unimportant, send `kill()` to the `Result` object. The `Future` messenger is sent a special remote message that tells the messenger to discard the return value instead of sending it across the network, thereby reducing network traffic.

Use the following command from the `\voyager1.0.0\examples\orb` directory to compile the `Future1.java` example program:

```
javac Future1.java
```

Start a server on port 8000 in one window, and then run `Future1.class` in a second window.

Window 1

```
>voyager 8000
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:8000
Construct Timer
start countdown( 5000 )
finish countdown( 5000 )
start countdown( 35000 )
start countdown( -1 )
start countdown( 25000 )
finish countdown( 35000 )
finish countdown( 25000 )
```

Window 2

```
>java examples.orb.Future1
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1172
about to send countdown( 5000 )
available = false
result = Done
available = true
exception = false
setting message timeouts to be 30000ms
about to send countdown( 35000 )
timer.countdown( 35000 ) -> COM.objectspace.voyager.TimeoutException:
30000ms elapsed
about to send countdown( -1 )
timer.countdown( -1 ) -> java.lang.IllegalArgumentException: delay cannot
be less than 0ms
        at examples.orb.Timer.countdown(Timer.java:27)
        thrown remotely from 208.6.239.200:8000
about to send countdown( 25000 )
telling future not to send reply...
>
```

Application voyager1.0.0\examples\orb\Future1.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.orb;

import COM.objectspace.voyager.*;

public class Future1
  {
  public static void main( String args[] )
    {
    try
      {
      VTimer timer = new VTimer( "localhost:8000" );

      try
        {
        System.out.println( "about to send countdown( 5000 )" );
        Result result = timer.countdown( 5000, new Future() );
        System.out.println( "available = " + result.isAvailable() );
        String string = (String) result.readObject();
        System.out.println( "result = " + string );
        System.out.println( "available = " + result.isAvailable() );
        System.out.println( "exception = " + result.isException() );
        }
      catch( Exception exception )
        {
        System.out.println( "timer.countdown( 5000 ) -> " + exception );
        }

      try
        {
        System.out.println( "setting message timeouts to be 30000ms" );
        timer.setMessageTimeout( 30000 );
        System.out.println( "about to send countdown( 35000 )" );
        Result result = timer.countdown( 35000, new Future() );
        String string = (String) result.readObject();
        System.out.println( "result = " + string );
        }
      catch( Exception exception )
        {
        System.out.println( "timer.countdown( 35000 ) -> " + exception );
        }

      try
        {
        System.out.println( "about to send countdown( -1 )" );
        Result result = timer.countdown( -1, new Future() );
        String string = (String) result.readObject();
        System.out.println( "result = " + string );
        }
      catch( Exception exception )
        {
        System.out.println( "timer.countdown( -1 ) -> " + exception );
```

```
      }

    System.out.println( "about to send countdown( 25000 )" );
    Result result = timer.countdown( 25000, new Future() );
    System.out.println( "telling future not to send reply..." );
    result.kill();

    // allow two seconds for the messages to leave this program
    try{ Thread.sleep( 2000 ); } catch( InterruptedException exception ) {}

    Voyager.shutdown();
    }
  catch( VoyagerException exception )
    {
    System.err.println( exception );
    }
  }
}
```

You can be notified when a future result arrives through the standard Java event/listener mechanism. When a result arrives, the `Result` sends `resultReceived()` with a `ResultEvent` object to every `ResultListener` that was either added to the `Future` object prior to sending the message or added to the `Result` object after the message was sent.

From the `\voyager1.0.0\examples\orb` directory, use the following command to compile the `Future2.java` example program:

```
javac Future2.java
```

Start a server on port 8000 in one window, and then run `Future2.class` in a second window.

Window 1

```
>voyager 8000
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:8000
Construct Timer
start countdown( 5000 )
finish countdown( 5000 )
```

Window 2

```
>java examples.orb.Future2
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1175
about to send countdown( 5000 )
listener gets result event
event source = UnicastResult( Reference(
208.6.239.200:8000/28-9-183-127-103-248-88-161-196-40-72-134-111-72-159-
192, 60, 872629366 ) )
object = Done
exception = false
source address =
208.6.239.200:8000/28-9-183-127-103-248-88-161-196-40-72-134-11
1-72-159-192
listener gets result event
event source = UnicastResult( Reference(
208.6.239.200:8000/28-9-183-127-103-248-88-161-196-40-72-134-111-72-159-
192, 60, 872629366 ) )
object = Done
exception = false
source address =
208.6.239.200:8000/28-9-183-127-103-248-88-161-196-40-72-134-111-72-
159-192
>
```

Application voyager1.0.0\examples\orb\Future2.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.orb;

import COM.objectspace.voyager.*;

public class Future2
  {
  public static void main( String args[] )
    {
    try
      {
      VTimer Timer = new VTimer( "localhost:8000" );
      System.out.println( "about to send countdown( 5000 )" );
      Future future = new Future();
      MyResultListener listener1 = new MyResultListener();
      future.addResultListener( listener1 ); // add prior to message send
      Result result = Timer.countdown( 5000, future );
      MyResultListener listener2 = new MyResultListener();
      result.addResultListener( listener2 ); // add after sending message
      }
    catch( VoyagerException exception )
      {
      System.err.println( exception );
      }
    }
  }

class MyResultListener implements ResultListener
  {
  static int count = 0;

  public void resultReceived( ResultEvent event )
    {
    System.out.println( "listener gets result event" );
    System.out.println( "event source = " + event.getSource() );
    System.out.println( "object = " + event.getObject() );
    System.out.println( "exception = " + event.isException() );
    System.out.println( "source address = " + event.getSourceAddress() );

    if( ++count == 2 )
      Voyager.shutdown();
    }
  }
```

More than one thread can invoke `read()` on a `Result`. When the `Result` receives the return value, all blocked threads are awakened and sent the value.

Use the following command from the `\voyager1.0.0\examples\orb` directory to compile the `Future3.java` example program:

```
javac Future3.java
```

Start a server on port 8000 in one window, and then run `Future3.class` in a second window.

Window 1

```
>voyager 8000
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:8000
Construct Timer
start countdown( 5000 )
finish countdown( 5000 )
```

Window 2

```
>java examples.orb.Future3
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1177
about to send countdown( 5000 )
waiting...
waiting...
reader thread gets Done
reader thread gets Done
>
```

Application voyager1.0.0\examples\orb\Future3.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.orb;

import COM.objectspace.voyager.*;

public class Future3
  {
  public static void main( String args[] )
    {
    try
      {
      VTimer Timer = new VTimer( "localhost:8000" );
      System.out.println( "about to send countdown( 5000 )" );
      Result result = Timer.countdown( 5000, new Future() );
      Thread thread1 = new ReaderThread( result );
      thread1.start();
      Thread thread2 = new ReaderThread( result );
```

```
      thread2.start();

      try
        {
        thread1.join();
        thread2.join();
        }
      catch( InterruptedException exception )
        {
        }

      Voyager.shutdown();
      }
    catch( VoyagerException exception )
      {
      System.err.println( exception );
      }
    }
  }

class ReaderThread extends Thread
  {
  Result result;

  ReaderThread( Result result )
    {
    this.result = result;
    }

  public void run()
    {
    System.out.println( "waiting..." );

    try
      {
      System.out.println( "reader thread gets " + result.readObject() );
      }
    catch( VoyagerException exception )
      {
      System.err.println( exception );
      }
    }
  }
```

Synchronous Messengers

A message sent using a `Sync` messenger returns a `Result` object only after the return value has been received. A `Sync` uses the current thread to deliver the message to the remote object, and then sends the reply to the waiting `Result` object.

Use the following command from the `\voyager1.0.0\examples\orb` directory to compile the `Sync1.java` example program:

```
javac Sync1.java
```

Start a server on port 8000 in one window, and then run `Sync1.class` in a second window.

Window 1

```
>voyager 8000
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:8000
Construct Timer
start countdown( 5000 )
finish countdown( 5000 )
```

Window 2

```
>java examples.orb.Sync1
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1179
read object...
countdown( 5000 ) = Done
>
```

Application voyager1.0.0\examples\orb\Sync1.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.orb;

import COM.objectspace.voyager.*;

public class Sync1
  {
  public static void main( String args[] )
    {
    try
      {
      VTimer Timer = new VTimer( "localhost:8000" );
      Sync sync = new Sync();
      Result result = Timer.countdown( 5000, sync );
      System.out.println( "read object..." );
      System.out.println( "countdown( 5000 ) = " + result.readObject() );
      Voyager.shutdown();
      }
    catch( VoyagerException exception )
      {
      System.err.println( exception );
      }
    }
  }
```

Dynamic Invocation

The `vcc` utility's dual-method-version approach (discussed in "Smart Messengers" on page 106) facilitates sending messages to a remote object using a smart messenger, but does not enable constructing and sending arbitrary messages dynamically. To do so, construct an instance of the required smart messenger, use `setSignature()` to set the message signature, use the `write()` functions to add the message arguments, and then use one of the `send()` methods of `VObject` to send the message.

The message signature must be written in standard Java virtual machine method format, as shown:

```
setSignature( "methodName( argument types ) [return value type]" )
```

where *argument types* is a non-delimited list of one or more of the following codes and *return value type*, an optional argument, is one of the following codes.

Type	Code
`boolean`	Z
`char`	C
`double`	D
`float`	F
`int`	I
`long`	J
`short`	S
`void`	V
object	L<*full class name*>;
array of type	[<*type*>

The `Dynamic1.java` example program uses the dynamic invocation feature to create and send three instance messages and one class message to an `Adder`. Use the following commands from the `\voyager1.0.0\examples\orb` directory to prepare and compile the program:

```
vcc Adder
javac Adder.java VAdder.java Dynamic1.java
```

Start a server on port 8000 in one window, and then run `Dynamic1.class` in a second window.

Window 1

```
>voyager 8000
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:8000
Construct Adder
```

Window 2

```
>java examples.orb.Dynamic1
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1181
adder.sum( 5000, 2000 ) = 7000
adder.sum( "Math", "Man" ) = MathMan
MyAdder.sum( 5, 2 ) = 7
Adder.subtract( 5, 2 ) = 3
>
```

Class voyager1.0.0\examples\orb\Adder.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.orb;

import java.util.Vector;

public class Adder
  {
  public Adder()
    {
    System.out.println( "Construct Adder" );
    }

  public int sum( int x, int y )
    {
    return x + y;
    }

  public String sum( String s1, String s2 )
    {
    return s1 + s2;
    }

  public static int subtract( int x, int y )
    {
    return x - y;
    }
  }
```

Application voyager1.0.0\examples\orb\Dynamic1.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.orb;

import COM.objectspace.voyager.*;

public class Dynamic1
  {
  public static void main( String args[] )
    {
```

```
    try
      {
      VAdder adder = new VAdder( "localhost:8000/MyAdder" );

      // execute sum() using a virtual reference and a sync
      Sync sync1 = new Sync();
      sync1.setSignature( "sum(II)" );
      sync1.writeInt( 5000 );
      sync1.writeInt( 2000 );
      Result sync1Result = adder.send( sync1 );
      System.out.println( "adder.sum( 5000, 2000 ) = "+sync1Result.readInt()
);

      // execute sum() using a virtual reference and a future
      Future future1 = new Future();
      future1.setSignature( "sum(Ljava.lang.String;Ljava.lang.String;)" );
      future1.writeObject( "Math" );
      future1.writeObject( "Man" );
      Result futureResult = adder.send( future1 );
      Object result = futureResult.readObject();
      System.out.println( "adder.sum( \"Math\", \"Man\" ) = " + result );

      // execute sum() using a static method in VObject and a sync
      Sync sync2 = new Sync();
      sync2.setSignature( "sum(II)" );
      sync2.writeInt( 5 );
      sync2.writeInt( 2 );
      Result sync2Result =
        VObject.send( sync2, "localhost:8000/MyAdder", 1000 );
      System.out.println( "MyAdder.sum( 5, 2 ) = " + sync2Result.readInt() );

      // execute substract() using a static method in VObject and a sync
      Sync sync3 = new Sync();
      sync3.setSignature( "subtract(II)" );
      sync3.writeInt( 5 );
      sync3.writeInt( 2 );
      Result sync3Result =
        VObject.send( "examples.orb.Adder", sync3, "localhost:8000", 10000 );
     System.out.println( "Adder.subtract( 5, 2 ) = "+sync3Result.readInt() );

      Voyager.shutdown();
      }
    catch( VoyagerException exception )
      {
      System.err.println( exception );
      }
    }
  }
```

The `TestScheduler.java` example program demonstrates the simplicity of creating a powerful utility using the dynamic invocation feature.

The `Scheduler` class allows you to request that a named message be sent to an object at a specified time. The main program, `TestScheduler.java`, creates an agent and asks it to

move to a remote program. The program then tells `Scheduler` to send the `remote()` message to the agent after 10 seconds elapse. When the agent arrives at the remote program, the program tells `Scheduler` to send the `local()` message to the agent after five seconds elapse, and then the program allows its thread to terminate.

Before `local()` is sent to the agent, the agent has no allocated thread. Therefore, the technique described above is useful when an agent needs to become inactive, give up its thread, and become active again at a particular time. Mobility and agents are described in more detail in Chapter 8, “Mobility,” and Chapter 9, “Agents.”

Note: This implementation of `Scheduler` is rather unsophisticated. A commercial-quality version of `TestScheduler.java` would maintain an ordered list of entries. Instead of awaking every second to check the entire list, the agent would wait until the next entry to check.

Use the following commands from the `\voyager1.0.0\examples\orb` directory to prepare and compile the `TestScheduler.java` program:

```
vcc TestAgent
javac Scheduler.java TestAgent.java VTestAgent.java TestScheduler.java
```

Start a server on port 8000 in one window, and then run `TestScheduler.class` in a second window.

Window 1

```
>voyager 8000
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:8000
arrived
send local() to me after 5s
my thread terminates but I do not die...
local!
remote!
now I die...
```

Window 2

```
>java examples.orb.TestScheduler
construct local TestAgent
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1183
move to localhost:8000
send remote() to remote agent after 10s
>
```

Class voyager1.0.0\examples\orb\Scheduler.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.orb;

import java.util.Date;
import java.util.Vector;
import COM.objectspace.voyager.*;

public class Scheduler implements Runnable
  {
  static Scheduler scheduler = new Scheduler();
  private Vector entries = new Vector();

  public static void invokeAfter(int seconds, Object object, String callback)
    throws VoyagerException
    {
    scheduler.addEntry( Voyager.getTime() + seconds, object, callback );
    }

  public static void invokeAt( Date date, Object object, String callback )
    throws VoyagerException
    {
    scheduler.addEntry( (int) ( date.getTime() / 1000 ), object, callback );
    }

  private Scheduler()
    {
    Thread thread = new Thread( this );
    thread.setDaemon( true );
    thread.start();
    }

  public void run()
    {
    while( true ) // loop every second
      {
      try
        {
        checkEntries();
        }
      catch( VoyagerException exception )
        {
        System.out.println( exception );
        }

      try{ Thread.sleep( 1000 ); } catch( InterruptedException exception ) {}
      }
    }

  private void addEntry( int time, Object object, String callback )
    throws VoyagerException
    {
    Entry entry = new Entry();
```

```
    entry.time = time;
    entry.callback = callback;

    // obtain a remote reference to the specified object
    if( object instanceof VObject )
      entry.vobject = (VObject) object; // already a remote reference
    else
      entry.vobject = VObject.forObject( object );

    entries.addElement( entry );
    }

  private void checkEntries() throws VoyagerException
    {
    int index = 0;

    while( index < entries.size() )
      {
      Entry entry = (Entry) entries.elementAt( index );

      // if the time has come, send the callback
      if( Voyager.getTime() >= entry.time )
        {
        OneWay oneway = new OneWay(); // create smart oneway messenger
        oneway.setSignature( entry.callback + "()" ); // set signature
        entry.vobject.send( oneway ); // send
        entries.removeElementAt( index ); // remove from list
        }
      else
        {
        ++index;
        }
      }
    }
  }

class Entry
  {
  int time; // when the callback should be sent
  VObject vobject; // remote reference to the callback target
  String callback; // the name of the callback
  }
```

Class voyager1.0.0\examples\orb\TestAgent.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.orb;

import COM.objectspace.voyager.*;

public class TestAgent extends Agent
  {
  public void go( String address ) throws VoyagerException
    {
    System.out.println( "move to " + address );
    moveTo( address, "arrived" );
    }

  public void arrived() throws VoyagerException
    {
    System.out.println( "arrived" );
    System.out.println( "send local() to me after 5s" );
    Scheduler.invokeAfter( 5, this, "local" ); // local callback
    System.out.println( "my thread terminates but I do not die..." );
    }

  public void local()
    {
    System.out.println( "local!" );
    }

  public void remote() throws VoyagerException
    {
    System.out.println( "remote!" );
    System.out.println( "now I die..." );
    dieNow();
    }
  }
```

Application voyager1.0.0\examples\orb\TestScheduler.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.orb;

import COM.objectspace.voyager.*;

public class TestScheduler
  {
  public static void main( String args[] )
    {
    try
      {
      System.out.println( "construct local TestAgent" );
      VTestAgent agent = new VTestAgent( "localhost" );
      agent.go( "localhost:8000" );
      System.out.println( "send remote() to remote agent after 10s" );
      Scheduler.invokeAfter( 10, agent, "remote" ); // remote callback
      }
    catch( VoyagerException exception )
      {
      System.out.println( exception );
      }
    }
  }
```

Virtual References to Remote Results

By default, `Future` and `Sync` messengers return copies of method results. If a result is large, undesirable network traffic can occur. Voyager allows a messenger to return a virtual reference to a result, thereby greatly reducing the amount of network traffic. If the result is not serializable, returning a virtual reference obviates the need for serialization, thus allows the method to be invoked successfully. The life span of the remote result is set with `dieWhenNoReferences()`, so, until destroyed, the virtual reference to the result keeps the remote result alive.

To request that a messenger return a virtual reference to a result, use `setVirtual()`. If the remote program has not used the virtual class before, the program attempts to load the class from the local program. Ensure that the local program has dynamically linked the virtual class before attempting to retrieve a virtual reference of that class type.

Use the following commands from the `\voyager1.0.0\examples\orb` directory to prepare and compile the `Reference1.java` example program:

```
vcc java.util.Vector
vcc MultiplicationTable
javac MultiplicationTable.java VMultiplicationTable.java Reference1.java
      VVector.java
```

Start a server on port 8000 in one window, and then run `Reference1.class` in a second window.

Window 1

```
>voyager 8000
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:8000
return a vector of size 10000
return a vector of size 10000
```

Window 2

```
>java examples.orb.Reference1
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1185
copy took 3190ms
last element = 1000000
virtual reference took 660ms
last element = 1000000
>
```

Class voyager1.0.0\examples\orb\MultiplicationTable.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.orb;

import java.util.Vector;

public class MultiplicationTable
  {
  public Vector getTable( int n )
    {
    Vector table = new Vector();

    for( int i = 1; i <= n; i++ )
      table.addElement( new Integer( i * i ) );

    System.out.println( "return a vector of size " + n );
    return table;
    }
  }
```

Application voyager1.0.0\examples\orb\Reference1.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.orb;

import COM.objectspace.voyager.*;
import java.util.Vector;
import java.util.Date;
import VVector;

public class Reference1
  {
  public static void main( String args[] )
    {
    try
      {
      // Create remote multiplication table.
      VMultiplicationTable table = new
VMultiplicationTable("localhost:8000");

      // Obtain a copy of the result vector and display last value.
      long start = (new Date()).getTime();
      table.setMessageTimeout( 300000 ); // allow up to 300 seconds
      Vector localTable = table.getTable( 10000 );
      long stop = (new Date()).getTime();
      System.out.println( "copy took " + (stop - start) + "ms" );
      System.out.println( "last element = " + localTable.elementAt( 999 ) );

      // Since the remote program hasn't seen examples.orb.VVector, it will
      // ask this program for the class. The next line of code makes sure
      // that this program has dynamically linked examples.orb.VVector prior
      // to receiving this request. If examples.orb.VVector had already been
      // used before this point in the program, the next line of code would
      // have been unnecessary.
      VVector dummy = new VVector(); // create null virtual reference

      // Obtain a virtual reference to the result vector
      // and display last value.
      Sync sync = new Sync();
      sync.setVirtual( true ); // return a virtual reference
      start = (new Date()).getTime();
      Result result = table.getTable( 10000, sync );
      VVector remoteTable = (VVector) result.readObject();
      stop = (new Date()).getTime();
     System.out.println( "virtual reference took " + (stop - start) + "ms" );
      System.out.println( "last element = " + remoteTable.elementAt( 999 ) );

      Voyager.shutdown();
      }
    catch( VoyagerException exception )
      {
      System.err.println( exception );
      }
    }
  }
```

7

Events, Listeners, and Assistants

ObjectSpace Voyager™ Core Technology (Voyager) follows the standard JavaBeans™ event model for event and listener syntax and semantics. Voyager's event reporting subsystem allows for general notification of most system- and object-level incidents. These events can be extremely useful to developers. Following are example applications of the event mechanism:

- Monitoring of system throughput by watching all message traffic
- Intelligent analysis of and corrective action for system-level exceptions that have not been handled
- Creation of object and agent audit trails
- Monitoring of agent movement patterns for security, performance optimization, and tracking purposes
- Use of *assistants*, listeners that are persisted and moved as the objects they are attached to are persisted and moved

Three event classes (`SystemEvent`, `ObjectEvent`, and `SubspaceEvent`) and two listener classes (`SystemListener` and `ObjectListener`) allow developers to add these and many other features to their systems.

Listening to an Object

Every object is a potential source of events. Just as `VObject` transparently adds methods to remote-enabled objects for movement, persistence, and life spans, `VObject` also adds methods for adding and removing `ObjectListener` objects. If an object adds itself as an `ObjectListener` to another object, then that listener receives each event that the object generates. If an object has no listeners, then it does not generate any events.

An `ObjectListener` can get notification of the object events for all objects in a Voyager program by adding itself as an `ObjectListener` to Voyager via the static method `Voyager.addObjectListener()`. If Voyager has no object listeners, the system generates no object events.

Note: To prevent an infinite loop, Voyager does not allow addition of a virtual reference as an `ObjectListener`.

Object Events

Object events are events associated with a particular object. Each object event, or instance of the `ObjectEvent` class, has the following.

- A source that is retrievable by the `getSource()` method. This source is the object that generated the event. The source field in `java.util.EventObject` is declared transient; therefore, if an `ObjectEvent` is sent between virtual machines, the source information is lost.
- A virtual reference to the source that is retrievable by the `getReference()` method. This virtual reference is stored in a nontransient field to enable remote listening. That is, a listener can use this reference when it is necessary to interact with the remote source.
- A code that identifies the type of event and is retrievable by the `getCode()` method. These codes are static members of the `ObjectEvent` class.
- Three arguments that are retrievable by the `getArg1()`, `getArg2()`, and `getArg3()` methods.

All Voyager object event codes and associated arguments are described below.

`SAVING`	Generated before the source is saved to persistent store. All three arguments are null.
`LOADED`	Generated after the source is loaded into memory from persistent store. All three arguments are null.
`FLUSHING`	Generated before the source is flushed from memory and into persistent store. All three arguments are null.
`REMOVING`	Generated before the source is removed from persistent store. All three arguments are null.
`DYING`	Generated before the source is deregistered from the Voyager program. All three arguments are null.
`MOVING`	Generated before the source moves. *Arg1* is a `String` object that contains the address to which the source is moving. *Arg2* and *Arg3* are null.

`DROPPING_FORWARDER`	Generated before the source drops a forwarder. *Arg1* is a `String` object that contains the address to which the forwarder sends messages. *Arg2* and *Arg3* are null.
`MOVE_FAILED`	Generated after the source tries to move and fails. *Arg1* is the thrown exception that causes the move to fail. *Arg2* and *Arg3* are null.
`ARRIVED`	Generated after the source arrives in a Voyager program. *Arg1* is a `String` object that contains the Voyager program address the source came from. *Arg2* is a `String` object that contains the Voyager program address the source moved to. *Arg3* is null.
`BEGIN_MESSAGE`	Generated before a message is invoked on the source. *Arg1* is the messenger delivering the message. *Arg2* and *Arg3* are null.
`END_MESSAGE`	Generated after a message is invoked on the source. *Arg1* is the messenger delivering the message. *Arg2* and *Arg3* are null.
`BEGIN_ENCOUNTER`	Generated when an object moves to the source, before the callback is invoked. *Arg1* is a virtual reference to the source and *Arg2* is a virtual reference to the moved object. Sometimes these virtual references are null; for instance, they might be null if the virtual classes have not yet been loaded from the network or if the underlying objects have not been processed with `vcc`. *Arg3* is null.
`END_ENCOUNTER`	Generated when an object moves to the source, after the callback is invoked. *Arg1* is a virtual reference to the source and *Arg2* is a virtual reference to the moved object. Sometimes these virtual references are null; for instance, they might be null if the virtual classes have not yet been loaded from the network or if the underlying objects have not been processed with `vcc`. *Arg3* is null.
`UNHANDLED_OBJECT_EXCEPTION`	Generated after the source throws an exception that is not caught by the program. *Arg1* is the thrown exception. *Arg2* and *Arg3* are null. **Note:** This event merely alerts developers that an exception was thrown; the Voyager run time catches the exception.
`HEARTBEAT`	Generated after the source receives a heartbeat pulse. All three arguments are null.

Listening to the System

The system is also a potential source of events. A `SystemEvent` is generated when Voyager has at least one `SystemListener` and a system-level incident of interest occurs.

System Events

System events are events associated with the system as a whole, not particular objects. Each system event, or instance of the `SystemEvent` class, has the following.

- A source that is retrievable by the `getSource()` method. This source is the address `String` of the Voyager program that generated the event. The source field in `java.util.EventSystem` is declared transient; therefore, if a `SystemEvent` is sent between virtual machines, the source address is lost.
- A string to the source that is retrievable by the `getAddress()` method. This string is stored in a nontransient field to enable remote listening. That is, a listener can retrieve this string when it is necessary to determine the remote Voyager program.
- A code that identifies the type of event and is retrievable by the `getCode()` method. These codes are static members of the `SystemEvent` class.
- Three arguments that are retrievable by the `getArg1()`, `getArg2()`, and `getArg3()` methods.

All Voyager system event codes and associated arguments are described below.

`PULSE`	Generated before the system instructs each reference in the virtual machine to pulse a heartbeat to its remote object if necessary. *Arg1* is an `Integer` object that contains the number of virtual references held by objects in the virtual machine. *Arg2* and *Arg3* are null.
`CLEANUP`	Generated before the system checks both the virtual machine and its persistent store for objects with expired life spans. *Arg1* is an `Integer` object that contains the number of virtual objects in the virtual machine. *Arg2* and *Arg3* are null.
`UNHANDLED_SYSTEM_EXCEPTION`	Generated after a system-level exception occurs that is not caught by the program. *Arg1* is the thrown exception. *Arg2* and *Arg3* are null. **Note:** This event merely alerts developers that an exception was thrown; the Voyager run time catches the exception.

`LOAD_CLASS_INTO_FILESYSTEM`	Generated before Voyager copies a class into the file system. *Arg1* is a `String` object that contains the fully qualified name of the class being loaded. *Arg2* is the complete file name in which the class is being saved. *Arg3* is null.
`LOAD_CLOSURE_FROM_REMOTE`	Generated before Voyager loads the closure of a class from a remote virtual machine. *Arg1* is a `String` object that contains the fully qualified name of the class whose closure is loaded from the remote program. *Arg2* is a `String` object that contains the Voyager program address in which the closure is loaded. *Arg3* is null.
`LOAD_CLASSES_FROM_REMOTE`	Generated before Voyager loads a set of classes (the classes that must be moved across the network as a result of loading the closure for a class) from a remote virtual machine. *Arg1* is a `Vector` object that contains `String` objects, which, in turn, each contain the fully qualified name of one of the classes loaded from the remote program. *Arg2* is a `String` object that contains the remote program address in which the classes are loaded. *Arg3* is null.
`LOAD_CLOSURE_FROM_CODEBASE`	Generated before Voyager loads the closure of a class from codebase. *Arg1* is a `String` object that contains the fully qualified name of the class whose closure is loaded from codebase. *Arg2* is a `String` object that contains the information that tells the Voyager program what the codebase is. *Arg3* is null.
`LOAD_CLOSURE_FOR_APPLET`	Generated before Voyager loads the closure of a class for an applet. *Arg1* is a `String` object that contains the fully qualified name of the applet class whose closure is loaded. *Arg2* is a `String` object that contains the information that tells the Voyager program what the codebase is. *Arg3* is null.
`SENDING`	Generated before Voyager sends a messenger or reply to another Voyager program. *Arg1* is a `Runnable` object that is the sent messenger or reply. *Arg2* is a `String` object that contains the program address to which the messenger or reply is sent. *Arg3* is the total number of bytes sent.

`RECEIVED`	Generated after Voyager receives a messenger or reply from another Voyager program. *Arg1* is a `Runnable` object that is the received messenger or reply. *Arg2* is a `String` object that contains the program address from which the messenger or reply is received. *Arg3* is the total number of bytes received.
`FORWARDED`	Generated after a messenger is forwarded to a Voyager program. *Arg1* is the forwarded messenger, which is configured with the new destination. *Arg2* and *Arg3* are null.
`SERVER_SOCKET_EXCEPTION`	Generated when the creation of a server socket results in an exception. *Arg1* is the thrown exception. *Arg2* and *Arg3* are null. **Note:** This event merely alerts developers that an exception was thrown; the Voyager run time catches the exception.
`SOCKET_EXCEPTION`	Generated when the creation or use of a nonserver socket results in an exception. *Arg1* is the thrown exception. *Arg2* and *Arg3* are null. **Note:** This event merely alerts developers that an exception was thrown; the Voyager run time catches the exception.
`VOYAGER_SHUTDOWN`	Generated before Voyager shuts down. All three arguments are null.

Listening to a Subspace

A subspace is another potential source of events. A `SubspaceEvent` is generated when a subspace has at least one `ObjectListener` and a subspace-level incident of interest occurs.

Subspace Events

The `SubspaceEvent` object extends `ObjectEvent` and thus inherits the `getCode()`, `getSource()`, `getReference()`, and the three `getArg()` methods.

All Voyager subspace event codes and associated arguments are described below.

`ADD`	Generated after a virtual reference to an object is added to a subspace. *Arg1* is the added reference. *Arg2* and *Arg3* are null.
`REMOVE`	Generated after a virtual reference to an object is removed from a subspace. *Arg1* is the removed reference. *Arg2* and *Arg3* are null. **Note:** No event is generated if the removal occurs because of purging.
`CONNECT`	Generated after a neighbor is connected to a subspace. *Arg1* is a virtual reference to the connected neighbor. *Arg2* and *Arg3* are null.
`DISCONNECT`	Generated after a neighbor is disconnected from a subspace. *Arg1* is a virtual reference to the disconnected neighbor. *Arg2* and *Arg3* are null. **Note:** No event is generated if the disconnect occurs because of purging.

To listen for subspace events, implement the `ObjectListener` interface and add the listening object as a listener or as an assistant to either the subspace or the Voyager server. Because the subspace event is received in the `objectEvent()` method, the event is typed as an `ObjectEvent`. Because an object event is received by the listener, the `instanceof` operator should be used to identify those object events that are subspace events. For example:

```
public void objectEvent( ObjectEvent event )
  {
  boolean isSubspaceEvent = event instanceof SubspaceEvent;
  if( isSubspaceEvent && ( event.getCode() == SubspaceEvent.ADD )
    System.out.println( "A reference was added to the subspace." );
  }
```

System Tracing

The `Listener1.java` example program demonstrates how to write a trace facility that dumps a log of all significant activity to `System.out`. The program uses a `SystemSpy` object, an `ObjectSpy` object, and the familiar `VVector` object.

From the `\voyager1.0.0\examples\listeners` directory, use the following commands to prepare and compile `Listener1.java`:

```
vcc java.util.Vector
javac VVector.java SystemSpy.java ObjectSpy.java Listener1.java
```

Now run `Listener1.class`.

```
>java examples.listeners.Listener1
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1220
event = ObjectEvent( end message Sync(
addObjectListener(LCOM.objectspace.voyager.ObjectListener;)V,
208.6.239.200:1220/1217097227 ->
208.6.239.200:1220/215-66-102-157-73-100-196-41-188-63-72-134-111-72-132-6
8 )
[208.6.239.200:1220/215-66-102-157-73-100-196-41-188-63-72-134-111-72-132-
68] )
event = ObjectEvent( begin message Sync( addElement(Ljava.lang.Object;)V,
208.6.239.200:1220/-387061123 ->
208.6.239.200:1220/215-66-102-157-73-100-196-41-188-63-72-134-111-72-132-6
8 )
[208.6.239.200:1220/215-66-102-157-73-100-196-41-188-63-72-134-111-72-132-
68] )
.
.
.
vector = [hello, goodbye]
event = SystemEvent( pulse 5 references [208.6.239.200:1220] )
event = SystemEvent( pulse 5 references [208.6.239.200:1220] )
event = SystemEvent( pulse 5 references [208.6.239.200:1220] )
event = SystemEvent( cleanup 6 objects [208.6.239.200:1220] )
event = SystemEvent( pulse 3 references [208.6.239.200:1220] )
>
```

Class voyager1.0.0\examples\listeners\SystemSpy.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.listeners;

import COM.objectspace.voyager.*;

public class SystemSpy implements SystemListener
  {
  public void systemEvent( SystemEvent event )
    {
    //print out all system events
    System.out.println( "event = " + event );
    }
  }
```

Class voyager1.0.0\examples\listeners\ObjectSpy.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.listeners;

import COM.objectspace.voyager.*;

public class ObjectSpy implements ObjectListener, java.io.Serializable
  {
  public void objectEvent( ObjectEvent event )
    {
    //print out all object events
    System.out.println( "event = " + event );
    }
  }
```

Application voyager1.0.0\examples\listeners\Listener1.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.listeners;

import COM.objectspace.voyager.*;
import VVector;

public class Listener1
  {
  public static void main( String[] args )
    {
    try
      {
      //create a system listener that just prints out events
      SystemSpy systemSpy = new SystemSpy();
      Voyager.addSystemListener( systemSpy );

      //create an object and add an object listener
      //that just prints out events
      VVector vector = new VVector( "localhost" );
      ObjectSpy objectSpy = new ObjectSpy();
      vector.addObjectListener( objectSpy );
      vector.addElement( "hello" );
      System.out.println( "vector = " + vector );

      //turn off object listening
      vector.removeObjectListener( objectSpy );
      vector.addElement( "goodbye" );
      System.out.println( "vector = " + vector );

     try{ Thread.sleep( 20000 ); } catch( InterruptedException exception ) {}
      Voyager.shutdown();
      }
    catch( VoyagerException exception )
      {
      System.err.println( exception );
      }
    }
  }
```

System Tracing with the Monitor

Because logging system activity to a console is such a common debugging technique, Voyager includes a `Monitor` object that implements `ObjectListener` and `SystemListener` and dumps most significant activity to `System.out`, much as the spy objects in the `Listener1.java` example do. Enable monitoring for a Voyager program by using the `–t` option from the command line:

```
voyager 8000 –t
```

Three options for the `–t` switch are `o`, `c`, and `h`.

- `o` Instructs Voyager to trace object events.
- `c` Instructs Voyager to trace communication system events, that is, `SystemEvent.SENDING` and `SystemEvent.RECEIVING`.
- `h` Instructs Voyager to trace housekeeping system events, that is all system events except `SystemEvent.SENDING` and `SystemEvent.RECEIVING`.

For example, the following line of code instructs Voyager to log object and housekeeping system events only.

```
voyager 8000 –toh
```

By default, all events are monitored; hence, `-t` is equivalent to `–tohc`.

To turn on monitoring in your programs, create a new monitor object and add it as a system listener to Voyager, as an object listener to Voyager, or both.

```
Monitor monitor = new Monitor();
Voyager.addSystemListener( monitor );
Voyager.addObjectListener( monitor );
```

As a system listener, the monitor can print all generated system events. As an object listener, the monitor prints all object events generated by all objects, excluding the `ObjectEvent.END_MESSAGE` event. (Because the `ObjectEvent.BEGIN_MESSAGE` event is printed, printing the `ObjectEvent.END_MESSAGE` event is unnecessary and would be distracting.)

The `Monitor` class has two methods useful to programmers: `public void monitorCommunications( boolean flag )` and `public void monitorHousekeeping( boolean flag )`. These methods are equivalent to the `–t` options `c` and `h`. If `flag` is true for either of these methods, then associated events are printed to `System.out`. If `flag` is false, printing of associated events is suppressed. By default, all system events are printed.

When monitoring is enabled, you might receive the `[virtual class not available]` message. You might get this message if an argument is intended to be a virtual reference to an object whose class has not been processed with `vcc`. You also might receive this message when events are generated as a result of an operation that depends on class loading to get the associated virtual classes. Because the class loading has not yet occurred, the virtual classes are not yet available. In both cases, the message is common and no cause for concern.

Use the following command from the `voyager1.0.0\examples\listeners` directory to compile the `Listener2.java` example program:

```
javac Listener2.java
```

Note: This command works only if you have already prepared and compiled the `VVector.java` example program, as shown in “System Tracing” on page 135.

Now run `Listener2.class`.

```
>java examples.listeners.Listener2
voyager(tm) 1.0.0, copyright objectspace 1997

ObjectEvent( begin message Sync( setAlias(Ljava.lang.String;)V,
208.6.239.200:1221/556401104 ->
208.6.239.200:1221/76-28-192-102-44-35-103-33-9-127-72-134-111-72-132-69
)
[208.6.239.200:1221/76-28-192-102-44-35-103-33-9-127-72-134-111-72-132-69]
 )

ObjectEvent( begin message Sync( liveForever()V,
208.6.239.200:1221/-458271365 ->
208.6.239.200:1221/76-28-192-102-44-35-103-33-9-127-72-134-111-72-132-69
)
[208.6.239.200:1221/76-28-192-102-44-35-103-33-9-127-72-134-111-72-132-69]
 )
address = 208.6.239.200:1221
.
.
.
vector = [hello, goodbye]

SystemEvent( pulse 5 references [208.6.239.200:1221] )

SystemEvent( pulse 5 references [208.6.239.200:1221] )

SystemEvent( pulse 5 references [208.6.239.200:1221] )

SystemEvent( cleanup 6 objects [208.6.239.200:1221] )

SystemEvent( pulse 3 references [208.6.239.200:1221] )
>
```

Application voyager1.0.0\examples\listeners\Listener2.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.listeners;

import COM.objectspace.voyager.*;
import COM.objectspace.voyager.util.Monitor;
import VVector;

public class Listener2
  {
  public static void main( String[] args )
    {
    try
      {
      //create a monitor that monitors both system and object events
      Monitor monitor = new Monitor();
      Voyager.addObjectListener( monitor );
      Voyager.addSystemListener( monitor );

      //create a vector and send it messages to see what events are generated
      VVector vector = new VVector( "localhost" );
      vector.addElement( "hello" );
      vector.addElement( "goodbye" );
      System.out.println( "vector = " + vector );

     try{ Thread.sleep( 20000 ); } catch( InterruptedException exception ) {}
      Voyager.shutdown();
      }
    catch( VoyagerException exception )
      {
      System.err.println( exception );
      }
    }
  }
```

When using the monitor to listen to object events, be aware that many events showing internal Voyager mechanisms are printed. For instance, when an object moves to a remote program, Voyager sends messages back and forth as part of the move-handshake process. Voyager sends other messages during lookup and garbage collection, identifiable by two leading underscores in such method names, like `__activate()` and `__ack()`.

Assistants

Listener applications yield surprisingly powerful results. For instance, a developer can create an `ObjectListener` that listens for `ARRIVED` events. Because developers can use listeners to modify and enhance the behavior of objects in any way desired, the application possibilities are endless.

In the JavaBeans model, however, collections of listeners are transient variables. This limits the functionality of listeners in Voyager programs because Voyager objects often move from one program to another and they often persist themselves to disk storage. In both cases, transient variables are lost.†

For nontransient listening, Voyager has a special kind of listener, an *assistant*. When you attach an assistant to an object, the assistant is persisted when the object is persisted and the assistant moves with the object as the object moves.

You can add an assistant to an object in two ways, both using the `addAssistant()` method. Use the following code to add an assistant and specify a key for the assistant.

```
public void addAssistant( String key, ObjectListener listener )
```

The key is stored and can be used later to remove the assistant. Several assistants can share a key, so removing by key results in removing all assistants associated with the key.

If key uniqueness is necessary in a program, use the following code to add an assistant.

```
public String addAssistant( ObjectListener listener )
```

A globally unique key is returned. This key is used for the assistant.

The `Assistant1.java` example program demonstrates adding a `Replicator` listener as an assistant. A `Replicator` listens for `END_MESSAGE` object events and replicates the messages to another object. The replicator is first configured with a `MessageSelector` object that allows the replicator to filter messages for only those methods in which it is interested. For example, a replicator could be used to keep two or more objects in sync; the replicator could filter all object events and forward only messages changing the state of the object to the other objects in the replicated group.

Use the following command from the `voyager1.0.0\examples\listeners` directory to compile `Assistant1.java`:

```
javac MethodSelector.java Replicator.java Assistant1.java
```

Start a server on port 8000 in one window, and then run `Assistant1.class` in a second window.

† Read the Java Serialization specification for more information on transient variables.

Window 1

```
>voyager 8000
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:8000
```

Window 2

```
>java examples.listeners.Assistant1
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1074
original = [clubs, diamonds, spades, hearts]
mirror = [clubs, diamonds, spades, hearts]
original = [spades, hearts]
mirror = [spades, hearts]
>
```

Class voyager1.0.0\examples\listeners\MethodSelector.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.listeners;

import java.util.*;
import COM.objectspace.voyager.*;
import COM.objectspace.voyager.util.*;

/**
 * MethodSelector is a selector that selects an object if it
 * is a Messenger whose signature is in my signature set.
 */
public class MethodSelector implements Selector
  {
  private Vector signatures = new Vector();

  public MethodSelector()
    {
    //Construct myself with an empty set of signatures.
    }

  public void addSignature( String signature )
    {
    //Add the specified signature to my set of signatures.
    signatures.addElement( signature );
    }

  public void removeSignature( String signature )
    {
    //Remove the specified signature from my set of signatures.
    signatures.removeElement( signature );
    }

  public boolean select( Object object )
    {
    //Return true if the specified object is a
    //Messenger that matches my signature set.
    return object instanceof Messenger &&
      signatures.contains( ((Messenger) object).getSignature() );
    }
  }
```

Class voyager1.0.0\examples\listeners\Replicator.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.listeners;

import COM.objectspace.voyager.*;
import COM.objectspace.voyager.util.*;

public class Replicator implements java.io.Serializable, ObjectListener
  {
  //the object I am keeping in sync with the event source
  private VObject reference;

  //object that decides what messages to replicate
  private Selector selector;

  public Replicator( VObject reference )
    {
    this( reference, null );
    }

  public Replicator( VObject reference, Selector selector )
    {
    this.reference = reference;
    this.selector = selector;
    }

  public void objectEvent( ObjectEvent event )
    {
    //replicate after the message completes
    if( event.getCode() == ObjectEvent.END_MESSAGE )
      {
      Messenger messenger = (Messenger) event.getArg1();

      if( selector == null || selector.select( messenger ) )
        {
        try
          {
          //replicate the message
          Messenger replica = reference.newDefaultMessenger();
          replica.setSignature( messenger.getSignature() );
          replica.setArguments( messenger.getArguments() );
          reference.send( replica );
          }
        catch( Exception exception )
          {
          }
        }
      }
    }
  }
```

Application voyager1.0.0\examples\listeners\Assistant1.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.listeners;

import COM.objectspace.voyager.*;
import VVector;

public class Assistant1
  {
  public static void main( String[] args )
    {
    try
      {
      //create a local vector and a remote mirror of it
      VVector original = new VVector( "localhost" );
      VVector mirror = new VVector( "localhost:8000" );

      //create a method selector that will select only methods
      //that change the state of the vector
      MethodSelector selector = new MethodSelector();
      selector.addSignature( "addElement(Ljava.lang.Object;)V" );
      selector.addSignature( "removeElement(Ljava.lang.Object;)Z" );

      //the replicator assistant takes the mirror and the selector
      Replicator replicator = new Replicator( mirror, selector );
      //add the assistant to the original vector
      original.addAssistant( replicator );

      //add elements to the vector
      original.addElement( "clubs" );
      original.addElement( "diamonds" );
      original.addElement( "spades" );
      original.addElement( "hearts" );

      //verify the vectors have the same elements
      System.out.println( "original = " + original );
      System.out.println( "mirror = " + mirror );

      //remove elements from the vector
      original.removeElement( "diamonds" );
      original.removeElement( "clubs" );

      //verify the vectors have the same elements
      System.out.println( "original = " + original );
      System.out.println( "mirror = " + mirror );

      Voyager.shutdown();
      }
    catch( VoyagerException exception )
      {
      System.err.println( exception );
      }
    }
  }
```

The `Assistant2.java` example program demonstrates how an object can monitor itself. The program uses an instance of the `MyAgent` class. The agent moves from one virtual machine to another and receives each event it generates.

Use the following commands from the `voyager1.0.0\examples\listeners` directory to prepare and compile `MyAgent`:

```
vcc MyAgent
javac MyAgent.java VMyAgent.java Assistant2.java
```

Start a server on port 8000 in one window, and then run `Assistant2.class` in a second window.

Window 1

```
>voyager 8000
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:8000
MyAgent event = ObjectEvent( begin encounter target [virtual class not
available], requestor
208.6.239.200:8000/208-246-46-240-30-134-225-153-241-17-72-134-111-72-132-
72
[208.6.239.200:8000/208-246-46-240-30-134-225-153-241-17-72-134-111-72-132
-72] )
.
.
.
MyAgent event = ObjectEvent( end encounter target [virtual class not
available], requestor
208.6.239.200:8000/208-246-46-240-30-134-225-153-241-17-72-134-111-72-132-
72
[208.6.239.200:8000/208-246-46-240-30-134-225-153-241-17-72-134-111-72-132
-72] )
```

Window 2

```
>java examples.listeners.Assistant2
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1224
MyAgent event = ObjectEvent( source: examples.listeners.MyAgent(
231-212-71-239-
193-83-108-83-206-38-217-134-111-217-132-211 ), begin message Sync( go()V,
208.6
.239.89:1107/-372373224 ->
208.6.239.89:1107/231-212-71-239-193-83-108-83-206-38
-217-134-111-217-132-211 ) )
moving to 8000
.
.
.
MyAgent event = ObjectEvent( source: examples.listeners.MyAgent(
231-212-71-239-
193-83-108-83-206-38-217-134-111-217-132-211 ), end encounter target
[virtual cl
ass not available], requestor
208.6.239.89:8000/231-212-71-239-193-83-108-83-206
-38-217-134-111-217-132-211 )
```

Class voyager1.0.0\examples\listeners\MyAgent.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.listeners;

import COM.objectspace.voyager.*;

public class MyAgent extends Agent implements ObjectListener
  {
  String home;

  public MyAgent()
    {
    //listen for any event I generate
    addAssistant( this );
    }

  public void go() throws VoyagerException
    {
    //move around and see what events I generate
    home = Voyager.getAddress();
    System.out.println( "moving to 8000" );
    moveTo( "localhost:8000", "at8000" );
    }

  public void at8000() throws VoyagerException
    {
    //move back and see what events I generate
    System.out.println( "at8000, going home" );
    moveTo( home, "atHome" );
    }

  public void atHome()
    {
    System.out.println( "at home" );

    //I could execute clean-up code when I get the DYING event...
    dieNow();
    }

  public void objectEvent( ObjectEvent event )
    {
    //just print out all objects events
    System.out.println( "MyAgent event = " + event );
    }
  }
```

Application voyager1.0.0\examples\listeners\Assistant2.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.listeners;

import COM.objectspace.voyager.*;
import VVector;

public class Assistant2
  {
  public static void main( String[] args )
    {
    try
      {
      //create an agent that listens to itself
      VMyAgent agent = new VMyAgent( "localhost" );

      //move it around and see what gets printed out
      agent.go();
      }
    catch( VoyagerException exception )
      {
      System.err.println( exception );
      }
    }
  }
```

The `Assistant3A.java` and `Assistant3B.java` example programs demonstrate a powerful application of assistants. This two-phase example uses the `Autosaver` object. An autosaver is an assistant that, like the replicator, is configured to listen to `END_MESSAGE` object events. The autosaver uses a method selector to filter those messages in which it is interested and throws away the rest. When an `END_MESSAGE` event is generated for a message that interests the autosaver, the autosaver saves its associated object. In general, an autosaver, like a replicator, can be configured to listen for messages that modify the state of an object. After such a message is invoked, the object is saved to disk.

In the first phase of this example, `Assistant3A.java` creates a `Vector` object with an autosaver assistant and sends the vector messages that modify its state. In the second phase, the virtual machines are shut down and restarted. `Assistant3B.class` is then run, causing the vector to be loaded from disk. You can verify that the vector has been autopersisted after each change.

Phase 1

Use the following command from the `voyager1.0.0\examples\listeners` directory to compile the example programs:

```
javac Autosaver.java Assistant3A.java Assistant3B.java
```

Note: This command works only if you have already prepared and compiled the `VVector.java` and `MethodSelector.java` example programs, as shown in “System Tracing” on page 135 and in “Assistants” on page 141.

Start a server on port 8000 in one window. This server must be started with a database to verify that the autosaver saves the changes made. Choose any database name, and then run `Assistant3.class` in a second window.

Window 1

```
>voyager 8000 -d saver.db
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:8000
database = 0 objects, 0 classes
```

Window 2

```
>java examples.listeners.Assistant3A
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1102
vector = [spades, hearts]
>
```

Phase 1 of this example is now complete. Notice that Window 2 shuts down automatically, but Window 1 remains active.

Phase 2

Shut down the Voyager server by pressing Ctrl+C in Window 1. Then restart the database used in Phase 1 of this example in Window 1. Finally, run `Assistant3B.class` in Window 2. Verify that the vector created in `Assistant3A` was saved appropriately.

Window 1

```
>voyager 8000 -d saver.db
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:8000
database = 1 object, 0 classes
```

Window 2

```
>java examples.listeners.Assistant3A
voyager(tm) 1.0.0, copyright objectspace 1997
address = 127.0.0.1:1033, root = \
vector = [spades, hearts]
>java examples.listeners.Assistant3B
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1104
vector = [spades, hearts]
>
```

Class voyager1.0.0\examples\listeners\Autosaver.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.listeners;

import COM.objectspace.voyager.*;
import COM.objectspace.voyager.util.*;

public class Autosaver implements java.io.Serializable, ObjectListener
  {
  //object that decides what messages to initiate saving
  private Selector selector;

  public Autosaver()
    {
    }

  public Autosaver( Selector selector )
    {
    this.selector = selector;
    }

  public void objectEvent( ObjectEvent event )
    {
    if( event.getCode() == ObjectEvent.END_MESSAGE )
      {
      //save after the message completes
      Messenger messenger = (Messenger) event.getArg1();

      try
        {
        if( selector == null || selector.select( messenger ) )
          {
          //save the object that generated the event
          if( event.getReference() != null )
            event.getReference().saveNow();
          }
        }
      catch( VoyagerException exception )
        {
        VoyagerException.unhandledException( exception );
        }
      }
    }
  }
```

Application voyager1.0.0\examples\listeners\Assistant3A.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.listeners;

import COM.objectspace.voyager.*;
import VVector;

public class Assistant3A
  {
  public static void main( String[] args )
    {
    try
      {
      //create a vector at a name we can use to get to it later
      VVector vector = new VVector( "localhost:8000/MyVector" );
      vector.liveForever();

      //select messages that change the state of the vector
      MethodSelector selector = new MethodSelector();
      selector.addSignature( "addElement(Ljava.lang.Object;)V" );
      selector.addSignature( "removeElement(Ljava.lang.Object;)Z" );

      //create and add a listener that will save the vector each time
      //one of the above messages is sent to it
      Autosaver autosaver = new Autosaver( selector );
      vector.addAssistant( autosaver );

      //modify the vector
      vector.addElement( "clubs" );
      vector.addElement( "diamonds" );
      vector.addElement( "spades" );
      vector.addElement( "hearts" );
      vector.removeElement( "diamonds" );
      vector.removeElement( "clubs" );

      //print out the vector...
      //compare this to what is printed out in Assistant3B
      System.out.println( "vector = " + vector );

      Voyager.shutdown();
      }
    catch( VoyagerException exception )
      {
      System.err.println( exception );
      }
    }
  }
```

Application voyager1.0.0\examples\listeners\Assistant3B.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.listeners;

import COM.objectspace.voyager.*;
import VVector;

public class Assistant3B
  {
  public static void main( String[] args )
    {
    try
      {
      //load the vector from the previous example
      VVector vector =
        (VVector) VObject.forObjectAt( "localhost:8000/MyVector" );

      //print and verify all changes made to the vector were autosaved
      System.out.println( "vector = " + vector );
      Voyager.shutdown();
      }
    catch( VoyagerException exception )
      {
      System.err.println( exception );
      }
    }
  }
```

8

Mobility

Any object that has a virtual class and implements the `java.io.Serializable` interface can be moved, even while it is receiving remote messages. There are at least two reasons why object mobility is useful:

- Objects that exchange large numbers of messages can be moved closer to each other to reduce network traffic and increase throughput. A local message is often at least 1,000 times faster than its remote equivalent. This technique is known as *load balancing*.
- A program can move objects into a mobile device so that the program can remain with the device after the device has been disconnected from the network.

To move an object in ObjectSpace Voyager™ (Voyager) Core Technology, use `moveTo()` with the move destination as the first parameter. If you want the object to continue executing after the move completes, you can specify an optional callback function with an optional callback argument. The optional callback argument can be an object of any type.

There are two kinds of destination:

- **Program Destination**. If the destination is a program address, the object is moved from its current program to the destination program. If the move is successful and a callback function is specified, the public method whose name matches the callback function is executed on the object with the optional callback argument.
- **Object Destination**. If the destination is either the address of or a virtual reference to another object, the object is moved from its current program to the destination object's program. If the move is successful and a callback is specified, the public method whose name matches the callback argument is executed on the moved object. The callback contains a local Java™ reference to the destination object and the optional callback argument. The object that moved and the destination object cannot be moved by other objects until the callback completes.

When an object receives `moveTo()`, it first allows all remote messages currently in progress to complete. The object then enters a frozen state in which all incoming remote messages are suspended until the move is complete. A copy of the object is then moved to its destination.

If the move is successful, the original object leaves behind a special object called a *forwarder* that forwards messages from the old location to the new location. The original object is then destroyed, and the copy becomes the new "original." Remote messages suspended at the original location are resumed and forwarded to the new location. If a callback was specified,

the callback is executed using a new thread allocated from the Voyager thread pool. Note that `moveTo()` returns immediately when the move is complete and does not wait for the separate callback thread to finish executing.

If an error occurs during the move process, the copy is destroyed, the original object is unfrozen, remote messages suspended at the original location are resumed, and an exception is thrown to the caller of `moveTo()`.

An object's new location is automatically propagated to all its virtual references using an efficient update-on-demand technique. A forwarded remote message remembers that it has been forwarded. When a forwarded message arrives, the remote object's new location is either piggybacked onto the message return value (for synchronous and future messages) or sent back to the virtual reference by an explicit message (for one-way messages).

The life span and persistence of a forwarder is inherited from the original object. For example, if an object is programmed to die at a particular point in time, its forwarders also die at that time. Similarly, if an object is programmed to die when there are no remote references to it, its forwarders will also die when there are no remote references to them. A heartbeat sent via a forwarder keeps the forwarder alive and also causes an update of the virtual reference's knowledge of the new location. When an object's virtual references are updated with the object's new location, the object's forwarders no longer receive heartbeats and thus are garbage-collected.

When a moved object dies, it sends a one-way `dieNow()` message to each of its forwarders.

Performance Benefits of Mobility

Test programs were run to demonstrate the performance benefits of mobility. A client program used a server program to perform 5,000 calculations using three different modes of communication: sending remote messages via a virtual reference, sending local messages via a virtual reference, and sending local messages via a raw Java reference. The following table summarizes the tests result, with time shown in milliseconds.

Communications Mode	Time to Move	Time to Calculate	Total Time
Remote messages via virtual reference	n/a	28,141	28,141
Local messages via virtual reference	611	3,455	4,066
Local messages via raw Java reference	631	10	641

Use the following commands from the `\voyager1.0.0\examples\mobility` directory to prepare and compile the programs for this example.

```
vcc Client Server
javac Client.java VClient.java Server.java VServer.java
javac Remote.java Local.java Raw.java
```

Start a server on port 8000 in one window, and then run `Remote.class`, `Local.class`, and `Raw.class` in succession in a second window.

Note: Although you do not explicitly run `Client.class` and `Server.class` in this example, you remote-enable and compile `Client.java` and `Server.java` before compiling the main example programs. This is because the client and server programs are used within the other three programs.

Window 1

```
>voyager 8000
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:8000
construct Server( one )
construct Server( two )
time to move Client( two ) = 611ms
total time = 4066ms
construct Server( three )
time to move Client( three ) = 631ms
total time = 641ms
```

Window 2

```
>java examples.mobility.Remote
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1198
construct Client( one )
total time = 28141ms
>java examples.mobility.Local
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1200
construct Client( two )
>java examples.mobility.Raw
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1203
construct Client( three )
>
```

Class voyager1.0.0\examples\mobility\Client.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.mobility;

import java.util.Date;
import COM.objectspace.voyager.*;

public class Client implements java.io.Serializable
  {
  static final int LOOPS = 5000;
  String name;
  long created = (new Date()).getTime(); // creation time
  VServer vserver; // remote reference to server

  public Client( String name, VServer vserver )
    {
    this.name = name;
    this.vserver = vserver;
    System.out.println( "construct " + this );
    }

  public String toString()
    {
    return "Client( " + name + " )";
    }

  public void longCalculation() throws VoyagerException
    {
    for( int i = 0; i < 6; i++ )
      {
      System.out.println( this + " cube( " + i + " ) = " + vserver.cube( i ) );
      try{ Thread.sleep( 1000 ); } catch( InterruptedException exception ) {}
      }
```

```
  }

public void calculate()
  {
  try
    {
    for( int i = 0; i < LOOPS; i++ )
      vserver.square( i );

    long total = (new Date()).getTime() - created;
    System.out.println( "total time = " + total + "ms");
    }
  catch( VoyagerException exception )
    {
    System.err.println( exception );
    }
  }

public void atProgram()
  {
  long time = (new Date()).getTime() - created;
  System.out.println( "time to move " + this + " = " + time + "ms" );
  calculate();
  }

public void atServer( Server server )
  {
  long time = (new Date()).getTime() - created;
  System.out.println( "time to move " + this + " = " + time + "ms" );

  for( int i = 0; i < LOOPS; i++ )
    server.square( i );

  long total = (new Date()).getTime() - created;
  System.out.println( "total time = " + total + "ms" );
  }
}
```

Class voyager1.0.0\examples\mobility\Server.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.mobility;

public class Server implements java.io.Serializable
  {
  String name;

  public Server( String name )
    {
    this.name = name;
    System.out.println( "construct " + this );
    }

  public String toString()
    {
    return "Server( " + name + " )";
    }

  public int square( int x )
    {
    return x * x;
    }

  public int cube( int x )
    {
    System.out.println( this + " receives cube( " + x + " )" );
    return x * x * x;
    }

  public void atProgram()
    {
    System.out.println( this + " arrives at new program" );
    }

  public void atProgram( String message )
    {
    System.out.println( this + " " + message );
    }
  }
```

Application voyager1.0.0\examples\mobility\Remote.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.mobility;

import COM.objectspace.voyager.*;

public class Remote
  {
  public static void main( String args[] )
    {
    try
      {
      VServer server = new VServer( "one", "localhost:8000" );
      VClient client = new VClient( "one", server, "localhost" );

      // perform remote calculation
      client.calculate();
      Voyager.shutdown();
      }
    catch( VoyagerException exception )
      {
      System.out.println( exception );
      }
    }
  }
```

Application voyager1.0.0\examples\mobility\Local.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.mobility;

import COM.objectspace.voyager.*;

public class Local
  {
  public static void main( String args[] )
    {
    try
      {
      VServer server = new VServer( "two", "localhost:8000" );
      VClient client = new VClient( "two", server, "localhost" );

      // move, perform local calculation
      client.moveTo( server.getProgramAddress(), "atProgram" );
      Voyager.shutdown();
      }
    catch( VoyagerException exception )
      {
      System.out.println( exception );
      }
    }
  }
```

Application voyager1.0.0\examples\mobility\Raw.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.mobility;

import COM.objectspace.voyager.*;

public class Raw
  {
  public static void main( String args[] )
    {
    try
      {
      VServer server = new VServer( "three", "localhost:8000" );
      VClient client = new VClient( "three", server, "localhost" );

      // move to server, perform calculation using raw Java reference
      client.moveTo( server, "atServer" );
      Voyager.shutdown();
      }
    catch( VoyagerException exception )
      {
      System.out.println( exception );
      }
    }
  }
```

Invoking a Callback

When a callback is specified and a callback argument is provided, Voyager tries to invoke the most specific callback possible. This is consistent with Java signature mapping rules, but identifying the appropriate callback method is not always intuitive and is sometimes impossible within the scope of Java's mapping rules. For example, identifying an appropriate callback method is impossible when there is no method with a signature that matches the signature implied by the callback and callback argument. Sometimes the Java mapping rules cannot limit the set of possible methods to only one method. Both the nonexistent and ambiguous cases result in a `MethodNotFound` exception.

Use the following commands from the `\voyager1.0.0\examples\mobility` directory to compile the `Callback1.java` example program:

```
vcc Target
javac VTarget.java Target.java Callback1.java
```

Start a server on port 8000 in one window, and then run `Callback1.class` in a second window.

Window 1

```
>voyager 8000
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:8000
No arg version of callback() invoked
String version of callback() invoked with: callback argument
String version of callback() invoked with: callback argument
```

Window 2

```
>java examples.mobility.Callback1
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1206
COM.objectspace.voyager.MethodNotFoundException:
examples.mobility.Target.nonexi
stent_callback()
        thrown remotely from 208.6.239.89:8000
COM.objectspace.voyager.MethodNotFoundException: callback ambiguity
between [pub
lic void examples.mobility.Target.callback(java.lang.Object), public void
exampl
es.mobility.Target.callback(java.lang.Runnable), public void
examples.mobility.T
arget.callback(java.io.Serializable)]
        thrown remotely from 208.6.239.89:8000
>
```

Class voyager1.0.0\examples\mobility\Target.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.mobility;

import java.io.Serializable;

public class Target implements Runnable, Serializable
  {
  public Target()
    {
    }

  public void callback()
    {
    System.out.println( "No arg version of callback() invoked" );
    }

  public void callback( String string )
    {
    System.out.println(
      "String version of callback() invoked with: " + string );
    }

  public void callback( Object object )
    {
    System.out.println(
      "Object version of callback() invoked with: " + object );
    }

  //If callback() is invoked with a Target, both of the following methods
  //are suitable, so the Java Mapping rules cannot narrow down the set of
  //possible methods to a unique choice.  This results in ambiguity.
  public void callback( Runnable runnable )
    {
    System.out.println(
      "Runnable version of callback() invoked with: " + runnable );
    }

  public void callback( Serializable serializable )
    {
    System.out.println(
      "Serializable version of callback() invoked with: " + serializable );
    }


  //needed to implement Runnable
  public void run()
    {
    }
  }
```

Application voyager1.0.0\examples\mobility\Callback1.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.mobility;

import COM.objectspace.voyager.*;

public class Callback1
  {
  public static void main( String args[] )
    {
    try
      {
      VTarget object = new VTarget( "localhost" );

      //move, invoke callback with no argument
      object.moveTo( "localhost:8000", "callback" );

      // move, invoke callback with a String argument
      object.moveTo( "localhost", "callback", "callback argument" );

      //move, this seems like it will call the object version, but Java
      //mapping rules dictate that the string version be called
      object.moveTo("localhost:8000","callback",(Object)"callback argument");

      try
        {
        //move, invoke nonexistent callback, should throw exception
        object.moveTo( "localhost", "nonexistent_callback" );
        //since the move failed, the object is still remote
        }
      catch( VoyagerException exception )
        {
        System.out.println( exception );
        }

      try
        {
        //move, since Target implements both Runnable and Serializable,
        //this callback is ambiguous
        object.moveTo( "localhost:8000", "callback", new Target() );
        //since the move failed, the object is still remote

       //Uncomment the following code and try to compile. This verifies that
        //Reflectionis hiding the fact that invocation of callback() with a
        //Target object is considered ambiguous.
        /*
        Target one = new Target();
        Target two = new Target();
        one.callback( two );
        */
        }
      catch( VoyagerException exception )
        {
```

```
      System.out.println( exception );
      }

    Voyager.shutdown();
    }
  catch( VoyagerException exception )
    {
    System.out.println( exception );
    }
  }
}
```

Loading a Class

When an object is moved to a different program, classes that the object directly or indirectly references from its source code are immediately loaded into the destination program, if not already there.

This class loading feature is designed to work well under intermittent network connectivity. If Voyager were designed to load remote classes only when they are first used by an object, a network interruption could cause problems. For example, an object could be moved into a Java telephone, become disconnected from the network, and then be asked to perform a task requiring remote class loading from the original program.

One consequence of Voyager's load-on-move design is that objects can become quite heavyweight if you are not careful. If an object that references 20 large classes is moved around a network, a potentially large amount of class loading could occur if some of the programs do not already contain those classes.

An easy way to accidentally incur significant class loading is to move an object to another object and declare a callback function in the moving object's class that takes an instance of the destination object's class as its parameter. This coupling between classes means that the class of the destination object is loaded into every program the moving object visits. The client/server example program on page 156 has this problem. The `Client` class is coupled to the `Server` class because the callback function `atServer()` takes a `Server` as its argument.

Java interfaces allow you to avoid this kind of coupling. Create an interface for the destination object's class and define the callback to accept any argument that implements that interface. The moving object is then coupled to the lightweight interface rather than to the heavyweight implementation. In the program on page 156, you could define a `Server` interface that is implemented by the `ServerImpl` class. Then, the callback function could be defined to take a `Server` as an argument rather than a `ServerImpl`. The librarian example on page 183 of Chapter 9, "Agents," uses this approach.

Moving an Active Object

It is safe to move an object even when it is actively receiving remote messages. When a remote message arrives at an object, the message prevents the object from moving until its thread has finished executing. Each move operation on that object blocks until all the object's remote messages have completed.

It is unsafe to move an object if local Java references point to it from outside the context of Voyager or if the object has one or more threads not associated with a remote message.

The `LoadBalancer.java` example program shows how Voyager's mobility features allow creation of a simple load balancer. `LoadBalancer.java` uses the variation of `moveTo()` that allows an additional callback argument to be specified. A server object is moved between programs as it receives requests from two different client objects.

Use the following command from the `\voyager1.0.0\examples\mobility` directory to compile `LoadBalancer.java`:

```
javac LoadBalancer.java
```

Start a server on port 8000 in one window, and then run `LoadBalancer.class` in a second window.

Window 1

```
>voyager 8000
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:8000
construct Server( one )
construct Client( two )
Server( one ) receives cube( 0 )
Server( one ) receives cube( 0 )
Client( two ) cube( 0 ) = 0
Server( one ) receives cube( 1 )
Server( one ) receives cube( 1 )
Client( two ) cube( 1 ) = 1
Client( two ) cube( 2 ) = 8
Client( two ) cube( 3 ) = 27
Server( one ) moved again
Server( one ) receives cube( 4 )
Client( two ) cube( 4 ) = 64
Server( one ) receives cube( 4 )
Server( one ) receives cube( 5 )
Client( two ) cube( 5 ) = 125
Server( one ) receives cube( 5 )
```

Window 2

```
>java examples.mobility.LoadBalancer
construct Client( one )
Client( one ) cube( 0 ) = 0
Client( one ) cube( 1 ) = 1
Server( one ) moved once
Server( one ) receives cube( 2 )
Server( one ) receives cube( 2 )
Client( one ) cube( 2 ) = 8
Server( one ) receives cube( 3 )
Server( one ) receives cube( 3 )
Client( one ) cube( 3 ) = 27
Server( one ) receives cube( 4 )
Client( one ) cube( 4 ) = 64
Server( one ) receives cube( 4 )
Client( one ) cube( 5 ) = 125
From the Voyager 8000 Server
construct Server( one )
construct Client( two )
Server( one ) receives cube( 0 )
Server( one ) receives cube( 0 )
Client( two ) cube( 0 ) = 0
Server( one ) receives cube( 1 )
Server( one ) receives cube( 1 )
Client( two ) cube( 1 ) = 1
Client( two ) cube( 2 ) = 8
Client( two ) cube( 3 ) = 27
Client( two ) cube( 4 ) = 64
Server( one ) moved again
Server( one ) receives cube( 5 )
Server( one ) receives cube( 5 )
Client( two ) cube( 5 ) = 125
>
```

Application voyager1.0.0\examples\mobility\LoadBalancer.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.mobility;

import COM.objectspace.voyager.*;

public class LoadBalancer
  {
  public static void main( String args[] )
    {
    try
      {
      VServer server = new VServer( "one", "localhost:8000" );
      VClient client1 = new VClient( "one", server, "localhost" );
      VClient client2 = new VClient( "two", server, "localhost:8000" );

      // spawn two threads, one for each client
      client1.longCalculation( new OneWay() );
      client2.longCalculation( new OneWay() );

      // move the server every two seconds
      try{ Thread.sleep( 2000 ); } catch( InterruptedException exception ) {}
      server.moveTo( Voyager.getAddress(), "atProgram", "moved once" );
      try{ Thread.sleep( 2000 ); } catch( InterruptedException exception ) {}
      server.moveTo( "localhost:8000", "atProgram", "moved again" );

      // Allow time for completion
     try{ Thread.sleep( 10000 ); } catch( InterruptedException exception ) {}

      Voyager.shutdown();
      }
    catch( VoyagerException exception )
      {
      System.err.println( exception );
      }
    }
  }
```

Move Exceptions

A move can fail for several reasons, including the following:

- The address of the destination program is specified in an illegal format.
- A network connection to the destination program cannot be established.
- The destination is an object that cannot be found.
- A network communications error occurs during the move.
- The object to be moved is not serializable.
- An error occurs during object serialization.
- The callback function does not exist or is not public.
- The object's class was not processed by the `vcc` utility.

If any of these exceptions occur during a call to `moveTo()`, the move is aborted and an exception is thrown.

If a move completes successfully and the object throws an uncaught exception in the callback, then the Voyager destination virtual machine catches the exception and generates an `ObjectEvent.UNHANDLED_EXCEPTION` event.

The `Exceptions.java` example program demonstrates many of these exception conditions. Use the following command from the `\voyager1.0.0\examples\mobility` directory to compile the example:

```
javac Exceptions.java
```

Start a server on port 8000 in one window, and then run `Exceptions.class` in a second window.

Window 1

```
>voyager 8000
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:8000
```

Window 2

```
>java examples.mobility.Exceptions
construct a server in localhost
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.89:1196
construct Server( Thomas )
square( 3 ) = 9
move to illegal address localhost:xxx
COM.objectspace.voyager.InvalidAddressException: Illegal port in address
        thrown remotely from 208.6.239.89:1196
move to nonexistent object localhost/Fred
COM.objectspace.voyager.ObjectNotFoundException: Fred
        thrown remotely from 208.6.239.89:1196
move server to localhost:8000 with bad callback
COM.objectspace.voyager.MethodNotFoundException:
examples.mobility.Server.nonexi
stent()
        thrown remotely from 208.6.239.89:8000
move server to nonexistent localhost:9000
COM.objectspace.voyager.TransportException: java.net.ConnectException:
Connectio
n refused
square( 4 ) = 16
>
```

Application voyager1.0.0\examples\mobility\Exceptions.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.mobility;

import COM.objectspace.voyager.*;

public class Exceptions
  {
  public static void main( String args[] )
    {
    try
      {
      System.out.println( "construct a server in localhost" );
      VServer server = new VServer( "Thomas", "localhost" );
      System.out.println( "square( 3 ) = " + server.square( 3 ) );

      try
        {
        System.out.println( "move to illegal address localhost:xxx" );
        server.moveTo( "localhost:xxx" );
        }
      catch( InvalidAddressException exception )
        {
        System.out.println( exception );
        }
```

```
      try
        {
        System.out.println( "move to nonexistent object localhost/Fred" );
        server.moveTo( "localhost/Fred" );
        }
      catch( ObjectNotFoundException exception )
        {
        System.out.println( exception );
        }

      try
        {
       System.out.println("move server to localhost:8000 with bad callback");
        server.moveTo( "localhost:8000", "nonexistent" );
        }
      catch( MethodNotFoundException exception )
        {
        System.out.println( exception );
        }

      try
        {
        System.out.println( "move server to nonexistent localhost:9000" );
        server.moveTo( "localhost:9000" );
        }
      catch( TransportException exception )
        {
        System.out.println( exception );
        }

      // show that the object is still ok after all the exceptions
      System.out.println( "square( 4 ) = " + server.square( 4 ) );
      Voyager.shutdown();
      }
    catch( VoyagerException exception )
      {
      System.err.println( exception );
      }
    }
  }
```

Message Forwarding

You may disable message forwarding using `setForwarding()`. If `setForwarding()` is set to `false`, an object does not drop a forwarder when moved. Messages sent to the object's old location cause an `ObjectNotFoundException` to be thrown.

Use the following command from the `\voyager1.0.0\examples\mobility` directory to compile the `Forwarding.java` example program:

```
javac Forwarding.java
```

Start a server on port 8000 in one window, and then run `Forwarding.class` in a second window.

Window 1

```
>voyager 8000
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:8000
Server( Server ) arrives at new program
```

Window 2

```
>java examples.mobility.Forwarding
construct server in local program
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1228
construct Server( Server )
square( 3 ) = 9
disable forwarding...
move server to program on port 8000
request a calculation from the moved server...
COM.objectspace.voyager.ObjectNotFoundException:
27-142-14-250-1-149-14-4-10-128
-72-134-111-72-132-76
        thrown remotely from 208.6.239.200:1228
>
```

Application voyager1.0.0\examples\mobility\Forwarding.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.mobility;

import COM.objectspace.voyager.*;

public class Forwarding
  {
  public static void main( String args[] )
    {
    try
      {
      System.out.println( "construct server in local program" );
      VServer server = new VServer( "Server", "localhost" );
      System.out.println( "square( 3 ) = " + server.square( 3 ) );
      System.out.println( "disable forwarding..." );
      server.setForwarding( false );
      System.out.println( "move server to program on port 8000" );
      server.moveTo( "localhost:8000", "atProgram" );

      try
        {
        System.out.println("request a calculation from the moved server...");
        System.out.println( "square( 3 ) = " + server.square( 3 ) );
        }
      catch( ObjectNotFoundException exception )
        {
        System.out.println( exception );
        }

      Voyager.shutdown();
      }
    catch( VoyagerException vexception )
      {
      System.out.println( vexception );
      }
    }
  }
```

9

Agents

An ObjectSpace Voyager™ Core Technology (Voyager) agent can be moved like all simple objects in Voyager; however, unlike a simple object, a Voyager agent can move itself and continue to execute. Autonomy is useful for many reasons.

- If a task must be performed independently of the computer that launches the task, a mobile agent can be created to perform this task. Once constructed, the agent can move into the network and complete the task in a remote program.
- If a program needs to send a large number of messages to objects in remote programs, an agent can be constructed to visit each program in turn and send the messages locally. Local messages are often between 1,000 and 1,000,000 times faster than remote messages.
- If you want to partition your programs to execute in parallel, you can distribute the processing to several agents, which migrate to remote programs and communicate with each other to achieve the final goal.
- If periodic monitoring of a remote object is required, creating an agent that meets the remote object and monitors it locally is more efficient than monitoring the device across the network.
- If a series of operations must be performed inside a consumer device that is only occasionally connected to a network, such as a Java™ phone or Java pager, then an agent can move into the device, perform its task, and move back into the network only when necessary.

It is important to avoid “force-fitting” agent technology into a program. Voyager’s remote messages are adequate for many applications, and simple object mobility is often enough to close the gap between two objects communicating on a network. However, as you become familiar with the power of agents and the ease of creating them, you will find many ways to agent-enhance your current and future programs.

Creating an Agent

To create an agent, extend `COM.objectspace.voyager.Agent`. You do not have to define or override any special methods. All agents are automatically serializable and have a default life span of one day.

Moving to a Program

All rules associated with moving a simple object, described in Chapter 8, “Mobility,” also apply to moving an agent. However, an agent can also independently move to a program by sending itself the `moveTo()` message.

A successful call to `moveTo()` conceptually causes the thread of control to stop in the agent before it moves and to resume from the callback function in the agent after it moves. Therefore, only exception-handling code should follow a `moveTo()`. However, there is no Java mechanism to prevent a developer from following a call to `moveTo()` with additional source code. In this situation, after moving, the agent continues to execute the code at its original location, which is clearly incorrect.

When the callback function completes, the agent does not die—it simply becomes inactive until it receives another message. Therefore, after an agent has performed its duties, it can park while awaiting further instructions.

In the `Launch.java` and `Dismiss.java` example programs, the `Launch` program creates a `SalesPerson` agent that moves to each program in its itinerary and then parks. The `Dismiss.java` program contacts the parked `SalesPerson` and tells it to die. Note that the `Dismiss.java` program can connect to the `SalesPerson` via its original home address because the `SalesPerson` leaves forwarders behind.

From the `\voyager1.0.0\examples\agents` directory, use the following commands to prepare and compile `Launch.java` and `Dismiss.java`:

```
vcc SalesPerson
javac SalesPerson.java VSalesPerson.java Launch.java Dismiss.java
```

Start a server on each of ports 7000, 8000, and 9000 in three separate windows. In a fourth window, first run `Launch.class` to launch the agent, and then, when `Launch.class` completes, run `Dismiss.class` in the same window.

Window 1

```
>voyager 7000
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:7000
visiting 208.6.239.200:7000 for 3s
moving to localhost:8000
```

Window 2

```
>voyager 8000
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:8000
visiting 208.6.239.200:8000 for 3s
moving to localhost:9000
```

Window 3

```
>voyager 9000
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:9000
visiting 208.6.239.200:9000 for 3s
parking at 208.6.239.200:9000
dismiss
```

Window 4

```
>java examples.agents.Launch
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1307
moving to localhost:7000
SalesPerson is at 208.6.239.200:7000
SalesPerson is at 208.6.239.200:7000
SalesPerson is at 208.6.239.200:8000
SalesPerson is at 208.6.239.200:8000
SalesPerson is at 208.6.239.200:9000
>java examples.agents.Dismiss
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1313
>
```

Class voyager1.0.0\examples\agents\SalesPerson.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.agents;

import COM.objectspace.voyager.*;
import java.util.Vector;

public class SalesPerson extends Agent
  {
  Vector itinerary = new Vector(); // vector of applications to visit
  int index; // index of current application

  public void addToItinerary( String address )
```

```
  {
  itinerary.addElement( address );
  }

public void launch()
  {
  next();
  }

private void next()
  {
  if( index < itinerary.size() )
    {
    String destination = (String) itinerary.elementAt( index++ );
    System.out.println( "moving to " + destination );

    try
      {
      moveTo( destination, "atProgram" );
      }
    catch( VoyagerException exception )
      {
      System.out.println(exception);
      }
    }
  else
    {
    System.out.println( "parking at " + Voyager.getAddress() );
    }
  }

public void atProgram()
  {
  System.out.println( "visiting " + Voyager.getAddress() + " for 3s" );

  try{ Thread.sleep( 3000 ); } catch( InterruptedException exception ) {}

  next();
  }

public void dismiss() throws VoyagerException
  {
  System.out.println( "dismiss" );
  dieNow(); // kill myself and all my forwarders
  }
}
```

Application voyager1.0.0\examples\agents\Launch.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.agents;

import COM.objectspace.voyager.*;

public class Launch
  {
  public static void main( String args[] )
    {
    try
      {
      VSalesPerson person = new VSalesPerson( "localhost/Fred" );
      person.addToItinerary( "localhost:7000" );
      person.addToItinerary( "localhost:8000" );
      person.addToItinerary( "localhost:9000" );
      person.launch(); // send the agent on its way

      for( int i = 1; i <= 5; i++ ) // show progress of the agent
        {
        String address = person.getProgramAddress();
        System.out.println( "SalesPerson is at " + address );
        try{ Thread.sleep( 2000 ); } catch(InterruptedException exception) {}
        }

      Voyager.shutdown();
      }
    catch( VoyagerException exception )
      {
      System.err.println( exception );
      }
    }
  }
```

Application voyager1.0.0\examples\agents\Dismiss.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.agents;

import COM.objectspace.voyager.*;

public class Dismiss
  {
  public static void main( String args[] )
    {
    try
      {
      VSalesPerson person =
            (VSalesPerson) VObject.forObjectAt( "localhost:7000/Fred" );
      person.dismiss();
      Voyager.shutdown();
      }
    catch( VoyagerException exception )
      {
      System.err.println( exception );
      }
    }
  }
```

Moving to Another Object

In addition to the capability of moving itself to a program, an agent can also use the `moveTo()` message to move itself to a remote object.

This function causes the agent to move into the same program as the remote object—following forwarders if necessary to find the object's new location—and then execute a callback with a native Java reference to the remote object and the optional callback argument, if supplied. As with simple objects, if one or more agents move to a remote object, the remote object cannot move until all callbacks have completed.

The `Browse.java` example program creates a `Librarian` agent that moves to a `Library` to select all book titles comprised of exactly five characters. After storing all relevant titles, the `Librarian` moves home, displays the titles, and then dies. To reduce the class coupling discussed on page 166, the `Librarian` defines the move callback to accept a `Library` interface rather than a `LibraryImpl` class.

From the `\voyager1.0.0\examples\agents` directory, use the following commands to prepare and compile `Browse.java`:

```
vcc LibraryImpl
vcc Librarian
javac LibraryImpl.java VLibraryImpl.java Librarian.java VLibrarian.java
javac Browse.java
```

Start a server on port 8000 in one window, and then run `Browse.class` in a second window.

Window 1

```
>voyager 8000
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:8000
constructing a library with 1000 titles
at library, searching thru titles...
26 titles selected
moving home...
```

Window 2

```
>java examples.agents.Browse
construct library
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1318
move to library and then browse titles...
leaving 208.6.239.200:1318, moving to library...
arrived back home...
selected titles: [brtkf, judib, zxgwz, tmolz, wpdec, ipkce, ioryo, rzpjz,
kcsuu, ktokd, vpskv, uvvkj, wgfvn, kpigw, thzri, zxiht, usfzc, qvldf,
jcgxx, nkfxa, bhqx, luqqd, cihku, htdqv, chupk, ecdeo]
```

Interface voyager1.0.0\examples\agents\Library.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.agents;

import COM.objectspace.voyager.*;

public interface Library
  {
  String getTitle( int i );
  int getBookCount();
  }
```

Class voyager1.0.0\examples\agents\LibraryImpl.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.agents;

import java.util.*;

public class LibraryImpl implements Library
  {
  static final int TITLES = 1000; // number of titles in the library
  Vector titles = new Vector();

  public LibraryImpl()
    {
    System.out.println( "constructing a library with " + TITLES + " titles" );
    Random random = new Random();

    // fill the library with random titles
    // the length of each title ranges from 1 thru 50
    for( int i = 0; i < TITLES; i++ )
      {
      int length = Math.abs( random.nextInt() ) % 50 + 1;
      StringBuffer title = new StringBuffer();

      for( int j = 0; j < length; j++ )
        {
        int ascii = Math.abs( random.nextInt() ) % 26 + (int) 'a';
        title.append( (char) ascii );
        }

      titles.addElement( title.toString() );
      }
    }

  public String getTitle( int i )
    {
    return (String) titles.elementAt( i );
    }

  public int getBookCount()
```

```
    {
    return titles.size();
    }
  }
```

Class voyager1.0.0\examples\agents\Librarian.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.agents;

import java.util.Vector;
import COM.objectspace.voyager.*;

public class Librarian extends Agent
  {
  String home; // address of home
  Vector selection; // selected titles

  public void browse( VLibraryImpl library ) throws VoyagerException
    {
    System.out.println( "move to library and then browse titles..." );
    home = Voyager.getAddress();
    System.out.println( "leaving " + home + ", moving to library..." );
    moveTo( library, "atLibrary" );
    }

  public void atLibrary( Library library ) throws VoyagerException
    {
    System.out.println( "at library, searching thru titles..." );

    selection = new Vector();
    int count = library.getBookCount();

    for( int i = 0; i < count; i++ )
      {
      String title = library.getTitle( i );

      if( title.length() == 5 )
        selection.addElement( title );
      }

    System.out.println( selection.size() + " titles selected" );
    System.out.println( "moving home..." );
    moveTo( home, "atHome" );
    }

  public void atHome() throws VoyagerException
    {
    System.out.println( "arrived back home..." );
    System.out.println( "selected titles: " + selection );
    dieNow();
    }
  }
```

Application voyager1.0.0\examples\agents\Browse.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.agents;

import COM.objectspace.voyager.*;

public class Browse
  {
  public static void main( String args[] )
    {
    try
      {
      System.out.println( "construct library" );
      VLibraryImpl library = new VLibraryImpl( "localhost:8000" );
      library.liveForever(); // prevent garbage collection
      VLibrarian librarian = new VLibrarian( "localhost" );
      librarian.browse( library );
      }
    catch( VoyagerException exception )
      {
      System.err.println( exception );
      }
    }
  }
```

Moving to a Moving Object

An agent follows forwarders as it tries to move to a moving object. When the agent arrives at the moving object, the agent prevents the object from further movement until the callback completes.

The `CatChase.java` example program uses a cat to chase a moving mouse. Note that even though the mouse tries to move after four seconds, it is prevented from actually moving because the cat catches the mouse and plays with it for eight seconds. When the cat finishes playing with the mouse and moves home, the mouse is released and thus free to move.

From the `\voyager1.0.0\examples\agents` directory, use the following commands to prepare and compile `CatChase.java`:

```
vcc Cat Mouse
javac Cat.java VCat.java Mouse.java VMouse.java CatChase.java
```

Start a server on each of ports 8000 and 9000 in two separate windows, and then run `CatChase.class` in a third window.

Window 1

```
>voyager 8000
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:8000
mouse at localhost:8000
mouse resting 4 seconds before running...
cat catches mouse and plays with it
squeak!
cat rests for 8 seconds...
mouse tries to run to localhost:9000...
cat lets mouse go, moves home
```

Window 2

```
>voyager 9000
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:9000
mouse escapes to localhost:9000!
```

Window 3

```
>java examples.agents.CatChase
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1298
mouse running to localhost:8000...
cat waiting 2 seconds until chase...
cat chases mouse at
208.6.239.200:1298/216-90-198-113-118-139-169-66-130-16-72-1
34-111-72-133-146
cat arrived home
```

Class voyager1.0.0\examples\agents\Cat.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.agents;

import COM.objectspace.voyager.*;

public class Cat extends Agent
  {
  String home;

  public void chase( VMouse mouse ) throws VoyagerException
    {
    home = Voyager.getAddress(); // remember where I came from
    System.out.println( "cat waiting 2 seconds until chase..." );
    try{ Thread.sleep( 2000 ); } catch( InterruptedException exception ) {}
    System.out.println( "cat chases mouse at "+mouse.getLastObjectAddress()
);
    moveTo( mouse, "atMouse" );
    }

  public void atMouse( Mouse mouse ) throws VoyagerException
    {
    System.out.println( "cat catches mouse and plays with it" );
    mouse.play();
    System.out.println( "cat rests for 8 seconds..." );
    try{ Thread.sleep( 8000 ); } catch( InterruptedException exception ) {}
    System.out.println( "cat lets mouse go, moves home" );
    moveTo( home, "atHome" );
    }

  public void atHome() throws VoyagerException
    {
    System.out.println( "cat arrived home" );
    dieNow();
    }
  }
```

Class voyager1.0.0\examples\agents\Mouse.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.agents;

import COM.objectspace.voyager.*;

public class Mouse extends Agent
  {
  public void hide() throws VoyagerException
    {
    System.out.println( "mouse running to localhost:8000..." );
    moveTo( "localhost:8000", "at8000" );
    }

  public void at8000() throws VoyagerException
    {
    System.out.println( "mouse at localhost:8000" );
    System.out.println( "mouse resting 4 seconds before running..." );
    try{ Thread.sleep( 4000 ); } catch( InterruptedException exception ) {}
    System.out.println( "mouse tries to run to localhost:9000..." );
    moveTo( "localhost:9000", "at9000" );
    }

  public void play()
    {
    System.out.println( "squeak!" );
    }

  public void at9000() throws VoyagerException
    {
    System.out.println( "mouse escapes to localhost:9000!" );
    dieNow();
    }
  }
```

Application voyager1.0.0\examples\agents\CatChase.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.agents;

import COM.objectspace.voyager.*;

public class CatChase
  {
  public static void main( String args[] )
    {
    try
      {
      VMouse mouse = new VMouse( "localhost" );
      mouse.hide();
      VCat cat = new VCat( "localhost" );
      cat.chase( mouse );
      }
    catch( VoyagerException exception )
      {
      System.err.println( exception );
      }
    }
  }
```

Releasing an Object Early

To release a destination object before a callback has completed, an agent must send itself `releaseObject()`. Once an object is released, it is free to move. At this point, the callback argument is a *stale reference* and should not be used by the agent, because that object can move from the current program at any time.

The `DogChase.java` example program uses a dog to chase a moving mouse. When the dog (who is nicer than the cat) moves to the mouse, the dog immediately releases the mouse so that the mouse can continue to move.

From the `\voyager1.0.0\examples\agents` directory, use the following commands to prepare and compile `DogChase.java`:

```
vcc Dog
javac Dog.java VDog.java DogChase.java
```

Start a server on each of ports 8000 and 9000 in two separate windows, and then run `DogChase.class` in a third window.

Window 1

```
>voyager 8000
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:8000
mouse at localhost:8000
mouse resting 4 seconds before running...
dog catches mouse and plays with it
squeak!
dog lets mouse go...
mouse tries to run to localhost:9000...
dog moves home
```

Window 2

```
>voyager 9000
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:9000
mouse escapes to localhost:9000!
```

Window 3

```
>java examples.agents.DogChase
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1298
mouse running to localhost:8000...
cat waiting 2 seconds until chase...
cat chases mouse at
208.6.239.200:1298/216-90-198-113-118-139-169-66-130-16-72-1
34-111-72-133-146
cat arrived home
```

Class voyager1.0.0\examples\agents\Dog.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.agents;

import COM.objectspace.voyager.*;

public class Dog extends Agent
  {
  String home;

  public void chase( VMouse mouse ) throws VoyagerException
    {
    home = Voyager.getAddress(); // remember where I came from
    System.out.println( "dog waiting 2 seconds until chase..." );
    try{ Thread.sleep( 2000 ); } catch( InterruptedException exception ) {}
    System.out.println( "dog chases mouse at "+mouse.getLastObjectAddress()
);
    moveTo( mouse, "atMouse" );
    }

  public void atMouse( Mouse mouse ) throws VoyagerException
    {
    System.out.println( "dog catches mouse and plays with it" );
    mouse.play();
    System.out.println( "dog lets mouse go..." );
    releaseObject(); // allow the mouse to move
    try{ Thread.sleep( 8000 ); } catch( InterruptedException exception ) {}
    System.out.println( "dog moves home" );
    moveTo( home, "atHome" );
    }

  public void atHome() throws VoyagerException
    {
    System.out.println( "dog arrived home" );
    dieNow();
    }
  }
```

Application voyager1.0.0\examples\agents\DogChase.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.agents;

import COM.objectspace.voyager.*;

public class DogChase
  {
  public static void main( String args[] )
    {
    try
      {
      VMouse mouse = new VMouse( "localhost" );
      mouse.hide();
      VDog dog = new VDog( "localhost" );
      dog.chase( mouse );
      }
    catch( VoyagerException exception )
      {
      System.err.println( exception );
      }
    }
  }
```

10

Applets

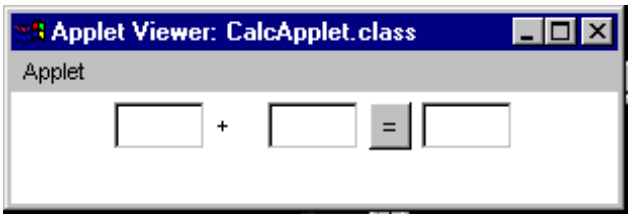
An applet is a program that is stored on a server and executed remotely in a client browser. This chapter describes the process for creating Voyager-enabled applets by examining three examples. Each example is more complex than the preceding one.

- A *calculator* applet performs calculations using a remote calculator stored in a server. The process for running an applet either in your local machine or from a Web server is explained.
- A *chat* applet demonstrates applet messaging and applet-to-applet communications.
- A *shopper* applet shows how agents can be launched from applets and can communicate with applets as they move.

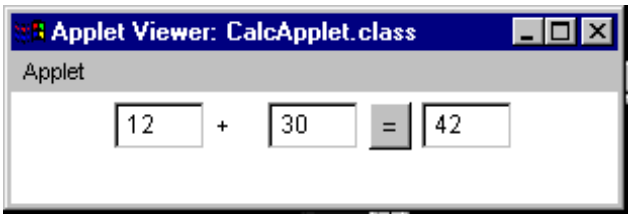
Note: All ObjectSpace Voyager™ Core Technology applets are considered nontrusted and are fully restrained by Java's sandbox model. Future versions of Voyager will fully support signed applets and more liberal security policies.

Calculator Example — Getting Started

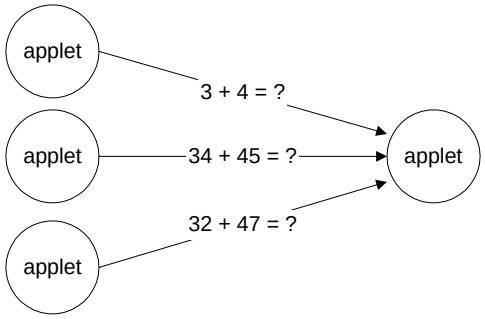
The first example is a simple calculator applet that uses Voyager's object request broker features. The applet is pictured below.



When you enter a number into the first two text fields and select the = button, the result is displayed in the third text field.



The architecture of the applet is simple. The server program executes on port 6000 and creates an instance of `Calculator` with the alias `MyCalculator`. When the applet begins, it connects to the remote calculator. When the user selects the = button, `add()` is sent to the remote calculator, the calculator returns the result, and the result displays. Because the server is multithreaded, it can support many simultaneous applet requests.



In general, a Voyager-enabled applet must be associated with a Voyager program that acts as its server; however, many applets can share the same server. A server provides a Voyager-enabled applet with two important services.

- Performs network class loading on behalf of the applet, thereby automatically installing classes in the applet's codebase (by fetching them from the network) and creating the appropriate class files in the applet's file system.
- Acts as a software router for messages and objects that want to move from the applet to another applet or to an arbitrary program. This routing is necessary because a nontrusted applet can establish only a network connection to its server.

For an example of the software routing capability of an applet server, see the chat example on page 201.

The calculator application is comprised of the following example program files.

- `Calculator.java`

  The calculator. This program is straightforward and does not contain any code that is specific to applet programming.

- `CalcServer.java`

  The program that creates the calculator and makes it available to applets. This program looks like a typical Voyager program except it passes an extra argument to `Voyager.startup()`. All programs that serve an applet must do this. The extra argument specifies the root directory of all Web pages on the Web server. The specified root directory is used by the server to perform network class loading on behalf of an applet. If you wish to run an applet on your local machine without a Web server, set the root directory to backslash (\) in Windows and to forward slash (/) in UNIX. Rather than hardcoding these values, pass them as command line arguments so you can execute the same program with or without a Web server.

- `CalcApplet.java`

  The applet that uses the remote calculator on the server. A Voyager-enabled applet is similar to a standalone Voyager program except the applet must initialize itself using either `startup( Applet applet )` or `startup( Applet applet, int port )`. Each of these startup routines causes the applet to establish a network connection with its associated server program. All applets must initialize themselves in this way, typically as the first line of `init()`. To connect to the remote calculator, the calculator applet gets the address of its server program using `Voyager.getServerAddress()`, which in turn gets the information from `Calculator.html`.

- `Calculator.html`

  The HTML code that describes the applet and its associated server. This program contains the information required by the Web browser and the applet to run the program. It uses standard HTML tags to specify the following information.

  - The title of the applet.
  - The codebase of the applet. If omitted, the codebase of the applet is the same as the directory that contains the `.html` file. In this example, the calculator program is not stored in a package, so the codebase can be defaulted to the location of `Calculator.html`.
  - The path to the class that defines the applet, relative to the codebase.
  - An optional list of `.jar` files to load before the applet. The archive paths are relative to the codebase. Voyager applets typically have `voyager1.0.0.jar` in this list of Java archives.
  - The initial width and height of the applet.

  In addition, each `.html` file that declares a Voyager applet must set the Voyager-specific server property to the port number of the applet's server program. The following example uses port 6000.

The preferred format for downloading applets over the Web is the Java™ archive (jar) format. A `.jar` file is a collection of `.class` files archived into one file for easy Web distribution. The following examples demonstrate applets using Voyager packaged in a `.jar` file. These examples assume that the `voyager1.0.0.jar` file is located in a directory named `\lib` relative to the codebase.

Compiling the Programs

From the `\voyager1.0.0\examples\applets\calculator` directory, use the following commands to compile the calculator programs.

```
vcc Calculator
javac *.java
```

Running the Applet from a Local Machine

To run the applet on your local machine without a Web server, first start the calculator server in one window.

Window 1

```
>java CalcServer 6000 \
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:6000
```

Note: Set the Web root to a backslash (\) in Windows and a forward slash (/) in UNIX, because the applet is running from a local machine rather than on a Web server.

Now run the client applet in the `.html` file in a second window by invoking the applet viewer or a JDK 1.1-enabled browser from the directory `\voyager1.0.0\examples\applets\calculator`.

Window 2

```
>appletviewer Calculator.html
voyager(tm) 1.0.0, copyright objectspace 1997
address = 127.0.0.1:6000;127.0.0.1:1052
```

Running the Applet from a Web Server

To run a Voyager applet from a Web server, first install the Voyager core package on the Web server. There are many different ways to organize applets in a server, including the preferred approach described in this section.

First, make a note of the directory your Web server treats as the root for all Web pages. For example, the ObjectSpace Web server named `pulsar` stores Web pages in the `\webpages` directory.

Next, create a directory structure in your user account under the Web root. This directory structure should mirror the structure from your local machine and should copy across the contents of the `bin` and `lib` directories. You need not copy the `vcc` executable files because they are required for development purposes only. Add the `\bin` directory to your path setting on the Web server. The directory structure can be created in any subdirectory of the Web root. For example, if your account is rooted in `\webpages\gglass` on `pulsar`, you might create the following directory structure on `pulsar`:

```
\webpages
   \gglass
      \voyager1.0.0
         \bin        Voyager executable files
         \lib        voyager1.0.0.jar
```

When your directory structure is in place, you are ready to install specific applets on the Web server. To install an applet, copy across the directory structure that contains the applet code, and then copy across the applet's `.html` and `.class` files. You need not copy the `.java` files.

For example, to install the calculator applet on the `pulsar` Web server, add the directories and files highlighted below.

```
\webpages
   \gglass
      \voyager1.0.0
         \bin
         \examples
            \applets
               \calculator    Calculator .html and all .class files
         \lib
```

You can now run the applet. Open a window in the Web server, move to the `calculator` directory, and use the following command to start the calculator server:

Window 1

```
>java CalcServer 6000 \webpages
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:6000
```

where *webpages* is your Web server's root. To run the applet from your local machine, open a second window and invoke the appletviewer or browser with the path name of the applet's `.html` file relative to the Web root.

Window 2

```
>appletviewer
http://pulsar/gglass/voyager1.0.0/examples/applets/calculator/Calculator.html
```

where *pulsar/gglass* is the name of your Web server and the path of your user account.

Class voyager1.0.0\examples\applets\calculator\Calculator.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

public class Calculator
  {
  public int add( int x, int y )
    {
    return x + y;
    }
  }
```

Class voyager1.0.0\examples\applets\calculator\CalcServer.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

import COM.objectspace.voyager.*;

public class CalcServer
  {
  public static void main( String args[] )
    {
    try
      {
      int port = Integer.parseInt( args[ 0 ] ); // port to serve
      String root = args[ 1 ]; // root of server's visible directory tree
      Voyager.startup( port, root ); // serve applets

      // create immortal calculator
      VCalculator calculator = new VCalculator( "localhost/MyCalculator" );
      calculator.liveForever();
      }
    catch( VoyagerException exception )
      {
      System.err.println( exception );
      }
    }
  }
```

Applet voyager1.0.0\examples\applets\calculator\CalcApplet.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import COM.objectspace.voyager.*;

public class CalcApplet extends Applet
  {
  TextField operand1 = new TextField( "", 3 );
  Label plus = new Label( "+" );
  TextField operand2 = new TextField( "", 3 );
  Button add = new Button( "=" );
  TextField result = new TextField( "", 3 );
  VCalculator calculator;

  public void init()
    {
    try
      {
      Voyager.startup( this ); // startup in applet mode
      setLayout( new FlowLayout() );
      add( operand1 );
      add( plus );
      add( operand2 );
      add( add );
      add( result );
      add.addActionListener( new ActionListener()
      { public void actionPerformed( ActionEvent event ) { add(); } } );
      String address = Voyager.getServerAddress() + "/MyCalculator";
      // connect to remote calculator in server
      calculator = (VCalculator) VObject.forObjectAt( address );
      }
    catch( VoyagerException exception )
      {
      System.err.println( exception );
      }
    }

  void add()
    {
    int x = Integer.parseInt( operand1.getText() );
    int y = Integer.parseInt( operand2.getText() );

    try
      {
      result.setText( Integer.toString( calculator.add( x, y ) ) );
      }
    catch( VoyagerException exception )
      {
      result.setText("Error");
      }
    }
  }
```

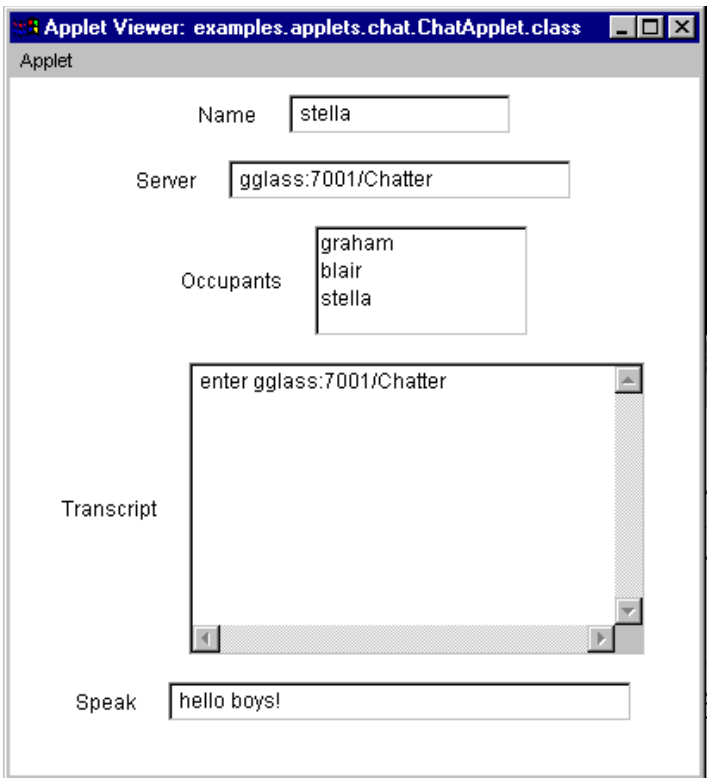
HTML voyager1.0.0\examples\applets\calculator\Calculator.html

```
<title>Calculator Applet</title>
<hr>
<applet code="CalcApplet.class" archive="../../../lib/voyager.jar" width=300
height=50>
<param name=Server value=6000>
</applet>
<hr>
```

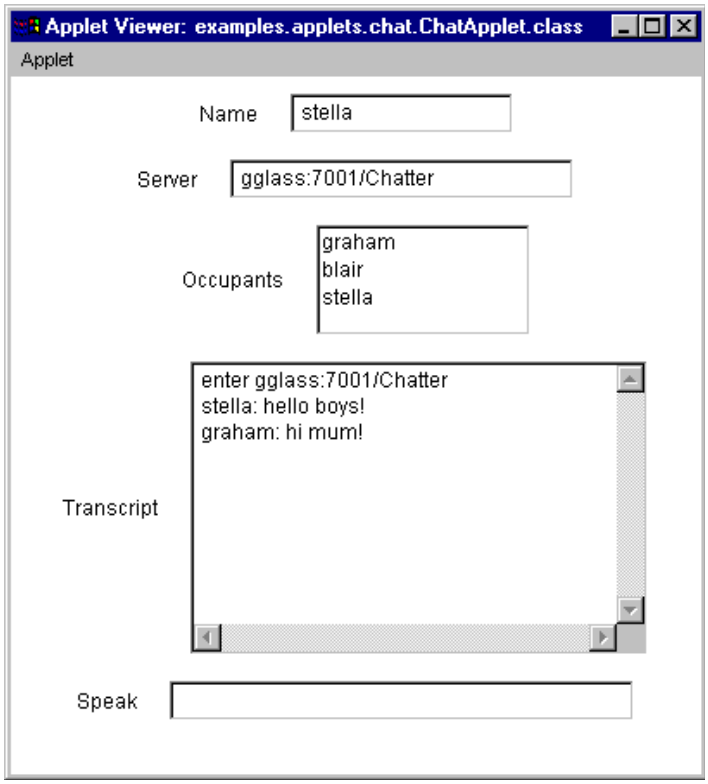
Chat Example — Applet Communications

This section presents a chat applet that allows you to enter a chat room and communicate with the occupants. After staring the applet, enter your name in the Name field and the address of the desired chat room in the Server field. The names of the chat room occupants display in the Occupants field. Then enter a message in the Speak field and press Enter to send it to all chat room occupants. The Speak field clears when you send the message. All messages exchanged in the current chat room display in the Transcript field. To leave the current chat room, close the chat applet or enter the address of another chat room.

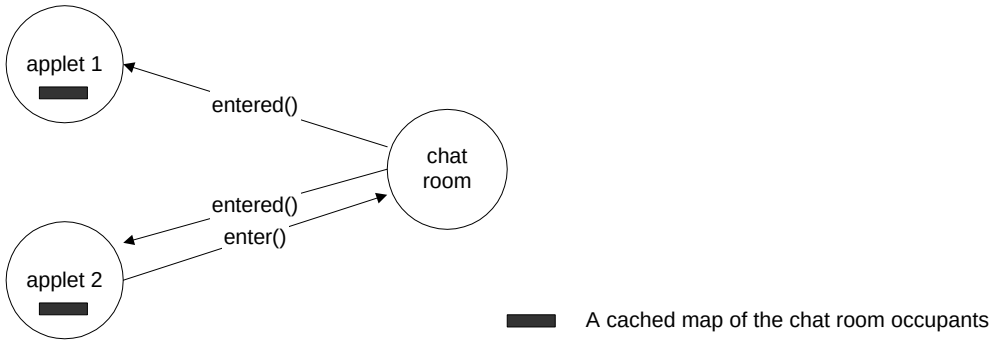
The picture below shows a chat applet after Stella enters the chat room at gglass:7001/Chatter and is about to send the message hello boys! to the other two occupants.



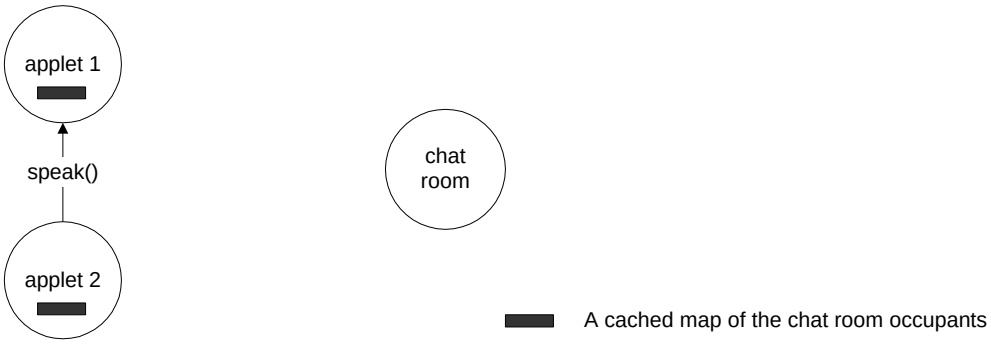
Pictured below is the chat applet after Stella sends the hi boys! message, and Graham sends back a hi mum! message.



The architecture of the chat application uses direct applet-to-applet communication. When an applet connects to a chat room, the applet asks the chat room for a hash table of its occupants. The hash table associates the name of an occupant with a virtual reference to that occupant's applet. A copy of the hash table is then cached in the applet. The applet sends to the chat room the `enter()` message with the name of the new occupant and a virtual reference to the new applet. The chat room in turn updates each of its occupant's applets using the `entered()` message. This mechanism keeps the occupant caches of each chat applet current. The `leave()` and `leaved()` messages are used in a similar way to update the occupant caches when someone leaves a chat room. The following diagram illustrates the message flow that occurs when `applet 2` joins the chat room.



When text is entered in the Speak field, the applet uses its cached map of the chat room occupants to send `speak()` directly to each of the occupant's applets. Hence, the chat room is used only for tracking the occupants of the chat room and is not involved in the chat room conversations. The diagram below shows the message flow when `applet 2` sends a message to each of the other applets in the chat room.



Two features of Voyager support easy implementation of this architecture.

- Voyager allows an applet to be processed using `vcc` so that remote messages can be sent directly to an applet.
- Voyager allows messages to be sent directly between applets or from an applet to any program.

Preparing an Applet for Remote Messaging

There are two ways to process an applet class using `vcc` so that the applet can receive remote messages. Assume your applet class `myPackage.MyApplet` has the following class derivation:

```
java.lang.Object
  java.awt.Component
    java.awt.Container
      java.awt.Panel
        java.applet.Applet
          myPackage.MyApplet
```

To send any message to your applet, including messages defined in its superclasses, first run `vcc` on each of the standard `java.*` classes in turn, starting with `java.awt.Component`. Use the `-p` option to place the virtual classes into the same package as your program.

```
vcc -p myPackage java.awt.Component java.awt.Container java.awt.Panel
  java.applet.Applet
```

Note: For the chat example, run `vcc` from `\voyager1.0.0\examples\applets\chat`.

Then run `vcc` on your applet class using the `-m` option, which places the applet's superclass `java.applet.Applet` into `myPackage`.

```
vcc -m java.applet.Applet myPackage myPackage.MyApplet
```

If you want to send only your applet the messages that are defined in `MyApplet`, avoid processing the `java.*` classes by running `vcc` on the applet class using the `-e` option with `java.lang.Object`. This option processes your applet class as if it extended `java.lang.Object`.

```
vcc -e java.lang.Object myPackage.MyApplet
```

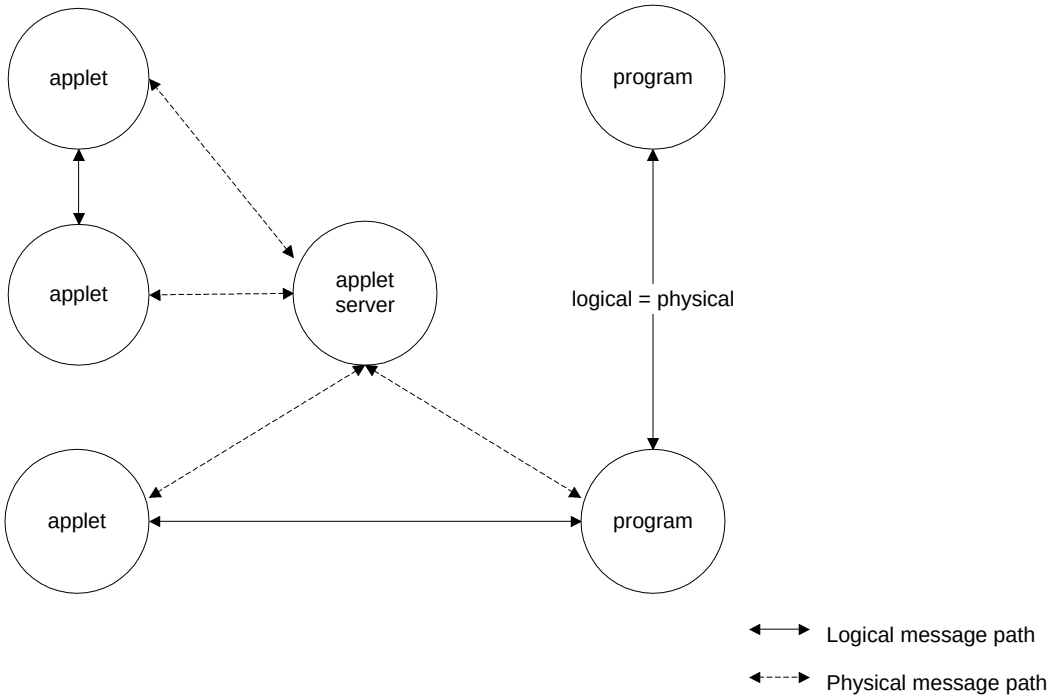
In the chat example, the only applet methods invoked remotely are `entered()`, `speak()`, and `left()`, which are defined in `ChatApplet`. The virtual version of `ChatApplet` is therefore created using the `-e` option.

Network Routing

For security reasons, an applet can open only a direct network connection to its server. However, the architecture of the chat program requires that an applet be able to send a message directly to another applet and to an arbitrary chat room program. Voyager supports this level of flexibility by allowing an applet's server program to function as a router. The routing mechanism is based on two simple rules.

- The full address of an applet is a two-stage address with the format *server address*`;`*applet address*. For example, if an applet's address is `gglass:1456` and its server address is `gglass:7000`, the full address of the applet is `gglass:7000;gglass:1456`. When a message or object is sent to an applet, it travels to the first stage of the address first and then completes the journey to the second stage.
- When a message or object is sent from an applet, it always travels to the applet's server first, regardless of its final destination. When the message or object arrives at the server, it awakens and continues on its journey.

When these two rules are combined, the result is a powerful, multistage routing system. The following diagram shows the logical and physical network traffic between some applets and programs.



The chat application is comprised of the following files:

- `ChatRoom.java`

  Allows one or more chat rooms to be created. This program maintains a hash table that tracks all occupants in the chat room and updates each occupant as other occupants enter and leave. It defines a `main()` function that allows creation of a new chat room on the local host with a specified port and alias.

- `ChatApplet.java`

  Allows a user to enter a chat room. This program uses `startup( Applet )` to initialize itself. When it enters a chat room, it sends the chat room the name of the occupant together with a virtual reference to the applet so the chat room can communicate back to the applet at a future time.

- `Chat.html`

  Describes the applet and its associated server in HTML code. This program defines the applet codebase as three directory levels above itself because the applet is stored in a package that is relative to `\voyager1.0.0` on the Web server. For the chat example, the applet's server program is assigned to port 7000.

Compiling the Programs

To compile the chat example programs, use the following commands from the `\voyager1.0.0\examples\applets\chat` directory:

```
vcc -e java.lang.Object ChatApplet
vcc ChatRoom
javac *.java
```

Running the Applet from a Local Machine

To run the chat applet on your local machine without a Web server, first open a window and run the applet server from any location.

Window 1

```
>voyager 7000
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:7000
```

Then, from `\voyager1.0.0\examples\applets\chat`, start a chat room named `Chatter` on port 7001 in a second window. Each chat room must have a unique port number and name.

Window 2

```
>java examples.applets.chat.ChatRoom 7001 Chatter
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:7001
```

Note: You could start as many additional chat rooms as you want in this fashion; however, only one additional chat room is started in this example.

To start the chat applet, run appletviewer or a JDK 1.1-enabled browser on `Chat.html` from the `\voyager1.0.0\examples\applets\chat` directory.

Window 3

```
>appletviewer Chat.html
voyager(tm) 1.0.0, copyright objectspace 1997
address = 127.0.0.1:7000;127.0.0.1:1059
```

When the applet starts, enter your unique user name in the Name field. Enter the name of your local machine followed by :7001/Chatter, the address of the chat room, in the Server field. You are then connected to the specified chat room. Any text entered in the Speak field appears in the Transcript field of each chat applet, preceded by your unique user name.

Running the Applet from a Web Server

To run the chat applet from a Web server, first install the chat program on the server by adding the directories and files highlighted below.

```
\webpages
   \gglass
      \voyager1.0.0
         \bin
         \examples
            \applets
               \chat   Chat .html and all .class files
         \lib
```

Note: As before, this assumes your account is rooted in `\webpages\gglass`.

To start an applet server that provides network class loading and message routing for the chat applets, type the following command in a window on the Web server from the `\voyager1.0.0\examples\applets\chat` directory.

Window 1

```
>voyager 7000 -r \webpages
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:7000
```

where *webpages* is your Web server's root.

Then, start a chat room named `Chatter` on port 7001 by opening a second window and typing the following command from `\voyager1.0.0\examples\applets\chat` on the Web server.

Window 2

```
>java examples.applets.chat.ChatRoom 7001 Chatter
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:7001
```

To run a chat applet from your local machine, open a third window and invoke the appletviewer or JDK 1.1-enabled browser with the path name of the applet's `.html` file relative to the Web root.

Window 3

```
>appletviewer
http://pulsar/gglass/voyager1.0.0/examples/applets/chat/Chat.html
```

where *pulsar/gglass* is the name of your Web server and the path of your user account.

When the applet starts, enter your unique user name in the Name field and enter the name of your Web server followed by :7001/Chatter in the Server field. You are connected to the specified chat room. Any text entered in the Speak field appears in the Transcript field of each chat applet.

Application voyager1.0.0\examples\applets\chat\ChatRoom.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.applets.chat;

import java.util.Hashtable;
import java.util.Enumeration;
import COM.objectspace.voyager.*;

public class ChatRoom
  {
  Hashtable occupants = new Hashtable(); // name->applet mapping

  public static void main( String args[] )
    {
    try
      {
      int port = Integer.parseInt( args[ 0 ] ); // port for chat room
      String alias = args[ 1 ]; // alias for chat room
      Voyager.startup( port );

      // construct chat room
      VChatRoom room = new VChatRoom( "localhost/" + alias );
      room.liveForever(); // inhibit garbage collection
      }
    catch( VoyagerException exception )
      {
      System.err.println( exception );
      }
    }

  public void enter( String name, VChatApplet applet )
    throws VoyagerException
    {
```

```
    // someone has entered this chat room
    occupants.put( name, applet ); // add to map
    Enumeration enumeration = occupants.elements();

    while( enumeration.hasMoreElements() ) // notify all occupants
      ((VChatApplet) enumeration.nextElement()).entered( name, applet );
    }

  public void leave( String name ) throws VoyagerException
    {
    // someone has left this chat room
    Enumeration enumeration = occupants.elements();

    while( enumeration.hasMoreElements() ) // notify all occupants
      ((VChatApplet) enumeration.nextElement()).left( name );

    occupants.remove( name ); // remove from map
    }

  public Hashtable getOccupants()
    {
    return occupants;
    }
  }
```

Applet voyager1.0.0\examples\applets\chat\ChatApplet.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.applets.chat;

import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.util.Hashtable;
import java.util.Enumeration;
import COM.objectspace.voyager.*;

public class ChatApplet extends Applet
  {
  Label nameLabel = new Label( "Name" );
  TextField nameField = new TextField( "", 15 );
  Label roomLabel = new Label( "Server" );
  TextField roomField = new TextField( "", 25 );
  Label occupantsLabel = new Label( "Occupants" );
  List occupantsList = new List( 4, true ); // allow multiple
  Label textLabel = new Label( "Transcript" );
  TextArea textArea = new TextArea( 10, 35 );
  Label speakLabel = new Label( "Speak" );
  TextField speakField = new TextField( "", 35 );
  VChatRoom room; // remote reference to current chat room
  VChatApplet self; // remote reference to myself
  Hashtable occupants; // cache of room occupants

  public void init()
```

```
  {
  try
    {
    Voyager.startup( this ); // startup in applet mode
    self = (VChatApplet) VObject.forObject( this ); // reference to myself
    self.liveForever(); // don't generate heartbeats
    setLayout( new FlowLayout() );

    Panel panel1 = new Panel();
    panel1.add( nameLabel );
    panel1.add( nameField );
    add( panel1 );

    Panel panel2 = new Panel();
    panel2.add( roomLabel );
    panel2.add( roomField );
    roomField.addActionListener( new ActionListener()
    { public void actionPerformed( ActionEvent event ) { connect(); } } );
    add( panel2 );

    Panel panel3 = new Panel();
    panel3.add( occupantsLabel );
    panel3.add( occupantsList );
    add( panel3 );

    Panel panel4 = new Panel();
    panel4.add( textLabel );
    textArea.setEditable( false );
    panel4.add( textArea );
    add( panel4 );

    Panel panel5 = new Panel();
    panel5.add( speakLabel );
    panel5.add( speakField );
    speakField.addActionListener( new ActionListener()
    { public void actionPerformed( ActionEvent event ) { speak(); } } );
    add( panel5 );
    }
  catch( VoyagerException exception )
    {
    System.err.println( exception );
    }
  }

public void destroy()
  {
  leaveChatRoom();
  }

void connect()
  {
  String address = roomField.getText();

  try
    {
```

```
    leaveChatRoom(); // leave old chat room

    // connect to new room
    room = (VChatRoom) VObject.forObjectAt( address );
    addText( "enter " + address );
    occupants = room.getOccupants(); // cache room occupancy
    room.enter( nameField.getText(), self ); // enter room
    updateOccupants(); // display occupancy
    }
  catch( Exception exception )
    {
    addText( "could not connect to " + address );
    }
  }

void updateOccupants()
  {
  occupantsList.removeAll(); // clear list
  Enumeration enumeration = occupants.keys();

  while( enumeration.hasMoreElements() ) // build new list
    occupantsList.add( (String) enumeration.nextElement() );
  }

void addText( String text )
  {
  textArea.append( text + "\n" );
  }

void speak()
  {
  Enumeration iterator = occupants.elements();

  while( iterator.hasMoreElements() )
    {
    try
      {
      // get associated applet and send it the speak text
      VChatApplet applet = (VChatApplet) iterator.nextElement();
      applet.speak( nameField.getText(), speakField.getText() );
      }
    catch( Exception exception )
      {
      }
    }

  speakField.setText( "" ); // clear speak area
  }

void leaveChatRoom()
  {
  if( room != null )
    {
    try
      {
```

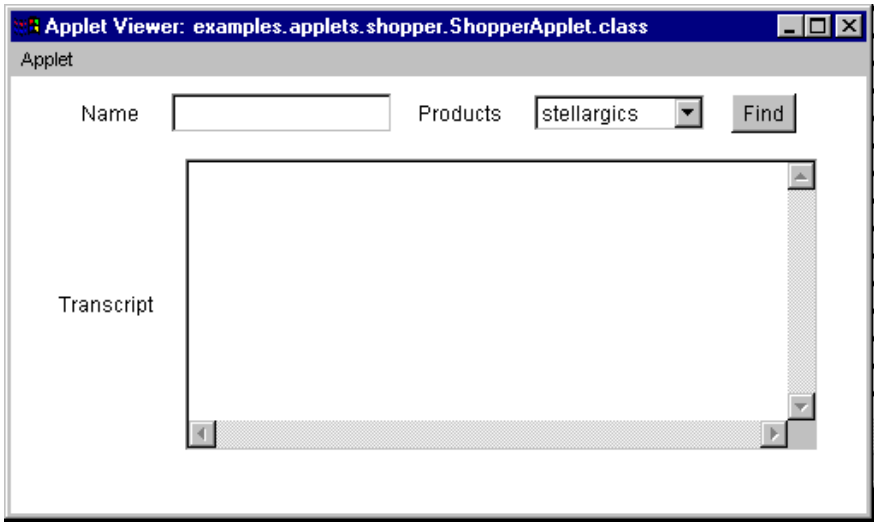
```
        room.leave( nameField.getText() ); // leave room
        room = null;
        }
      catch( VoyagerException exception )
        {
        System.err.println( exception );
        }
      }
    }

  public void speak( String name, String text )
    {
    // the occupant with the specified name said something
    addText( name + ": " + text );
    }

  public void entered( String name, VChatApplet applet )
    {
    // somebody entered the room
    occupants.put( name, applet );
    updateOccupants();
    }

  public void left( String name )
    {
    // somebody left the room
    occupants.remove( name );
    updateOccupants();
    }
  }
```

HTML voyager1.0.0\examples\applets\chat\Chat.html

```
<title>Chat Applet</title>
<hr>
<applet codebase="../../.." code="examples.applets.chat.ChatApplet.class"
archive="lib/voyager.jar" width=400 height=400>
<param name=Server value=7000>
</applet>
<hr>
```

Shopper Example — Applets and Agents

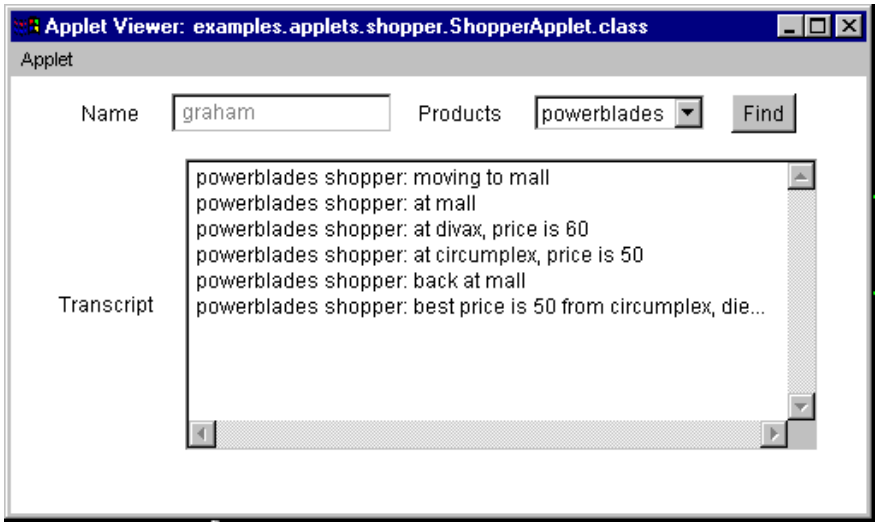
The third example is an applet version of the shopping example presented in Chapter 3, “Guided Tour.” When you start the shopper applet, a pull-down menu is populated with a list of all products available from stores at the mall.



To create and launch an agent to find the best price for a particular product, enter your name in the Name field, select the product from the Products pull-down menu, and select the Find button. The agent moves into the mall, gets a list of all stores, and then visits each store in turn. The agent continually sends its status to the shopper applet as long as the applet is open. The status displays in the Transcript field. After visiting each store, the agent moves back to the mall, sends the best price to the applet, and dies. If the applet is closed before the agent completes its task, the agent waits at the mall until an applet with your name is reopened; the agent then sends the results.

Note: You can launch new agents without waiting for the previous agent to complete its task.

Pictured below is an applet opened by Graham after he tells his agent to search for the best price on power blades and the agent finds two stores offering power blades, delivers the name of the store offering the best price, and dies.

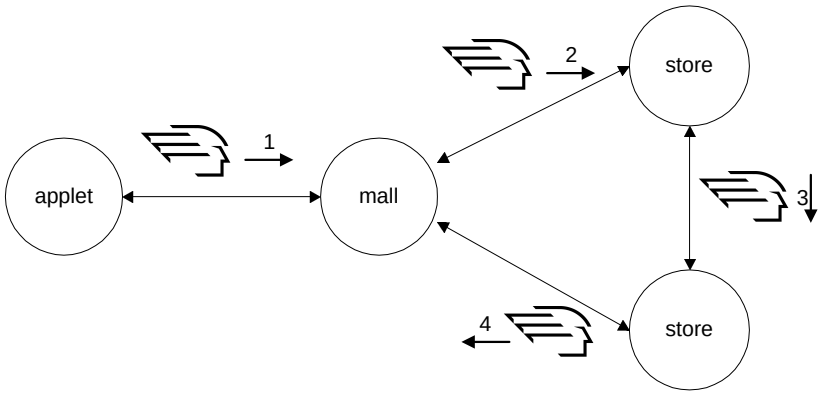


The architecture of the shopper application uses a Voyager program to double as the mall and the applet server. The mall program contains a single instance of `Mall` with the alias `Mall`. Each store program creates a single instance of `Store` and registers it with the mall. The mall also keeps a list of all products sold by each store.

When a shopper applet is started and the user's name is entered in the Name field, the applet registers itself with the mall by sending it `entered()`, enabling the mall to track all applets currently connected to it. When the Find button is selected, a `Shopper` agent is constructed with its user name, a virtual reference to the applet, and a virtual reference to the mall. The `find()` message causes the agent to move to the mall, get a list of all stores, and then move to each store in turn. The agent uses the virtual reference to the applet to send it regular status messages. If at any point a message cannot be delivered to the applet, the agent assumes the applet has been closed and stops sending status messages.

After the `Shopper` agent visits each store, it moves back to the mall and tries to locate the applet associated with its user name. If the applet is open, the agent sends the best price to the applet and dies. If the applet is not open, the agent waits at the mall until notified by the mall that the applet has become available.

The following diagram shows the path of the `Shopper` agent:



The shopper application is comprised of the following files:

- `Store.java` and `StoreImpl.java`

  The interface and implementation of the store. These programs define a store and are straightforward, containing no special applet-related code.

- `Mall.java` and `MallImpl.java`

  The interface and implementation of the mall. These programs define a mall. Because `MallImpl.java` is a program that services an applet, it uses the special version of `startup()` that takes a Web root as its second argument. `MallImpl.java` contains data structures to track all stores, products, applets, and waiting shopper agents.

- `Store1.java` and `Store2.java`

  The programs that create and stock each store. These simple programs each create, stock, and register a single instance of `StoreImpl`. Each program requires the port number of the mall as its command line argument.

- `Shopper.java`

  The program that defines the mobile shopping agent. It uses interfaces to communicate with a store and a mall so that its class closure does not include the code for the `StoreImpl.class` and `MallImpl.class`.

- `ShopperApplet.java`

  The program that defines the shopping applet. It uses the variation of `startup()` that takes an applet argument to initialize itself. It is a relatively simple applet that registers or deregisters with the mall and accepts callbacks from the shopping agents that it launches.

- `Shopper.html`

  The HTML code that describes the applet and its associated server. This program defines the applet codebase as three directory levels above itself because the applet is stored in a package relative to `\voyager1.0.0` on the Web server. For the shopper example, the applet's server application is assigned to port 8000.

Compiling the Programs

Use the following commands from the `\voyager1.0.0\examples\applets\shopper` directory to prepare and compile the shopper programs:

```
vcc -e java.lang.Object ShopperApplet
vcc MallImpl Shopper StoreImpl
javac *.java
```

Running the Applet from a Local Machine

To run the shopper applet on your local machine without a Web server, first open a window and start the mall server on port 8000.

Window 1

```
>java examples.applets.shopper.MallImpl 8000 \
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:8000
```

Then, in second and third windows, create the two stores.

Window 2

```
>java Store1 8000
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1064
Build divax
stock poloscope @ $50
stock powerblades @ $60
stock stellargics @ $65
```

Window 3

```
>java examples.applets.shopper.Store2 8000
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1066
Build circumplex
stock powerblades @ $50
stock trollkit @ $60
```

In a fourth window, start a shopper applet.

Window 4

```
>appletviewer Shopper.html
voyager(tm) 1.0.0, copyright objectspace 1997
address = 127.0.0.1:8000;127.0.0.1:1068
```

When the applet starts, enter a unique name in the Name field, select a desired product from the Products pull-down menu, and select the Find button. The shopper agent displays its status in the Transcript field.

Running the Applet from a Web Server

To run the shopper applet from a Web server, first install the shopper application on the server by adding the directories and files highlighted below.

```
\webpages
   \gglass
      \voyager1.0.0
         \bin
         \examples
            \applets
               \shopper  Shopper .html and all .class files
         \lib
```

To start the mall server that provides the mall, network class loading, and message routing for the shopper applets, open a window and type the following command from the `\voyager1.0.0\examples\applets\shopper` directory.

Window 1

```
>java examples.applets.shopper.MallImpl 8000 -r \webpages
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:8000
```

where *webpages* is your Web server's root.

Then, in second and third windows, create the two stores.

Window 2

```
>java examples.applets.shopper.Store1 8000
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1356
```

Window 3

```
>java examples.applets.shopper.Store2 8000
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:2156, root = \
```

To run a shopper applet from your local machine, open a fourth window and invoke the appletviewer or JDK 1.1-enabled browser with the path name of the applet's `.html` file relative to the Web root.

Window 4

```
>appletviewer
http://pulsar/gglass/voyager1.0.0/examples/applets/shopper/Shopper.html
```

where *pulsar/gglass* is the name of your Web server and the path of your user account.

Interface voyager1.0.0\examples\applets\shopper\Store.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.applets.shopper;

import java.util.Vector;

public interface Store
  {
  void stock( String product, int price );
  int getPrice( String product );
  void purchase( String product ) throws IllegalArgumentException;
  Vector getProducts();
  }
```

Class voyager1.0.0\examples\applets\shopper\StoreImpl.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.applets.shopper;

import java.util.Hashtable; // utilize a JDK Hashtable
import java.util.Vector;
import java.util.Enumeration;

public class StoreImpl implements Store
  {
  String name;
  Hashtable products = new Hashtable(); // contains product->price pairs

  public StoreImpl( String name )
    {
    this.name = name;
    System.out.println( "Build " + this );
    }

  public String toString()
    {
    return name;
    }

  public void stock( String product, int price )
    {
    System.out.println( "stock " + product + " @ $" + price );
    products.put( product, new Integer( price ) ); // add product to stock
    }

  public int getPrice( String product )
    {
    Integer integer = (Integer) products.get( product ); // get price
    return integer == null ? 0 : integer.intValue(); // zero if not in stock
    }

  public void purchase( String product ) throws IllegalArgumentException
    {
    int price = getPrice( product );

    if( price == 0 )
      throw new IllegalArgumentException("product " + product + " not found");

    System.out.println( "purchase " + product + " @ $" + price );
    }

  public Vector getProducts()
    {
    Vector vector = new Vector();
    Enumeration enumeration = products.keys();

    while( enumeration.hasMoreElements() )
      vector.addElement( enumeration.nextElement() );
```

```
    return vector;
    }
  }
```

Interface voyager1.0.0\examples\applets\shopper\Mall.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.applets.shopper;

import java.util.Vector;

public interface Mall
  {
  void register( Store store );
  Vector getProducts();
  void enter( String name, VShopperApplet applet );
  void leave( String name );
  VShopperApplet getShopperApplet( String name );
  void wait( Shopper shopper );
  Vector getStores();
  }
```

Application voyager1.0.0\examples\applets\shopper\MallImpl.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.applets.shopper;

import java.util.Vector;
import java.util.Hashtable;
import COM.objectspace.voyager.*;

public class MallImpl implements Mall
  {
  Vector stores = new Vector(); // list of all stores
  Vector products = new Vector(); // list of all products
  Hashtable applets = new Hashtable(); // all registered applets
  Vector shoppers = new Vector(); // shoppers waiting for applets

  public static void main( String args[] )
    {
    try
      {
      int port = Integer.parseInt( args[ 0 ] ); // port to serve
      String root = args[ 1 ]; // root of server's visible directory tree
      Voyager.startup( port, root ); // serve applets
      VMallImpl mall = new VMallImpl( "localhost/Mall" );
      mall.liveForever(); // inhibit garbage collection
      }
    catch( VoyagerException exception )
      {
      System.err.println( exception );
      }
```

```
  }

public void register( Store store )
  {
  stores.addElement( store ); // add to list of stores
  Vector list = store.getProducts(); // get list of products

  // add products to product list
  for( int i = 0; i < list.size(); i++ )
    {
    String product = (String) list.elementAt( i );

    if( !products.contains( product ) )
      products.addElement( product );
    }
  }

public Vector getProducts()
  {
  return products;
  }

public void enter( String name, VShopperApplet applet )
  {
  // an applet has registered with the mall
  applets.put( name, applet );

  synchronized( shoppers )
    {
    int i = 0;

    // alert all shoppers that were waiting for this applet
    while( i < shoppers.size() )
      {
      Shopper shopper = (Shopper) shoppers.elementAt( i );

      if( shopper.getOwner().equals( name ) )
        {
        try
          {
          shopper.setShopperApplet( applet );
          shopper.report(); // transmit report and then die
          shoppers.removeElementAt( i ); // update shopper list
          }
        catch( VoyagerException exception )
          {
          shoppers.removeElement( shopper ); // remove unreachable element
          }
        }
      else
        {
        i++;
        }
      }
    }
```

```
  }

public void leave( String name )
  {
  // an applet has left the mall
  applets.remove( name );
  }

public VShopperApplet getShopperApplet( String name )
  {
  return (VShopperApplet) applets.get( name );
  }

public void wait( Shopper shopper )
  {
  shoppers.addElement( shopper ); // add to shopper list
  }

public Vector getStores()
  {
  return stores;
  }
}
```

Application voyager1.0.0\examples\applets\shopper\Store1.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.applets.shopper;

import COM.objectspace.voyager.*;

public class Store1
  {
  public static void main( String args[] )
    {
    try
      {
      int port = Integer.parseInt( args[ 0 ] ); // port to serve

      // create and stock an immortal store
      VStoreImpl store = new VStoreImpl( "divax", "localhost" );
      store.liveForever();
      store.stock( "poloscope", 50 );
      store.stock( "powerblades", 60 );
      store.stock( "stellargics", 65 );

      // contact mall and register store
      VMallImpl mall =
        (VMallImpl) VObject.forObjectAt( "localhost:" + port + "/Mall" );
      mall.register( store );
      }
    catch( VoyagerException exception )
      {
      System.err.println( exception );
      }
    }
  }
```

Application voyager1.0.0\examples\applets\shopper\Store2.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.applets.shopper;

import COM.objectspace.voyager.*;

public class Store2
  {
  public static void main( String args[] )
    {
    try
      {
      int port = Integer.parseInt( args[ 0 ] ); // port to serve

      // create and stock an immortal store
      VStoreImpl store = new VStoreImpl( "circumplex", "localhost" );
      store.liveForever();
      store.stock( "powerblades", 50 );
      store.stock( "trollkit", 60 );

      // contact mall and register store
      VMallImpl mall =
        (VMallImpl) VObject.forObjectAt( "localhost:" + port + "/Mall" );
      mall.register( store );
      }
    catch( VoyagerException exception )
      {
      System.err.println( exception );
      }
    }
  }
```

Class voyager1.0.0\examples\applets\shopper\Shopper.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.applets.shopper;

import java.util.Vector;
import COM.objectspace.voyager.*;

public class Shopper extends Agent
  {
  VShopperApplet applet; // remote reference to my applet
  VMallImpl mall; // remote reference to mall
  String owner; // name associated with applet
  String product; // product to search for
  Vector itinerary; // list of stores to visit
  int index; // index into itinerary of current store
  VStoreImpl bestStore = null; // store with cheapest price
  int bestPrice = Integer.MAX_VALUE; // current best price

  public Shopper( String owner, VShopperApplet applet, VMallImpl mall )
    {
    this.owner = owner;
    this.applet = applet;
    this.mall = mall;
    }

  public void find( String product ) throws VoyagerException
    {
    this.product = product;
    status( "moving to mall" );
    moveTo( mall, "atMall" );
    }

  public void atMall( Mall mall ) throws VoyagerException
    {
    status( "at mall" );
    itinerary = mall.getStores(); // get list of stores to visit
    moveTo( getStore(), "atStore" ); // move to first store
    }

  public void atStore( Store store ) throws VoyagerException
    {
    int price = store.getPrice( product ); // get price

    if( price == 0 )
      {
      status( "at " + store + ", " + product + " not sold" );
      }
    else
      {
      status( "at " + store + ", price is " + price );

      if( price < bestPrice )
        {
```

```
      bestStore = getStore(); // this is the best store so far
      bestPrice = price;
      }
    }

  // sleep to simulate activity
  try{ Thread.sleep( 4000 ); } catch( InterruptedException exception ) {}

  // if not finished, move to next store, else move back to mall
  if( ++index < itinerary.size() )
    moveTo( getStore(), "atStore" );
  else
    moveTo( mall, "finished" );
  }

VStoreImpl getStore()
  {
  return (VStoreImpl) itinerary.elementAt( index );
  }

public void finished( Mall mall ) throws VoyagerException
  {
  status( "back at mall" );
  applet = mall.getShopperApplet( owner );

  // if applet is available, transmit best price and then die, else wait
  if( applet != null )
    report();
  else
    mall.wait( this );
  }

public void setShopperApplet( VShopperApplet applet )
  {
  this.applet = applet;
  }

public void report() throws VoyagerException
  {
  status("best price is " + bestPrice + " from " + bestStore + ", die...");
  dieNow();
  }

public String getOwner()
  {
  return owner;
  }

void status( String string ) throws VoyagerException
  {
  String message = product + " shopper: " + string;

  try
    {
    System.out.println( message ); // print to standard output
```

```
      if( applet != null )
        applet.addText( message ); // transmit to applet
      }
    catch( Exception exception )
      {
      applet = null; // could not contact applet, don't bother next time
      }
    }
  }
```

Applet voyager1.0.0\examples\applets\shopper\ShopperApplet.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.applets.shopper;

import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.util.Vector;
import COM.objectspace.voyager.*;

public class ShopperApplet extends Applet
  {
  Label nameLabel = new Label( "Name" );
  TextField nameField = new TextField( "", 15 );
  Label productsLabel = new Label( "Products" );
  Choice productsChoice = new Choice();
  Button findButton = new Button( "Find" );
  Label textLabel = new Label( "Transcript" );
  TextArea textArea = new TextArea( 10, 50 );
  String user; // user name
  VMallImpl mall;
  VShopperApplet self;

  public void init()
    {
    try
      {
      Voyager.startup( this ); // startup in applet mode
      self = (VShopperApplet) VObject.forObject( this );
      self.liveForever();
      mall =
       (VMallImpl) VObject.forObjectAt( Voyager.getServerAddress() + "/Mall"
);
      setLayout( new FlowLayout() );

      Panel panel1 = new Panel();
      panel1.add( nameLabel );
      panel1.add( nameField );
      nameField.addActionListener( new ActionListener()
      { public void actionPerformed(ActionEvent event) { updateName(); } } );
      add( panel1 );
```

```
    Panel panel2 = new Panel();
    panel2.add( productsLabel );
    Vector products = mall.getProducts();

    for( int i = 0; i < products.size(); i++ )
      productsChoice.add( (String) products.elementAt( i ) );

    panel2.add( productsChoice );
    add( panel2 );

    Panel panel3 = new Panel();
    panel3.add( findButton );
    findButton.addActionListener( new ActionListener()
    { public void actionPerformed( ActionEvent event ) { find(); } } );
    add( panel3 );

    Panel panel4 = new Panel();
    panel4.add( textLabel );
    textArea.setEditable( false );
    panel4.add( textArea );
    add( panel4 );

    setSize( 500, 250 );
    }
  catch( VoyagerException exception )
    {
    System.out.println( exception );
    }
  }

void updateName()
  {
  try
    {
    user = nameField.getText();
    mall.enter( user, self );
    nameField.setEnabled( false ); // freeze
    }
  catch( VoyagerRuntimeException exception )
    {
    System.err.println( exception );
    }
  }

public void destroy()
  {
  try
    {
    mall.leave( user );
    }
  catch( VoyagerRuntimeException exception )
    {
    System.err.println( exception );
    }
  }
```

```
  public void addText( String text )
    {
    textArea.append( text + "\n" );
    }

  void find()
    {
    try
      {
      if( !nameField.getText().equals( user ) )
        updateName(); // update current name

      VShopper shopper = new VShopper( user, self, mall, "localhost" );
      shopper.find( productsChoice.getSelectedItem() );
      }
    catch( VoyagerException exception )
      {
      System.err.println( exception );
      }
    }
  }
```

HTML voyager1.0.0\examples\applets\shopper\Shopper.html

```
<title>Shopper Applet</title>
<hr>
<applet codebase="../../.."
code="examples.applets.shopper.ShopperApplet.class"
archive="lib/voyager.jar" width=500 height=250>
<param name=Server value=8000>
</applet>
<hr>
```

11

Security

ObjectSpace Voyager™ (Voyager) Core Technology includes support for the standard Java™ security manager system. You have the option of installing a security manager in every Voyager program. Once a security manager has been installed, it cannot be uninstalled or replaced. Each time an object attempts to execute an operation that could compromise security, the Java run-time machinery checks with the program's security manager to determine whether the operation is permitted. If the program has no security manager or if the security manager permits the operation, Voyager proceeds as normal. If the operation is disallowed, a run-time `SecurityException` is thrown.

Java applets are automatically initialized with a very restrictive security manager called `AppletSecurityManager`. Voyager applets must therefore abide by these settings.

Java applications have no security manager by default, which means that objects have free rein to perform any kind of operation.

Voyager includes a security manager called `VoyagerSecurityManager`, which you can install at the start of a program to restrict operations. The Voyager security manager considers instances of a class loaded via the program's `CLASSPATH` as native objects and instances of a class loaded across the network from another program as foreign objects. The Voyager security manager allows native objects to perform any operation but restricts foreign objects on a per-operation basis.

Users can modify or extend the Voyager security manager behavior by extending the `VoyagerSecurityManager` class.

The table below lists the operations that can be performed by an object in an applet, a native object, and a foreign object.

Operation	Object in Applet	Native Object	Foreign Object
Accept connections from any host	server only	yes	yes
Connect to any host	server only	yes	yes
Listen on any port	no	yes	yes
Perform multicast operations	no	yes	yes
Set socket factories	no	yes	no
Manipulate threads	yes	yes	no
Manipulate thread groups	yes	yes	no
Execute a process	no	yes	no
Exit the virtual machine	no	yes	no
Access AWT event queue	yes	yes	yes
Access the system clipboard	no	yes	no
Create windows	yes	yes	yes
Create class loader	no	yes	no
Delete files	no	yes	no
Read files, excluding socket file descriptors	no	yes	no
Write files, excluding socket file descriptors	no	yes	no
Access security APIs	no	yes	no
Shut down Voyager	yes	yes	no
Link to a dynamic library	no	yes	no
Access private/protected data and methods	no	yes	no
Access packages	no	yes	yes
Define classes in packages	no	yes	yes
Print	no	yes	no
Manipulate properties	limited	yes	yes
Manipulate databases	n/a	yes	no
Set multihome addresses	yes	yes	no

There are two ways to install a Voyager security manager. To start a Voyager server that has a Voyager security manager, execute `voyager` with the `-s` (secure) option. To install a Voyager security manager into a Voyager program, use the `Voyager.installVoyagerSecurityManager()` method. To install a custom security manager, use the `Voyager.installVoyagerSecurityManager()` method with an instance of `VoyagerSecurityManager` or one of its subclasses.

The following example demonstrates how a Voyager security manager can be used to restrict operations by foreign objects.

The `voyager1.0.0\examples\security\native` directory contains a `Visitor` class and a `Native.java` program that installs a Voyager security manager. Because `Visitor` is defined in the same package as `Native.java`, the Voyager security manager of `Native.java` considers `Visitor` a native class.

The `voyager1.0.0\examples\security\foreign` directory contains a `Foreign.java` program that also installs a Voyager security manager. Because the `Visitor` class is not accessible to `Foreign.java` via the `CLASSPATH`, the Voyager security manager of `Foreign.java` considers `Visitor` a foreign class.

When `Security1.java` moves a `Visitor` into `Foreign.java` and tells `Visitor` to create a thread, an exception is thrown—the Voyager security manager does not allow foreign objects to manipulate threads.

From the `\voyager1.0.0\examples\security\native` directory, use the following commands to prepare and compile the `Visitor.java` and `Security1.java` example programs:

```
vcc Visitor
javac Visitor.java VVisitor.java Security1.java
```

Then, from the `\voyager1.0.0\examples\security\foreign` directory, use the following command to compile `Foreign.java`:

```
javac Foreign.java
```

From the `\voyager1.0.0\examples\security\foreign` directory, run `Foreign.class` in one window. Then, from the `\voyager1.0.0\examples\native` directory, run `Native.class` and `Security1.class` in second and third windows.

Window 1

```
>java Foreign
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:7000
>
```

Window 2

```
>java Native
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:8000
new thread is counting...
0
1
2
3
>
```

Window 3

```
>java Security1
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1226
java.lang.SecurityException: foriegn objects/messages may not manipulate a
thread group
        thrown remotely from 208.6.239.200:7000
>
```

Application voyager1.0.0\examples\security\native\Security1.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

import COM.objectspace.voyager.*;

public class Security1
  {
  public static void main( String args[] )
    {
    try
      {
      VVisitor visitor = new VVisitor( "localhost" );
      visitor.moveTo( "localhost:8000" );
      visitor.loopUsingThread();
      visitor.moveTo( "localhost:7000" );
      visitor.loopUsingThread();
      Voyager.shutdown();
      }
    catch( Exception exception )
      {
      System.err.println( exception );
      }
    }
  }
```

Class voyager1.0.0\examples\security\native\Visitor.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

import COM.objectspace.voyager.*;

public class Visitor implements java.io.Serializable, Runnable
  {
  public synchronized void loopUsingThread()
    {
    Thread thread = new Thread( this );
    thread.start();

    try
      {
      wait(); // wait for thread to complete
      }
    catch( InterruptedException exception )
      {
      }
    }

  public synchronized void run()
    {
    System.out.println( "new thread is counting..." );

    for( int i = 0; i < 4; i++ )
      System.out.println( i );

    notify(); // notify that thread has completed
    }
  }
```

Application voyager1.0.0\examples\security\native\Native.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

import COM.objectspace.voyager.*;

public class Native
  {
  public static void main( String args[] )
    {
    try
      {
      Voyager.startup( 8000 );
      Voyager.installVoyagerSecurityManager();
      }
    catch( VoyagerException exception )
      {
      System.err.println( exception );
      }
    }
  }
```

Application voyager1.0.0\examples\security\foreign\Foreign.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

import COM.objectspace.voyager.*;

public class Foreign
  {
  public static void main( String args[] )
    {
    try
      {
      Voyager.startup( 7000 );
      Voyager.installVoyagerSecurityManager();
      }
    catch( VoyagerException exception )
      {
      System.err.println( exception );
      }
    }
  }
```

12

Customizing Voyager Applications

This chapter describes several ways to customize applications developed with ObjectSpace Voyager™ Core Technology (Voyager). Specifically, you can use custom sockets rather than the default sockets, you can extend class loading behavior, and you can configure your computer with more than one domain name.

Custom Sockets

By default, Voyager sockets are JDK system default sockets with system default socket implementations. However, developers can instruct Voyager to use custom sockets instead for functionality like firewall tunneling, SSL encryption, and compression.

Socket Factories

Socket factories are objects that can create sockets. Voyager socket factories must implement the `VoyagerSocketFactory` interface, just as Voyager sockets must implement the `VoyagerSocket` interface. If a connection needs to be opened to a Voyager program at an IP address for which a custom socket factory has been specified, then Voyager uses a custom socket factory to create the socket. (Typically, both communicating Voyager programs need to use the same kind of socket.) If no custom socket factory has been specified for connections to programs at the given address, Voyager uses the default socket factory to create the socket. Voyager also uses the default socket factory to create the server socket for the Voyager program in which it is running.

Customizing the Default Socket Factory

The default socket factory can be replaced if a custom server socket is needed or if every client connection must be made with a custom socket. The default socket factory can only be set at startup because it is used to create the initial server socket. Use one of the Voyager `startup()` methods that accept a `VoyagerSocketFactory` to set up the default socket factory.

Adding Custom Socket Factories

Sometimes custom sockets are needed only by a limited set of hosts thus should not be used by default or for server socket functionality. To instruct Voyager to use a different kind of socket when connecting to programs at a specific host, invoke Voyager's `addSocketFactory()` method. This method takes a `String` containing the domain name or IP address of another machine and an object that implements the `VoyagerSocketFactory` interface, which is used to create sockets for connections opened to any Voyager program running on that remote machine.

There are two requirements for creating a custom socket factory capable of producing custom sockets. First, the socket factory must implement the `VoyagerSocketFactory` interface. This interface provides one method for getting a client socket and one method for getting a server socket. Second, the custom socket implementations that the custom socket factory creates must implement the `VoyagerSocket` or `VoyagerServerSocket` interface, as appropriate. Once these two requirements are met, developers are free to implement the sockets in any way. In particular, custom sockets need not extend `java.net.Socket`.

Custom Class Loading

Voyager includes a class loader capable of loading classes from the `CLASSPATH`, from the `CODEBASE` (for Applets), and across the network. Some programs might need a class loader with additional capabilities. For instance, a program might need all Voyager servers to load classes from a centralized database of class files. To support this, Voyager provides the interface `VoyagerClassLoader`. By implementing this simple interface and invoking Voyager's `registerClassLoader()` method with an instance of this implementation, developers can easily extend class loading behavior.

Class loaders are used in two scenarios. One scenario is when the `loadClass()` method is invoked on the class loader in an attempt to load a class with a given name into the local virtual machine. Before trying any of the custom class loaders, Voyager tries the standard class loading strategy, that is, loading from the `CLASSPATH`, the network, or the `CODEBASE`, depending on the situation. If the standard class loading strategy completes but the class file is still not found, Voyager tries each of the custom class loaders in the order in which they are registered with the Voyager program.

The other scenario in which class loaders are used is when Voyager asks for the bytes that define a class file. This can happen if Voyager has been asked for these bytes by a remote Voyager program, such as when a Voyager program serves up class files for the remote Voyager network class loader. The custom class loader should be able to return the bytes of a class file for any class it can load.

Voyager on Multihomed Computers

Multihomed computers are machines that are configured with more than one domain name. For a Voyager program to run on such a machine, Voyager must be aware of all possible names for the machine. By default, Voyager is aware of the host name returned by `java.net.InetAddress.getHostName()` and all IP addresses associated with the host name. To make Voyager aware of another name by which the machine is known, invoke the `addMultiHome()` method with a string containing the other name. Voyager is then able to function with the name and all IP numbers associated with the name.

Part 3

ObjectSpace Voyager Services

1 3

Introduction

Part 3 is comprised of four chapters that describe the various services included with the ObjectSpace Voyager™ Core Technology (Voyager).

- Read this chapter for a summary of the Part 3 chapters.
- Read Chapter 14, “Database-Independent Persistence,” for an explanation of how to persist, flush, and load objects using Voyager’s integrated database support. Some of the main Voyager classes introduced are `Db`, `DbInfo`, and `VoyagerDb`.
- Read Chapter 15, “Space: Scalable Group Communication,” for a description of Voyager’s innovative Space™ architecture for large-scale distributed computing, including one-way multicast, distributed events, JavaBeans™ integration, and state-of-the-art publish/subscribe. Some of the main Voyager classes introduced are `Subspace`, `OneWayMulticast`, and `SubspaceEvent`.
- Read Chapter 16, “Federated Directory Service,” for an explanation of how to use the Voyager directory service to locate objects and agents in a distributed system.

14

Database-Independent Persistence

Most programs require the ability to create an object that has a long life span. Because a program is volatile and will lose objects from its memory if the host crashes, databases are usually employed for saving objects on a secondary, nonvolatile medium such as a hard disk.

Several kinds of databases can be used to store an object, including relational databases and object-oriented databases. Each kind of database generally requires a different approach to storing an object. For example, an object database vendor might require you to post-process your Java™ classes to store a Java object; whereas, a relational database vendor might require you to decompose your object into smaller pieces that can be stored into relational tables. Ideally, this level of detail should be hidden from the programmer so that the act of saving an object to the database is simple and independent of the storage device.

ObjectSpace Voyager™ Core Technology (Voyager) supports database-independent, distributed-object persistence. Any database adapter that implements `Db` can be used by Voyager for storing and retrieving objects. Voyager includes an object storage system, `VoyagerDb`, that implements `Db` and uses the Java serialization mechanism to persist any serializable object without modification. Implementations of `Db` can be created to support most of the popular database systems. All examples in this chapter use `VoyagerDb` for object storage.

Concepts

This section summarizes the concepts underlying Voyager's support for persistence. An example of each concept is presented later in this chapter.

Assigning a Database

When a Voyager program is started, it can be assigned to an optional database using `setDb()`. A program that is assigned to a database is said to *support persistence*. In Voyager, two programs should not share the same database. Once a Voyager server is assigned to a database, the server should not be reassigned to a different database.

Saving an Object

To make an object persistent and immediately save it in the database, send the object `saveNow()` via a virtual reference. The object is copied into its program's database, overwriting the previous version, if present. If the object resides in a program that does not support persistence, a `DbException` is thrown. If the program is shut down and then restarted, the persistent objects are initially left in the database and do not consume any memory.

Loading an Object

Any attempt to communicate with a persistent object not currently in memory causes the object to be automatically loaded from the database. This feature is called *autoloading*. Objects can also be autoloaded at program startup by using `setAutoload()`. In either case, an object can listen for the `ObjectEvent.LOADED` event to perform an action upon reload.

Saving a Virtual Reference

Saving a virtual reference uses little storage space because saving a virtual reference does not cause a save of the object it references. Similarly, autoloading a virtual reference does not cause an autoload of the object it references.

Distributing Persistent Objects

A group of persistent objects can be distributed in a network, each with virtual references to other objects in the group. Each object is persisted in its own program's database, which might be a relational database in one case and an object database in another. Object autoloading enables an individual object to be flushed and its program to be shutdown and restarted without impacting other objects in the network.

Moving a Persistent Object

If a persistent object is moved, it is removed from the source program's database and saved in the destination program's database. The object leaves behind a persistent forwarder. Mobile persistence works even if the source program has a different type of database than the destination program.

Removing a Persistent Object

When a persistent object dies, the copy of the object is automatically removed from the program database. To force an object to die, send it `dieNow()`.

Garbage-Collecting a Persistent Object

A persistent object's program contains enough information to garbage-collect the object even when it is not in memory. If a persistent object is garbage-collected, it is reloaded and then sent `dieNow()`.

Flushing a Persistent Object

To conserve memory, you can use one of the `flush()` family of methods to flush a persistent object from memory to a database. If the object is an agent, it can listen for an `ObjectEvent.FLUSHING` event to perform an action immediately prior to being flushed. As before, any subsequent attempt to communicate with a flushed persistent object causes the object to be reloaded from the database.

Persisting a Class

By default, a Voyager program that supports persistence automatically persists Java classes that are loaded into a program across a network. In this way, the Java classes need not be reloaded when the program is restarted. This default setting ensures that if a foreign object is saved in a program's database and the program is then restarted, the foreign object's class definition is immediately available. Use `Voyager.setPersistentNetworkClasses()` to change this setting.

Modifying the Persistence of an Object

To make an object persistent and immediately save the object in its database, call `setPersistent(true)` on the object's virtual reference. To remove an object from its database and make it nonpersistent, call `setPersistent(false)`.

Starting a Persistent Voyager Server

To start a Voyager server and associate it with a `VoyagerDb` database, use the `-d` option of the `voyager` command line utility. The `-d` option instructs Voyager either to use the named database file for its persistent storage or to create a file if one does not exist. The `-c` option clears the database if it already exists.

Using the following command:

```
voyager 8000 -cd MyStorage.db
```

is equivalent to explicitly using `setDb()` as shown:

```
import COM.objectspace.voyager.*;
import COM.objectspace.voyager.db.*;

public class Example
  {
  public static void main( String args[] )
    {
    try
      {
      // attach to "MyStorage.db", replace if already present
      Voyager.setDb( new VoyagerDb( "MyStorage.db", true ) );
      // program code goes here
      Voyager.shutdown();
      }
    catch( VoyagerException exception )
      {
      System.err.println(exception);
      }
    }
  }
```

You can name a database file arbitrarily. The `.db` extension is suggested but not required.

A common use case is to start an empty, persistent Voyager server on a particular port using the `-cd MyStorage.db` command, create and save objects in the server, and then shut down the server. Later, the persistent server is restarted *on the same port* using `-d MyStorage.db`, and its objects are autoloaded when they receive messages. If you do not restart the server on the same port, virtual references cannot find and autoload the persistent objects.

There is currently no way to use the `voyager` command line utility to start a persistent Voyager server that is attached to any other kind of database.

Saving, Loading, and Deleting a Persistent Object

The following two-phase example shows how to save, load, and delete a persistent object.

In the first phase, a Voyager persistent server is started on port 8000 and assigned to the database file `8000.db`. Then, the `Db1A.java` example program creates a persistent vector with alias `Alphas` in the server. Finally, the persistent server is shut down by pressing Ctrl+C.

In the second phase, the persistent server is restarted and assigned to the database created in the first phase. Then, the `Db1B.java` example program connects to the persistent vector, displays its contents, and tells the vector to die, removing the vector from the database.

Because the default database `VoyagerDb` uses Java serialization to save an object, any serializable object can be saved without modifying its class in any way. Many JDK classes, including `Vector`, `Hashtable`, `String`, and `Integer`, are serializable. When making your own classes serializable, be sure to use the `transient` keyword when appropriate to prevent unwanted objects from being part of an object's serialized form.

Phase 1

From the `\voyager1.0.0\examples\db` directory, use the following commands to prepare and compile `Db1A.java` and `Db1B.java`:

```
vcc java.util.Vector
javac VVector.java Db1A.java Db1B.java
```

Start a persistent server on port 8000 in one window, and then run `Db1A.class` in the second window.

Window 1

```
>voyager 8000 -cd 8000.db
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:8000
database = 0 objects, 0 classes
```

Window 2

```
>java examples.db.Db1A
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1108
saved [2, hi, 3.14]
>
```

When `Db1A.java` finishes, terminate the server by pressing Ctrl+C in the first window. Phase 1 is now complete.

Phase 2

To see the size of the database file, use the `dir` command on Windows platforms and the `ls -l` command on UNIX platforms. Restart the persistent server in Window 1 and run `Db1B.class` in Window 2.

Window 1

```
>voyager 8000 -cd 8000.db
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:8000
database = 0 objects, 0 classes
>dir 8000.db
8000     DB          2,056  09-03-97 10:33a 8000.db
>voyager 8000 -d 8000.db
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:8000
database = 1 object, 0 classes
```

Window 2

```
>java examples.db.Db1A
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1108
saved [2, hi, 3.14]
>java examples.db.Db1B
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1116
loaded [2, hi, 3.14]
>
```

Application voyager1.0.0\examples\db\Db1A.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.db;

import COM.objectspace.voyager.*;
import VVector;

public class Db1A
  {
  public static void main( String args[] )
    {
    try
      {
      // create vector in server 8000
      VVector vector = new VVector( "localhost:8000/Alphas" );
      vector.addElement( new Integer( 2 ) );
      vector.addElement( "hi" );
```

```
      vector.addElement( new Float( 3.14 ) );
      vector.liveForever(); // do not garbage collect
      vector.saveNow(); // save copy to database of server 8000
      System.out.println( "saved " + vector );
      Voyager.shutdown();
      }
    catch( VoyagerException exception )
      {
      System.err.println(exception);
      }
    }
  }
```

Application voyager1.0.0\examples\db\Db1B.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.db;

import COM.objectspace.voyager.*;
import VVector;

public class Db1B
  {
  public static void main( String args[] )
    {
    try
      {
      // connect to persistent vector in server 8000
      VVector vector = (VVector)
VObject.forObjectAt("localhost:8000/Alphas");
      System.out.println( "loaded " + vector ); // display
      vector.dieNow(); // kill vector, remove from database
      Voyager.shutdown();
      }
    catch( VoyagerException exception )
      {
      System.err.println(exception);
      }
    }
  }
```

Distributed Persistence

An object can hold virtual references to persistent objects anywhere in a network. In a typical distributed application, objects often move back and forth from storage. Voyager's object autoloading mechanism hides from a programmer the details of whether a persistent object is currently in memory.

The following example demonstrates the power and simplicity of distributed persistence and autoloading.

A `Theater` class is created that represents a movie theater. The class is made serializable so that it can be stored in `VoyagerDb`, the default Voyager database. The `Db2A.java` example program creates two persistent theaters, one in a persistent server on port 7000, the other in a persistent server on port 8000. `Db2A.java` also creates a persistent vector with alias `Theaters` in server 8000, fills the vector with virtual references to the persistent theaters, and then saves the vector. The servers are then terminated.

When the servers are restarted, the persistent objects still reside in the database. When the `Db2B.java` example program connects to the persistent vector with alias `Theaters`, the vector is autoloaded into server 8000. Because autoloading a virtual reference does not cause its associated object to be autoloaded, autoloading the vector of virtual references to theaters does not cause the persistent theaters to be autoloaded. However, when `Db2B.java` begins printing each element of the array, the `toString()` message that is implicitly sent to each persistent theater via its virtual reference causes autoloading.

Distributed autoloading, combined with flushing (described on page 254), provides a powerful mechanism for building large-scale distributed systems that use memory conservatively.

From the `\voyager1.0.0\examples\db` directory, use the following commands to prepare and compile `Db2A.java` and `Db2B.java`:

```
vcc Theater
javac VTheater.java Theater.java Db2A.java Db2B.java
```

Start a persistent server on each of ports 7000 and 8000 in two separate windows. Then run `Db2A.class` in a third window. When `Db2A.class` finishes, terminate the servers by pressing Ctrl+C in Windows 1 and 2.

Window 1

```
>voyager 7000 -cd 7000.db
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:7000
database = 0 objects, 0 classes
construct Theater( monoplex )
>
```

Window 2

```
>voyager 8000 -cd 8000.db
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:8000
database = 0 objects, 0 classes
construct Theater( metropolis )
>
```

Window 3

```
>java examples.db.Db2A
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1120
theaters = [Theater( monoplex ), Theater( metropolis )]
>
```

Now restart the persistent servers in the first two windows and run `Db2B.class` in the third window. When `Db2B.class` finishes, terminate the servers again by pressing Ctrl+C in Windows 1 and 2.

Window 1

```
>voyager 7000 -cd 7000.db
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:7000
database = 0 objects, 0 classes
construct Theater( monoplex )
>voyager 7000 -d 7000.db
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:7000
database = 1 object, 0 classes
>
```

Window 2

```
>voyager 8000 -cd 8000.db
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:8000
database = 0 objects, 0 classes
construct Theater( metropolis )
>voyager 8000 -d 8000.db
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:8000
database = 2 objects, 0 classes
>
```

Window 3

```
>java examples.db.Db2A
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1120
theaters = [Theater( monoplex ), Theater( metropolis )]
>
>java examples.db.Db2B
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1124
theater 0 = Theater( monoplex )
theater 1 = Theater( metropolis )
>
```

Class voyager1.0.0\examples\db\Theater.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.db;

import java.util.Vector;

public class Theater implements java.io.Serializable
  {
  String name;
  Vector movies = new Vector();

  public Theater( String name )
    {
    this.name = name;
    System.out.println( "construct " + this );
    }

  public String toString()
    {
    return "Theater( " + name + " )";
    }

  public void addMovie( String movie )
    {
    movies.addElement( movie );
    }

  public Vector getMovies()
    {
    return movies;
    }
  }
```

Application voyager1.0.0\examples\db\Db2A.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.db;

import COM.objectspace.voyager.*;
import VVector;

public class Db2A
  {
  public static void main( String args[] )
    {
    try
      {
      // create vector in server 8000
      VTheater theater1 = new VTheater( "monoplex", "localhost:7000" );
      theater1.addMovie( "the fifth element" );
      theater1.addMovie( "men in black" );
      theater1.liveForever();
      theater1.saveNow();

      VTheater theater2 = new VTheater( "metropolis", "localhost:8000" );
      theater2.addMovie( "independence day" );
      theater2.addMovie( "ransom" );
      theater2.liveForever();
      theater2.saveNow();

      VVector theaters = new VVector( "localhost:8000/Theaters" );
      theaters.liveForever();
      theaters.addElement( theater1 );
      theaters.addElement( theater2 );
      theaters.saveNow();

      System.out.println( "theaters = " + theaters );
      Voyager.shutdown();
      }
    catch( VoyagerException exception )
      {
      System.err.println(exception);
      }
    }
  }
```

Application voyager1.0.0\examples\db\Db2B.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.db;

import COM.objectspace.voyager.*;
import VVector;

public class Db2B
  {
  public static void main( String args[] )
    {
    try
      {
      // connect to persistent vector
      VVector theaters =
        (VVector) VObject.forObjectAt( "localhost:8000/Theaters" );
      int count = theaters.size();

      for( int i = 0; i < count; i++ )
        System.out.println( "theater "+i+" = "+theaters.elementAt( i ) );

      Voyager.shutdown();
      }
    catch( VoyagerException exception )
      {
      System.err.println(exception);
      }
    }
  }
```

Flushing Objects

To flush an object from memory to database storage so that its memory can be reclaimed, use one of the following methods.

- `flushNow()`

  Saves the object in its program's database and allows it to be garbage-collected. Generates an `ObjectEvent.FLUSHING` event immediately prior to the save. Subsequent messages sent to the object via virtual references cause the object to be reloaded.

- `flushWhenInactiveFor( long` *milliseconds* `)`

  Tells the persistence service to send the object `flushNow()` if it does not receive any incoming messages or objects for the specified number of milliseconds.

- `flushNever()`

  Prevents the persistence service from sending the object `flushNow()`. Once `flushNever()` is executed, the object is not flushed unless it is explicitly sent `flushNow()`.

The rules governing when an object is actually flushed are similar to the rules of mobility. When it is time to flush an object, `flushNow()` uses `releaseObject()` to finish current encounters and waits for all outstanding remote messages to complete before actually flushing the object. Messages that arrive after the flush process begins are queued and resent after the object is flushed, causing the object to be autoloaded. In other words, Voyager carefully ensures that the flushing process is transparent to messaging.

The next section contains an example of flushing.

Mobile Persistence

If a persistent object is moved, it is removed from the source program's database and saved in the destination program's database. Unless you specify otherwise, the forwarder that the moved objects leaves behind is persistent.

The following example demonstrates mobile persistence and a common use of flushing. The `Db3A.java` example program creates and launches a persistent `MovieGoer` agent with alias `PugWash`. `PugWash` uses the persistent vector of theaters created in "Distributed Persistence" on page 249 to locate the individual theaters. `PugWash` then visits each theater in turn to watch a random movie.

The `setPersistent()` method is used to initially instruct the agent to be persistent without causing it to be saved immediately. When the servers are shut down and then restarted, the theaters, the agent, and the forwarders are in storage. The `Db3B.java` example program connects to the agent via its persistent forwarders, prints the agent, and then tells the agent to die. The `dieNow()` method causes the persistent agent and its persistent forwarders to be garbage-collected and removed from storage.

From the `\voyager\examples\db` directory, use the following commands to prepare and compile `Db3A.java` and `Db3B.java`:

```
vcc MovieGoer
javac VMovieGoer.java MovieGoer.java Db3A.java Db3B.java
```

Restart the persistent servers created in “Distributed Persistence” on page 249 in two separate windows. Then, run `Db3A.class` in a third window. When `Db3A.class` finishes, terminate the servers by pressing Ctrl+C in Windows 1 and 2.

Window 1

```
>voyager 7000 -d 7000.db
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:7000
database = 1 object, 0 classes
at Theater( monoplex ), see men in black
move to Theater( metropolis )
>
```

Window 2

```
>voyager 8000 -d 8000.db
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:8000
database = 2 objects, 0 classes
at Theater( metropolis ), see independence day
finished, go to sleep
>
```

Window 3

```
>java examples.db.Db3A
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1127
construct MovieGoer( Pugwash, seen: [] )
move to Theater( monoplex )
>
```

Now restart the persistent servers in the first and second windows and run `Db3B.class` in the third window.

Window 1

```
>voyager 7000 -d 7000.db
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:7000
database = 1 object, 0 classes
at Theater( monoplex ), see men in black
move to Theater( metropolis )
>voyager 7000 -d 7000.db
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:7000
database = 2 objects, 0 classes
>
```

Window 2

```
>voyager 8000 -d 8000.db
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:8000
database = 2 objects, 0 classes
at Theater( metropolis ), see independence day
finished, go to sleep
>voyager 8000 -d 8000.db
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:8000
database = 3 objects, 0 classes
>
```

Window 3

```
>java examples.db.Db3A
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.203:1093, root = \
construct MovieGoer( Pugwash, seen: [] )
move to Theater( monoplex )
>java examples.db.Db3B
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1142
moviegoer = MovieGoer( Pugwash, seen: [men in black, independence day] )
>
```

Class voyager1.0.0\examples\db\Moviegoer.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.db;

import java.util.*;
import COM.objectspace.voyager.*;
import VVector;

public class MovieGoer extends Agent implements ObjectListener
  {
  static Random random = new Random();
  int index;
  String name;
  Vector movies = new Vector();
  Vector itinerary;

  public MovieGoer( String name )
    {
    this.name = name;
    System.out.println( "construct " + this );

    //listening for events needs to be permanent, not transient
    //so add myself as an assistant not a listener
    addAssistant( this );
    }

  public String toString()
    {
    return "MovieGoer( " + name + ", seen: " + movies + " )";
    }

  public void seeMovies()
    {
    try
      {
      VVector theaters =
       (VVector) VObject.forObjectAt( "localhost:8000/Theaters" );
      itinerary = (Vector) theaters.clone();
      travel();
      }
    catch( VoyagerException exception )
      {
      System.err.println( exception );
      }
    }

  public void atTheater( Theater theater )
    {
    int choice = Math.abs( random.nextInt() ) % theater.getMovies().size();
    String movie = (String) theater.getMovies().elementAt( choice );
    System.out.println( "at " + theater + ", see " + movie );
    movies.addElement( movie );
    ++index;
    travel();
```

```
    }

  private void travel()
    {
    try
      {
      if( index < itinerary.size() )
        {
        VObject theater = (VTheater) itinerary.elementAt( index );
        System.out.println( "move to " + theater );

        try{ Thread.sleep( 2000 ); } catch(InterruptedException exception) {}

        moveTo( theater, "atTheater" );
        }
      else
        {
        System.out.println( "finished, go to sleep" );
        flushNow();
        }
      }
    catch( VoyagerException exception )
      {
      System.err.println( exception );
      }
    }

  public void objectEvent( ObjectEvent event )
    {
    if( event.getCode() == ObjectEvent.LOADED )
      System.out.println( "loaded " + this );
    }
  }
```

Application voyager1.0.0\examples\db\Db3A.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.db;

import COM.objectspace.voyager.*;
import COM.objectspace.voyager.db.*;

public class Db3A
  {
  public static void main( String args[] )
    {
    try
      {
      Voyager.setDb( new VoyagerDb( "Db3A.db", true ) );
     VMovieGoer moviegoer = new VMovieGoer( "Pugwash", "localhost/PugWash" );
      moviegoer.setPersistent( true );
      moviegoer.seeMovies();
      Voyager.shutdown();
      }
    catch( VoyagerException exception )
```

```
      {
      System.err.println(exception);
      }
    }
  }
```

Application voyager1.0.0\examples\db\Db3B.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.db;

import COM.objectspace.voyager.*;

public class Db3B
  {
  public static void main( String args[] )
    {
    try
      {
      VMovieGoer moviegoer =
        (VMovieGoer) VObject.forObjectAt( "localhost:7000/PugWash" );
      System.out.println( "moviegoer = " + moviegoer );
      moviegoer.dieNow();
      Voyager.shutdown();
      }
    catch( VoyagerException exception )
      {
      System.err.println(exception);
      }
    }
  }
```

Database Administration

The Voyager system accesses a database via the `Db` interface, allowing you to access a database directly. Although not necessary, direct access is useful when creating a database administrative tool.

Use the `Db` database interface and the `DbInfo` class to access a database directly. `Db` provides methods like `put()`, `get()`, and `remove()`. To load an object from a database, invoke `get()` with the object's GUID or alias.

`DbInfo` contains information about an object in the database, such as its GUID, alias, class name, and garbage collection data. `Db` has methods for obtaining `DbInfo` about all the objects in a database or a restricted subset of objects, such as all objects that are instances of a particular class. To save an object to a database, create a `DbInfo` that describes the object. Then, invoke `put()` on the database with the `DbInfo` and the object to be stored.

The `Db4.java` example program opens one of the databases created by the previous example, loads its `DbInfo`, and then loads all of the `Theater` objects into memory and prints them.

From the `\voyager1.0.0\examples\db` directory, use the following command to compile `Db4.java`:

```
javac Db4.java
```

Note: The `Db4.java` example program uses the `8000.db` database file created in the "Mobile Persistence" section on page 254.

Run `Db4.class` in a window.

```
>java examples.db.Db4
size = 2
info[ 0 ] -> DbInfo( java.util.Vector, Theaters, false, 0, 873926585,
163-83-24
-222-134-54-207-134-81-229-72-134-111-72-159-192 )
info[ 1 ] -> DbInfo( examples.db.Theater, null, false, 0, 873926584,
64-28-171-
8-203-212-143-68-74-110-72-134-111-72-159-192 )
voyager(tm) 1.0.0, copyright objectspace 1997
object[ 1 ] -> Theater( metropolis )
>
```

Class voyager1.0.0\examples\db\Db4.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.db;

import COM.objectspace.voyager.*;
import COM.objectspace.voyager.db.*;
import java.util.Vector;

public class Db4
  {
  public static void main( String args[] )
    {
    try
      {
      // open the database
      VoyagerDb db = new VoyagerDb( "8000.db", false );
      System.out.println( "size = " + db.size() );
      Vector infos = db.getAllDbInfo(); // get info

      // iterate through all info
      for( int i = 0; i < infos.size(); i++ )
        {
        DbInfo info = (DbInfo) infos.elementAt( i );
        System.out.println( "info[ " + i + " ] -> " + info );

        // load all objects whose class is "example.db.Theater"
        if( info.classname.toString().equals( "examples.db.Theater" ) )
          {
          Object object = db.get( info.guid );
          System.out.println( "object[ " + i + " ] -> " + object );
          }
        }

      db.close(); // close the database
      Voyager.shutdown();
      }
    catch( VoyagerException exception )
      {
      System.err.println(exception);
      }
    }
  }
```

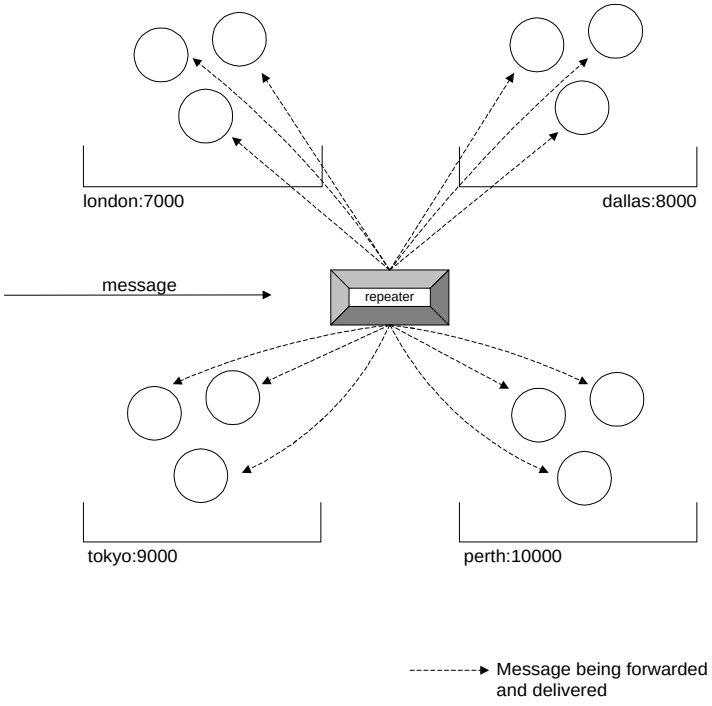
15

Space: Scalable Group Communication

Distributed systems require features for communicating with groups of objects. Following are some examples:

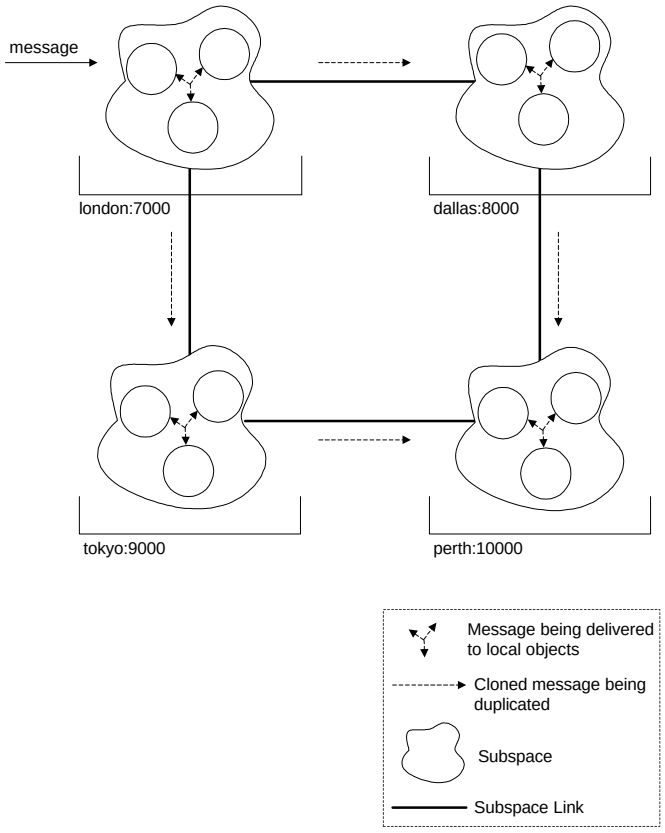
- Stock quote systems use a distributed event feature to send stock price events to customers around the world.
- Voting systems use a distributed messaging feature (multicast) to ask voters around the world for their views on a particular matter.
- News services use a distributed publish/subscribe feature to send broadcasts only to readers who are interested in the broadcast topic.

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 the number of objects in the target group is small, but does not scale well when large numbers of objects are involved.

ObjectSpace Voyager™ Core Technology (Voyager) uses an innovative architecture for message/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 to form a larger logical group, or *Space*. Each message sent into one of the subspaces is cloned to each of the neighboring subspaces before being delivered to every object in the local subspace, resulting in a rapid, parallel fanout of the message to every object in the Space. A mechanism in each subspace ensures that no message or event is processed more than once, regardless of how the subspaces are linked together.



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

Creating a Space

To create a logical Space, create subspaces anywhere in the network, add objects to each subspace using `add()`, and connect the subspaces together using `connect()`. Connection is symmetric, so if you connect `subspace1` to `subspace2`, you need not connect `subspace2` to `subspace1`.

A single subspace can hold different kinds of objects. You can add a virtual reference to a subspace even if the referenced object is not in the same program as the subspace.

When you send a message into a subspace, the message propagates itself to every neighbor of the subspace and then delivers itself to the references within the subspace. As the message propagates, it leaves behind a marker unique to that message and remembered by the subspace for a default period of five minutes. If a clone of that message re-enters the subspace, the clone detects the marker and automatically self-destructs. This marker allows you to connect subspaces together to form arbitrary topologies without the possibility of multiple message delivery. The parallel message fanout means that the more interconnected the subspaces, the more fault-tolerant they become in the face of individual network failures.

The `Space1.java` example program creates a distributed Space that holds instances of `SportsFan`. The program creates a subspace in each of servers 7000, 8000, and 9000, adds one or more virtual references to `SportsFan` objects in each subspace, and then connects the subspaces together to form a larger, logical Space.

`Space1.java` is used by subsequent example programs to initialize a Space before demonstrating features like multicast and distributed events. The `NewsListener` interface of the `SportsFan` and the `score()` and `newsFlash()` methods are not used by this example.

From the `\voyager1.0.0\examples\space` directory, use the following commands to prepare and compile the `Space1.java` example program:

```
vcc SportsFan
javac VSportsFan.java SportsFan.java Space1.java
```

Start a server on each of ports 7000, 8000, and 9000 in three different windows. Then run `Space1.class` in a fourth window. When `Space1.class` terminates, terminate the servers by pressing Ctrl+C in Windows 1, 2, and 3.

Window 1

```
>voyager 7000
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:7000
construct SportsFan( sally )
>
```

Window 2

```
>voyager 8000
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:8000
construct SportsFan( dave )
construct SportsFan( mary )
>
```

Window 3

```
>voyager 9000
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:9000
construct SportsFan( graham )
>
```

Window 4

```
>java examples.space.Space1
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1343
subspace1 = Subspace( objects: 1, neighbors: 2 )
subspace2 = Subspace( objects: 2, neighbors: 2 )
subspace3 = Subspace( objects: 1, neighbors: 2 )
>
```

Class voyager1.0.0\examples\space\SportsFan.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.space;

public class SportsFan implements NewsListener, java.io.Serializable
  {
  String name;

  public SportsFan( String name )
    {
    this.name = name;
    System.out.println( "construct " + this );
    }

  public String toString()
    {
    return "SportsFan( " + name + " )";
    }

  public String getName()
    {
    return name;
    }

  public void score( String team1, int x, String team2, int y )
    {
    System.out.println(
      this + " gets score: " + team1 + " " + x + ", " + team2 + " " + y );
    }

  public void newsFlash( NewsEvent event )
    {
    System.out.println( this + " gets " + event );
    }
  }
```

Application voyager1.0.0\examples\space\Space1.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.space;
import COM.objectspace.voyager.*;
import COM.objectspace.voyager.space.*;

public class Space1
  {
  public static void main( String args[] )
    {
    try
      {
      // create and populate subspace on local server 7000
      VSubspace subspace1 = new VSubspace( "localhost:7000/Subspace1" );
      subspace1.liveForever();
      VSportsFan fan1 = new VSportsFan( "sally", "localhost:7000" );
      fan1.liveForever();
      subspace1.add( fan1 );

      // create and populate subspace on local server 8000
      VSubspace subspace2 = new VSubspace( "localhost:8000/Subspace2" );
      subspace2.liveForever();
      VSportsFan fan2 = new VSportsFan( "dave", "localhost:8000" );
      fan2.liveForever();
      subspace2.add( fan2 );
      VSportsFan fan3 = new VSportsFan( "mary", "localhost:8000" );
      fan3.liveForever();
      subspace2.add( fan3 );

      // create and populate subspace on local server 9000
      VSubspace subspace3 = new VSubspace( "localhost:9000/Subspace3" );
      subspace3.liveForever();
      VSportsFan fan4 = new VSportsFan( "graham", "localhost:9000" );
      fan4.liveForever();
      subspace3.add( fan4 );

      // link subspaces to create larger logical space
      subspace1.connect( subspace2 );
      subspace1.connect( subspace3 );
      subspace2.connect( subspace3 );

      // display contents of each subspace
      System.out.println( "subspace1 = " + subspace1 );
      System.out.println( "subspace2 = " + subspace2 );
      System.out.println( "subspace3 = " + subspace3 );

      Voyager.shutdown();
      }
    catch( VoyagerException exception )
      {
      System.err.println( exception );
      }
    }
  }
```

Multicasting to a Space

With Voyager, you can multicast a regular Java™ message to a group of objects in a Space. You can send a one-way message to all objects of a particular type in a Space or to all objects that implement a particular interface in a Space. To do so, first create a virtual reference of the desired type, supplying a virtual reference to a subspace as an argument. Then send a message to the newly created virtual reference, which acts as a gateway into the Space. Messages sent via this virtual reference are sent by a `OneWayMulticast` messenger instead of a `Sync` messenger and are delivered only to objects in that Space that are type-compatible with the virtual reference. You can create multiple gateways of multiple types in the same logical Space.

Multicast messages can be selective about the objects they are delivered to. If you construct `OneWayMulticast` with an object that implements the interface `COM.objectspace.voyager.util.Selector`, the multicast messenger verifies that each object passes the selector's criteria before delivering a message to each object. Voyager's publish/subscribe mechanism is implemented with selective multicasts. See "Publishing Messages to a Space" on page 279 for details.

For performance reasons, a multicast message is delivered using a single thread per subspace. If a multicast message arrives at a subspace containing 10 objects, a single thread is allocated to the `OneWayMulticast` messenger, which then delivers the message to each of the 10 objects.

Because multicast is one-way, multicast messages have no return values. You can use the default message syntax for a multicast method only if it returns `void`. For example, the following code executes `void removeAllElements()` on every `VVector` in a Space:

```
VVector vector = new VVector( subspace ); // typesafe gateway into space
vector.removeAllElements(); // send to every VVector in the space
```

If you attempt to execute a method that returns anything other than `void`, a `ResultException` is thrown. If you wish to multicast a method that returns a value, you must explicitly specify a `OneWayMulticast` messenger when you execute the method call and ignore the returned `Result`. For example, you could use the code below to execute `boolean removeElement()` on every `VVector` object in a Space:

```
VVector vector = new VVector( subspace ); // typesafe gateway into space
vector.removeElement( new Integer( 42 ), new OneWayMulticast() );
```

The `Multicast1.java` example program creates a `VSportsFan` gateway into a subspace and then uses it to multicast `score()` to every `VSportsFan` in the Space. Use the following command to compile `Multicast1.java`:

```
javac Multicast1.java
```

Start a server on each of ports 7000, 8000, and 9000 in three different windows. Then, in a fourth window, run `Space1.class` to fill the Space with `VSportsFan` objects. When `Space1.class` completes, run `Multicast1.class` in the same window.

Window 1

```
>voyager 7000
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:7000
construct SportsFan( sally )
SportsFan( sally ) gets score: bulls 50, lakers 40
```

Window 2

```
>voyager 8000
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:8000
construct SportsFan( dave )
construct SportsFan( mary )
SportsFan( dave ) gets score: bulls 50, lakers 40
SportsFan( mary ) gets score: bulls 50, lakers 40
```

Window 3

```
>voyager 9000
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:9000
construct SportsFan( graham )
SportsFan( graham ) gets score: bulls 50, lakers 40
```

Window 4

```
>java examples.space.Space1
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1048
subspace1 = Subspace( objects: 1, neighbors: 2 )
subspace2 = Subspace( objects: 2, neighbors: 2 )
subspace3 = Subspace( objects: 1, neighbors: 2 )
>java examples.space.Multicast1
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1055
>
```

Application voyager1.0.0\examples\space\Multicast1.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.space;

import COM.objectspace.voyager.*;
import COM.objectspace.voyager.space.*;

public class Multicast1
  {
  public static void main( String args[] )
    {
    try
      {
      // connect to existing subspace on local server 7000
      VSubspace subspace1 =
        (VSubspace) VObject.forObjectAt( "localhost:7000/Subspace1" );

      // create a typesafe gateway into the subspace on local server 7000
      VSportsFan fans = new VSportsFan( subspace1 );

      // send a oneway message to every VSportsFan in the subspace
      fans.score( "bulls", 50, "lakers", 40 );

      // allow time for oneway multicast to flush
      try{ Thread.sleep( 2000 ); } catch( InterruptedException exception ) {}

      Voyager.shutdown();
      }
    catch( VoyagerException exception )
      {
      System.err.println( exception );
      }
    }
  }
```

Distributing JavaBeans Events

Voyager allows you to send events to objects in a Space using the standard JavaBeans™ event/listener model, defined as follows:

- Every Java event extends `java.util.EventObject` and contains data fields that describe the particular event.
- Every event class has an associated listener interface that extends `java.util.EventListener` and defines one or more methods that accept event arguments and return `void`.
- Every object that is to receive a given event implements the event's associated listener interface.
- Every object that is a source of events implements the `addListener()` and `removeListener()` methods, which allow listeners to register interest in receiving particular types of events.

You need not modify the event source or the listeners to take advantage of Voyager's distributed event feature. To broadcast events to every object in a Space that implements a particular listener interface, perform the following steps:

1. Use `vcc` to create a virtual class for the particular listener interface. The resulting virtual listener class implements the original listener interface.
2. Construct an instance of the virtual listener class with a single argument—a virtual reference to a subspace. This virtual listener becomes an event gateway into the Space.
3. Add the virtual listener to the event source using the appropriate `addListener()` method.
4. Send events to the virtual listener. The events are multicast to every object in the Space that implements the listener interface.

The `Events1.java` example program takes advantage of distributed JavaBean events. The program uses a `NewsSource` object to send `NewsEvent` events via `newsFlash()` to every object in a Space that implements the `NewsListener` interface. Because `SportsFan` implements `NewsListener`, every `SportsFan` object receives the `newsFlash()` method.

Use the following commands from the `voyager1.0.0\examples\space` directory to prepare and compile `Events1.java`:

```
vcc NewsListener
javac NewsListener.java VNewsListener.java NewsEvent.java
javac NewsSource.java Multicast1.java
```

Start a server on each of ports 7000, 8000, and 9000 in three different windows. Then run `Space1.class` in a fourth window to fill the Space with `VSportsFan` objects. When `Space1.class` completes, run `Events1.class` in the same window.

Window 1

```
>voyager 7000
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:7000
construct SportsFan( sally )
SportsFan( sally ) gets NewsEvent( surprise win by cowboys! )
```

Window 2

```
>voyager 8000
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:8000
construct SportsFan( dave )
construct SportsFan( mary )
SportsFan( dave ) gets NewsEvent( surprise win by cowboys! )
SportsFan( mary ) gets NewsEvent( surprise win by cowboys! )
```

Window 3

```
>voyager 9000
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:9000
construct SportsFan( graham )
SportsFan( graham ) gets NewsEvent( surprise win by cowboys! )
```

Window 4

```
>java examples.space.Space1
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1057
subspace1 = Subspace( objects: 1, neighbors: 2 )
subspace2 = Subspace( objects: 2, neighbors: 2 )
subspace3 = Subspace( objects: 1, neighbors: 2 )
>java examples.space.Events1
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1064
send news flash: surprise win by cowboys!
>
```

Class voyager1.0.0\examples\space\NewsEvent.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.space;

public class NewsEvent extends java.util.EventObject
  {
  String news; // the news

  public NewsEvent( NewsSource source, String news )
    {
    super( source );
    this.news = news;
    }

  public String toString()
    {
    return "NewsEvent( " + news + " )";
    }

  public String getNews()
    {
    return news;
    }
  }
```

Interface voyager1.0.0\examples\space\NewsListener.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.space;

public interface NewsListener extends java.util.EventListener
  {
  void newsFlash( NewsEvent event );
  }
```

Class voyager1.0.0\examples\space\NewsSource.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.space;

import java.util.Vector;

public class NewsSource
  {
  Vector listeners = new Vector();

  public void addNewsListener( NewsListener listener )
    {
    listeners.addElement( listener );
    }

  public void removeNewsListener( NewsListener listener )
    {
    listeners.removeElement( listener );
    }

  public void newsFlash( String news )
    {
    NewsEvent event = new NewsEvent( this, news );

    for( int i = 0; i < listeners.size(); i++ )
      ((NewsListener) listeners.elementAt( i )).newsFlash( event );
    }
  }
```

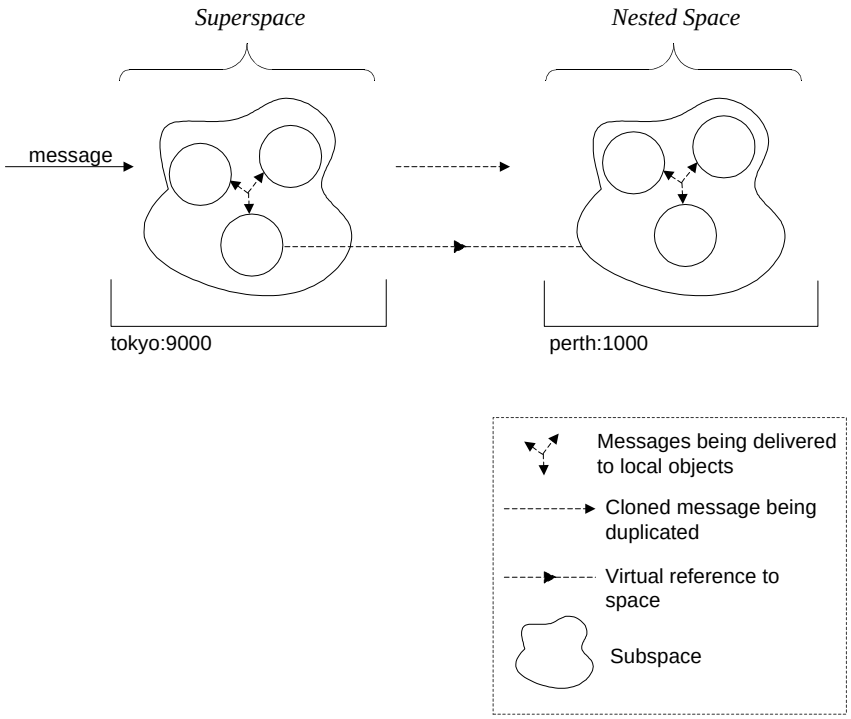
Application voyager1.0.0\examples\space\Events1.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.space;

import COM.objectspace.voyager.*;
import COM.objectspace.voyager.space.*;

public class Events1
  {
  public static void main( String args[] )
    {
    try
      {
      // connect to existing subspace on local server 7000
      VSubspace subspace1 =
        (VSubspace) VObject.forObjectAt( "localhost:7000/Subspace1" );

      // create news source and register subspace as listener
      NewsSource source = new NewsSource();
      source.addNewsListener( new VNewsListener( subspace1 ) );

      // send news event to every NewsListener in the space
      System.out.println( "send news flash: surprise win by cowboys!" );
      source.newsFlash( "surprise win by cowboys!" );

      // allow time for event to flush
      try{ Thread.sleep( 2000 ); } catch( InterruptedException exception ) {}

      Voyager.shutdown();
      }
    catch( VoyagerException exception )
      {
      System.err.println( exception );
      }
    }
  }
```

Nested Spaces

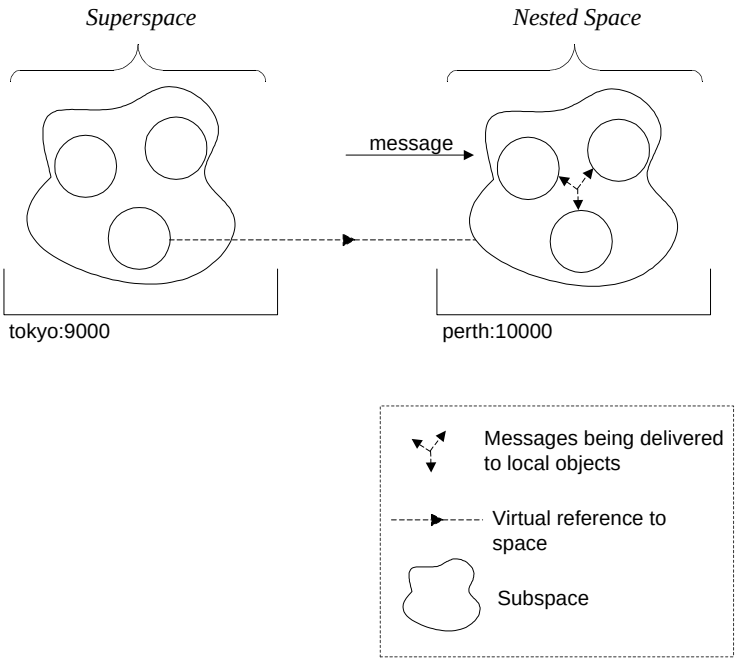
A Space that contains another Space is referred to as a *superspace*. In other words, one or more objects in a superspace are gateways to a contained Space. Consider a company that is geographically distributed across the world and multicasts to a Space of `Employee` objects. Using superspaces, the company could multicast to all employees at certain times and only to employees at a given regional office at other times.

When you send a multicast message to a superspace, the message is cloned and sent to all nested Spaces contained in the superspace. This concept is illustrated in the following diagram with a superspace that contains a nested Space. In reality, both a superspace and a nested Space can contain several subspaces; however, for simplicity, each is comprised of one subspace in this diagram.



When you send a multicast message to a Space that is nested within a superspace, the message is delivered to all objects in the nested Space. The message is not cloned to the superspace. The following diagram illustrates this concept. In reality, both a superspace and a nested Space

can contain several subspaces; however, for simplicity, each is comprised of one subspace in this diagram.



The `Nest1.java` example program demonstrates how to nest a subspace within another subspace. Larger Space topologies could be formed for more complex applications.

Use the following command from the `voyager1.0.0\examples\space` directory to compile `Nest1.java`:

```
javac Nest1.java
```

Start a server on port 8000 in one window, and then run `Nest1.class` in a second window.

Window 1

```
>voyager 8000
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:8000
construct SportsFan( Mickey )
SportsFan( Mickey ) gets score: USA 0, BRD 8
SportsFan( Mickey ) gets score: USA 0, Japan 11
```

Window 2

```
>java examples.space.Nest1
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1164
construct SportsFan( Minnie )
SportsFan( Minnie ) gets score: USA 0, Japan 11
>
```

Application voyager1.0.0\examples\space\Nest1.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.space;

import COM.objectspace.voyager.*;
import COM.objectspace.voyager.space.*;

public class Nest1
  {
  public static void main( String[] args )
    {
    try
      {
      // create a local subspace object. the logical space formed by
      // this single subspace will be the superspace
      VSubspace superspace = new VSubspace( "localhost" );

      // add a sports fan to the superspace
      superspace.add( new VSportsFan( "Minnie", "localhost" ) );

      // create a remote subspace object.  the logical space formed by
      // this single subspace wil be the contained space
      VSubspace containedSpace = new VSubspace( "localhost:8000" );

      // add a sports fan to the contained space
      containedSpace.add( new VSportsFan( "Mickey", "localhost:8000" ) );

      // now create the sports fan gateways
      // to the contained space and the superspace
      VSportsFan subGateway = new VSportsFan( containedSpace );
      VSportsFan superGateway = new VSportsFan( superspace );

      // the final step is to add the sub-gateway to the superspace -
      // this nests the spaces
      superspace.add( subGateway );

      // multicast to all items in the contained space
      subGateway.score( "USA", 0, "BRD", 8 );

      // multicast to all items in the superspace
      superGateway.score( "USA", 0, "Japan", 11 );

      // let the threads delivering th multicasts finish
      try{ Thread.sleep( 2000 ); }catch( InterruptedException e ){}

      Voyager.shutdown();
      }
    catch( VoyagerException exception )
      {
      System.out.println( exception );
      }
    }
  }
```

Publishing Messages to a Space

You can broadcast a message or event to every object in a Space that registers interest in a particular subject. Voyager supports this feature with subscription-based, one-way multicasts and with the properties mechanism described on page 53 in Chapter 5, “Fundamental ORB Features.” To publish a message to all objects in a Space interested in a particular subject, create a subscription object and use `addSubject()` to register the subject with the associated message. Specify subjects in a hierarchical way with fields separated by periods, like `sports.bulls` and `sports.lakers`. Next, create a `OneWayMulticast` message with the subscription as the argument to the constructor. Finally, broadcast the message into the Space with the one-way multicast as an explicit extra parameter to the message.

The `Subscription` class implements `COM.objectspace.voyager.util.Selector`. Therefore, publish/subscribe is a specific kind of selective multicast that selects objects based on whether or not they have declared interest in one or more of the subscription subjects.

To receive all messages that are published to a particular subject, use `addProperty()` with the key set to `Subscription.SUBSCRIBE` and the value set to the subject of interest. The asterisk (`*`) wildcard matches the next field and the left angle bracket (`<`) wildcard matches all remaining fields. For example, the first line of code below is used to receive all messages associated with a subject that has exactly two fields and begins with *sports*. The second line of code is used to receive all messages associated with a subject that has two or more fields and begins with *sports*.

```
object.addProperty( Subscription.SUBSCRIBE, "sports.*" );
object.addProperty( Subscription.SUBSCRIBE, "sports.<" );
```

The `Publish1.java` example program implements Voyager’s publish/subscribe feature with messages. The program sets up delivery of `score()` messages to all `VSportsFan` objects in a Space. For simplicity, all of the objects are placed into a single subspace, but the program would run identically if the objects were distributed across a network of connected subspaces.

Use the following command from the `voyager1.0.0\examples\space` directory to compile the `Publish1.java` example program:

```
javac Publish1.java
```

Then run `Publish1.class`.

```
>java examples.space.Publish1
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1166
construct SportsFan( sally )
sally subscribes to sports.bulls
construct SportsFan( jane )
jane subscribes to sports.celtics
construct SportsFan( alf )
alf subscribes to sports.*
construct SportsFan( smiffy )
smiffy subscribes to sports.bulls, sports.lakers
construct SportsFan( dave )
dave subscribes to nothing
publish 40, 50 to sports.bulls, sports.mavericks
publish 20, 15 to sports.bulls, sports.lakers
SportsFan( sally ) gets score: bulls 40, mavericks 50
SportsFan( sally ) gets score: bulls 20, lakers 15
SportsFan( alf ) gets score: bulls 40, mavericks 50
SportsFan( alf ) gets score: bulls 20, lakers 15
SportsFan( smiffy ) gets score: bulls 20, lakers 15
SportsFan( smiffy ) gets score: bulls 40, mavericks 50
>
```

Class voyager1.0.0\examples\space\Publish1.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.space;

import COM.objectspace.voyager.*;
import COM.objectspace.voyager.multicast.*;
import COM.objectspace.voyager.space.*;
import COM.objectspace.voyager.util.*;

public class Publish1
  {
  public static void main( String args[] )
    {
    try
      {
      VSubspace subspace = new VSubspace( "localhost" );
      VSportsFan fans = new VSportsFan( subspace );

      VSportsFan fan1 = new VSportsFan( "sally", "localhost" );
      fan1.addProperty( Subscription.SUBSCRIBE, "sports.bulls" );
      System.out.println( "sally subscribes to sports.bulls" );
      subspace.add( fan1 );

      VSportsFan fan2 = new VSportsFan( "jane", "localhost" );
      fan2.addProperty( Subscription.SUBSCRIBE, "sports.celtics" );
      System.out.println( "jane subscribes to sports.celtics" );
```

```
        subspace.add( fan2 );

        VSportsFan fan3 = new VSportsFan( "alf", "localhost" );
        fan3.addProperty( Subscription.SUBSCRIBE, "sports.*" );
        System.out.println( "alf subscribes to sports.*" );
        subspace.add( fan3 );

        VSportsFan fan4 = new VSportsFan( "smiffy", "localhost" );
        fan4.addProperty( Subscription.SUBSCRIBE, "sports.bulls" );
        fan4.addProperty( Subscription.SUBSCRIBE, "sports.lakers" );
        System.out.println("smiffy subscribes to sports.bulls, sports.lakers");
        subspace.add( fan4 );

        VSportsFan fan5 = new VSportsFan( "dave", "localhost" );
        System.out.println( "dave subscribes to nothing" );
        subspace.add( fan5 );

        Subscription subscription1 = new Subscription();
        subscription1.addSubject( "sports.bulls" );
        subscription1.addSubject( "sports.mavericks" );
        System.out.println("publish 40, 50 to sports.bulls, sports.mavericks");
        fans.score("bulls",40,"mavericks",50,new
OneWayMulticast(subscription1));

        Subscription subscription2 = new Subscription();
        subscription2.addSubject( "sports.bulls" );
        subscription2.addSubject( "sports.lakers" );
        System.out.println( "publish 20, 15 to sports.bulls, sports.lakers" );
        fans.score("bulls",20,"lakers",15,new OneWayMulticast(subscription2));

        // allow time for publish messages to flush
        try{ Thread.sleep( 2000 ); } catch( InterruptedException exception ) {}

        Voyager.shutdown();
        }
      catch( VoyagerException exception )
        {
        System.err.println( exception );
        }
    }
  }
```

Creating a Persistent Space

Voyager `Subspace` objects are serializable and fully compatible with Voyager's persistence subsystem. When a subspace is saved, virtual references in the subspace are saved but the associated objects are not.

You can create a persistent subspace that contains virtual references to persistent objects. Then you can shut down and restart the persistent servers, and the subspace and references are autoloaded as usual.

Use the following command from the `voyager1.0.0\examples\space` directory to compile the `PersistentSpace1.java` example program:

```
javac PersistentSpace1.java
```

Start a persistent server on each of ports 7000, 8000, and 9000 in three different windows. Run `PersistentSpace1.class` in a fourth window, and then terminate the servers by pressing Ctrl+C in Windows 1, 2, and 3.

Window 1

```
>voyager 7000 -cd 7000.db
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:7000
database = 0 objects, 0 classes
construct SportsFan( sally )
>
```

Window 2

```
>voyager 8000 -cd 8000.db
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:8000
database = 0 objects, 0 classes
construct SportsFan( dave )
construct SportsFan( mary )
>
```

Window 3

```
>voyager 9000 -cd 9000.db
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.238:9000, root = \
database = 0 objects, 0 classes
construct SportsFan( graham )
>
```

Window 4

```
>java examples.space.PersistentSpace1
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1168
subspace1 = Subspace( objects: 1, neighbors: 2 )
subspace2 = Subspace( objects: 2, neighbors: 2 )
subspace3 = Subspace( objects: 1, neighbors: 2 )
>
```

Now restart the persistent servers in the first three windows and run `Multicast1.class` (used in “Multicasting to a Space” on page 268) in the fourth window. The persistent subspaces and the persistent objects that the servers refer to are autoloaded as `Multicast1.class` executes.

Window 1

```
>voyager 7000 -cd 7000.db
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:7000
database = 0 objects, 0 classes
construct SportsFan( sally )
>voyager 7000 -d 7000.db
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:7000
database = 2 objects, 0 classes
SportsFan( sally ) gets score: bulls 50, lakers 40
```

Window 2

```
>voyager 8000 -cd 8000.db
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:8000
database = 0 objects, 0 classes
construct SportsFan( dave )
construct SportsFan( mary )
>voyager 8000 -d 8000.db
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:8000
database = 3 objects, 0 classes
SportsFan( dave ) gets score: bulls 50, lakers 40
SportsFan( mary ) gets score: bulls 50, lakers 40
```

Window 3

```
>voyager 9000 -cd 9000.db
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:9000
database = 0 objects, 0 classes
construct SportsFan( graham )
>voyager 9000 -d 9000.db
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:9000
database = 2 objects, 0 classes
SportsFan( graham ) gets score: bulls 50, lakers 40
```

Window 4

```
>java examples.space.PersistentSpace1
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1168
subspace1 = Subspace( objects: 1, neighbors: 2 )
subspace2 = Subspace( objects: 2, neighbors: 2 )
subspace3 = Subspace( objects: 1, neighbors: 2 )
>java examples.space.Multicast1
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1175
>
```

Application voyager1.0.0\examples\space\PersistentSpace1.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.space;

import COM.objectspace.voyager.*;
import COM.objectspace.voyager.space.*;

public class PersistentSpace1
  {
  public static void main( String args[] )
    {
    try
      {
      // create and populate subspace on local server 7000
      VSubspace subspace1 = new VSubspace( "localhost:7000/Subspace1" );
      subspace1.liveForever();
      VSportsFan fan1 = new VSportsFan( "sally", "localhost:7000" );
      fan1.liveForever();
      fan1.saveNow();
      subspace1.add( fan1 );

      // create and populate subspace on local server 8000
```

```
      VSubspace subspace2 = new VSubspace( "localhost:8000/Subspace2" );
      subspace2.liveForever();
      VSportsFan fan2 = new VSportsFan( "dave", "localhost:8000" );
      fan2.liveForever();
      fan2.saveNow();
      subspace2.add( fan2 );
      VSportsFan fan3 = new VSportsFan( "mary", "localhost:8000" );
      fan3.liveForever();
      fan3.saveNow();
      subspace2.add( fan3 );

      // create and populate subspace on local server 9000
      VSubspace subspace3 = new VSubspace( "localhost:9000/Subspace3" );
      subspace3.liveForever();
      VSportsFan fan4 = new VSportsFan( "graham", "localhost:9000" );
      fan4.liveForever();
      fan4.saveNow();
      subspace3.add( fan4 );

      // link subspaces to create larger logical space
      subspace1.connect( subspace2 );
      subspace1.connect( subspace3 );
      subspace2.connect( subspace3 );

      // persist the subspaces
      subspace1.saveNow();
      subspace2.saveNow();
      subspace3.saveNow();

      // display contents of each subspace
      System.out.println( "subspace1 = " + subspace1 );
      System.out.println( "subspace2 = " + subspace2 );
      System.out.println( "subspace3 = " + subspace3 );

      Voyager.shutdown();
      }
    catch( VoyagerException exception )
      {
      System.err.println( exception );
      }
    }
  }
```

Maintaining a Subspace

The objects and neighbors of a subspace can be disconnected or killed. By default, a subspace ignores such events. You can instruct a subspace to automatically purge itself of disconnected or dead objects and neighbors by using `setPurgePolicy()` with one of the following flags:

- `Subspace.DIED`

  Remove references to objects and neighbors that are dead.

- `Subspace.DISCONNECTED`

  Remove references to objects and neighbors that are disconnected.

- `Subspace.ALL`

  Remove references to objects and neighbors that are dead or disconnected.

- `Subspace.NONE`

  Never remove references (default policy).

Every 60 seconds, a subspace performs an internal purge in which the following actions occur:

- If the subspace policy is not `NONE`, each subspace neighbor is sent a lightweight message asking if it has been disconnected or killed. The subspace then reacts according to its policy.
- If the subspace policy is not `NONE`, each virtual reference in the subspace is sent a lightweight message asking if its objects have been disconnected or killed. The subspace then reacts according to its policy.
- All message markers that have exceeded their life spans are removed. The default life span of a marker is five minutes.

You can force a subspace to perform a purge at any time by sending the subspace a `purge()` message.

The next example demonstrates the effects of purging. First, the `Space1.java` example program is used to create and populate a network of subspaces. Next, `Maintain1A.java` is used to add a remote object to `subspace1`. At this point, `subspace1` contains a virtual reference to a local sports fan named `sally` and a virtual reference to the remote sports fan named `galileo` in the `Maintain1A.java` program. Finally, `Maintain1B.java` kills `subspace2` and then requests that you terminate `Maintain1A.java`. Once you do so, `subspace1` contains references to one dead neighbor and one disconnected object. The contents of `subspace2` are printed after every manual purge to demonstrate the effect of each purge flag.

Use the following command from the `voyager1.0.0\examples\space` directory to compile the `Maintain1A.java` and `Maintain1B.java` example programs:

```
javac Maintain1A.java Maintain1B.java
```

Start a server on each of ports 7000, 8000, and 9000 in three different windows. In a fourth window, first run `Space1.class`, and then run `Maintain1A.class` without terminating it. Finally, in a fifth window, run `Maintain1B.class`. When instructed by `Maintain1B`, terminate `Maintain1A` by pressing Ctrl+C in Window 4.

Window 1

```
>voyager 7000
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:7000
construct SportsFan( sally )
```

Window 2

```
>voyager 8000
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:8000
construct SportsFan( dave )
construct SportsFan( mary )
```

Window 3

```
>voyager 9000
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:9000
construct SportsFan( graham )
```

Window 4

```
>java examples.space.Space1
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1184
subspace1 = Subspace( objects: 1, neighbors: 2 )
subspace2 = Subspace( objects: 2, neighbors: 2 )
subspace3 = Subspace( objects: 1, neighbors: 2 )
>java examples.space.Maintain1A
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1191
construct SportsFan( galileo )
subspace1 = Subspace( objects: 2, neighbors: 2 )
terminate me when Maintain1B tells you to...
>
```

Window 5

```
>java examples.space.Maintain1B
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1193
subspace1 = Subspace( objects: 2, neighbors: 2 )
tell neighbor subspace2 to die...
sleeping for 10 seconds...
please terminate Maintain1A at this point
subspace1 = Subspace( objects: 2, neighbors: 2 )
purging disconnected objects/neighbors...
subspace1 = Subspace( objects: 1, neighbors: 2 )
purging dead objects/neighbors...
subspace1 = Subspace( objects: 1, neighbors: 1 )
>
```

Application voyager1.0.0\examples\space\Maintain1A.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.space;

import COM.objectspace.voyager.*;
import COM.objectspace.voyager.space.*;

public class Maintain1A
  {
  public static void main( String args[] )
    {
    try
      {
      // connect to subspace on local server 7000
      VSubspace subspace1 =
        (VSubspace) VObject.forObjectAt( "localhost:7000/Subspace1" );

      // add reference to local object into remote subspace
      VSportsFan fan5 = new VSportsFan( "galileo", "localhost" );
      fan5.liveForever();
      subspace1.add( fan5 );

      // display subspace1
      System.out.println( "subspace1 = " + subspace1 );
      System.out.println( "terminate me when Maintain1B tells you to..." );
      }
    catch( VoyagerException exception )
      {
      System.err.println( exception );
      }
    }
  }
```

Application voyager1.0.0\examples\space\Maintain1B.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.space;

import COM.objectspace.voyager.*;
import COM.objectspace.voyager.space.*;

public class Maintain1B
  {
  public static void main( String args[] )
    {
    try
      {
      VSubspace subspace1 =
        (VSubspace) VObject.forObjectAt( "localhost:7000/Subspace1" );
      System.out.println( "subspace1 = " + subspace1 );

      System.out.println( "tell neighbor subspace2 to die..." );
      VSubspace subspace2 =
        (VSubspace) VObject.forObjectAt( "localhost:8000/Subspace2" );
      subspace2.dieNow();

      System.out.println( "sleeping for 10 seconds..." );
      System.out.println( "please terminate Maintain1A at this point" );
     try{ Thread.sleep( 10000 ); } catch( InterruptedException exception ) {}
      System.out.println( "subspace1 = " + subspace1 );

      System.out.println( "purging disconnected objects/neighbors..." );
      subspace1.purge( Subspace.DISCONNECTED );
      System.out.println( "subspace1 = " + subspace1 );

      System.out.println( "purging dead objects/neighbors..." );
      subspace1.purge( Subspace.DIED );
      System.out.println( "subspace1 = " + subspace1 );

      Voyager.shutdown();
      }
    catch( VoyagerException exception )
      {
      System.err.println(exception);
      }
    }
  }
```

16

Federated Directory Service

The built-in lookup mechanism of ObjectSpace Voyager™ Core Technology is easy to use and, combined with forwarders, is a powerful way to get virtual references to remote and mobile objects. However, it requires up-front knowledge of the object's current or past location. Traditionally, the way to circumvent this requirement is to place a central directory service at a well-known location. An extension of this approach is to allow seamless chaining of distributed directory services, which minimizes the single-server bottleneck/point-of-failure associated with monolithic directory services. Distributed directory services that support transparent chaining are called *federated directory services*.

Voyager allows you to register an object in a distributed hierarchical directory structure. You can associate objects with path names comprised of simple strings separated by slashes, such as `fruit/citrus/lemon` or `animal/mammal/cat`. To create a directory structure, construct a `Directory` object. The `Directory` class is serializable and fully compatible with Voyager's persistence subsystem. To save a `Directory` object to its local database, use `saveNow()`.

To associate a simple string with an object, use the `put()` method with a virtual reference to another `Directory` anywhere in a network. If invoked with a path name, `put()` looks up the `VDirectory` associated with the head of the path name, and then forwards the `put()` message with the tail of the path name. If the head of the path name is not present or is not associated with a `VDirectory`, a `DirectoryException` is thrown. The same logic applies to the `get()` and `remove()` methods.

The `Directory1A.java` and `Directory1B.java` example programs demonstrate the construction and access of a persistent federated directory service. `Directory1A.java` creates a directory service that spans the persistent servers on ports 7000 and 8000, and `Directory1B.java` accesses the federated directory service.

From the `\voyager1.0.0\examples\directory` directory, use the following command to compile the directory service example programs:

```
javac Directory1A.java Directory1B.java
```

Start a persistent server on each of ports 7000 and 8000 in two different windows. Run `Directory1A.class` in a third window, and then terminate the servers by pressing Ctrl+C in Windows 1 and 2.

Window 1

```
>voyager 7000 -cd 7000.db
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:7000
database = 0 objects, 0 classes
>
```

Window 2

```
>voyager 8000 -cd 8000.db
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:8000
database = 0 objects, 0 classes
>
```

Window 3

```
>java examples.directory.Directory1A
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1334
created and saved {calcium->Ca, gold->Au}
created and saved {calcium->20, gold->79}
created and saved {number->Directory(
208.6.239.200:8000/235-87-26-169-220-147-1
55-34-99-143-72-134-111-72-159-192 ), symbol->Directory(
208.6.239.200:7000/180-
226-92-116-207-35-52-77-227-43-72-134-111-72-155-216 )}
>
```

Now restart the persistent servers in the first two windows and run `Directory1B.class` in the third window.

Window 1

```
>voyager 7000 -cd 7000.db
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:7000
database = 0 objects, 0 classes
>voyager 7000 -d 7000.db
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:7000
database = 1 object, 0 classes
```

Window 2

```
>voyager 8000 -cd 8000.db
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:8000
database = 0 objects, 0 classes
>voyager 8000 -d 8000.db
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:8000
database = 2 objects, 0 classes
```

Window 3

```
>java examples.directory.Directory1A
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.238:1176, root = \
created and saved {calcium->Ca, gold->Au}
created and saved {calcium->20, gold->79}
created and saved {number->Directory( 208.6.239.238:8000/30-...-192 )}
>java examples.directory.Directory1B
voyager(tm) 1.0.0, copyright objectspace 1997
address = 208.6.239.200:1200
symbols = {gold->Au, calcium->Ca}
numbers = {gold->79, calcium->20}
symbol/calcium = Ca
number/calcium = 20
add symbol/silver -> Ag
symbols = {gold->Au, silver->Ag, calcium->Ca}
symbol/silver = Ag
remove symbol/silver
symbol/silver = null
symbols = {gold->Au, calcium->Ca}
>
```

Application voyager1.0.0\examples\directory\Directory1A.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.directory;

import COM.objectspace.voyager.*;
import COM.objectspace.voyager.directory.*;

public class Directory1A
  {
  public static void main( String args[] )
    {
    try
      {
      // create persistent directory in local server 7000
      VDirectory symbols = new VDirectory( "localhost:7000/Symbols" );
      symbols.liveForever(); // prevent garbage collection
      symbols.put( "calcium", "Ca" );
      symbols.put( "gold", "Au" );
      symbols.saveNow(); // save copy to database
      System.out.println( "created and saved " + symbols );

      // create persistent directory in local server 8000
      VDirectory numbers = new VDirectory( "localhost:8000/Numbers" );
      numbers.liveForever(); // prevent garbage collection
      numbers.put( "calcium", new Integer( 20 ) );
      numbers.put( "gold", new Integer( 79 ) );
      numbers.saveNow(); // save copy to database
      System.out.println( "created and saved " + numbers );

      // create peristent higher-level directory in local server 8000
      VDirectory chemistry = new VDirectory( "localhost:8000/Chemistry" );
      chemistry.liveForever(); // prevent garbage collection
      chemistry.put( "number", numbers );
      chemistry.put( "symbol", symbols );

      // persist highest level directory
      chemistry.saveNow(); // save copy to database
      System.out.println( "created and saved " + chemistry );

      Voyager.shutdown();
      }
    catch( VoyagerException exception )
      {
      System.err.println( exception );
      }
    }
  }
```

Application voyager1.0.0\examples\directory\Directory1B.java

```
// Copyright(c) 1997 ObjectSpace, Inc.

package examples.directory;

import COM.objectspace.voyager.*;
import COM.objectspace.voyager.directory.*;

public class Directory1B
  {
  public static void main( String args[] )
    {
    try
      {
      // connect to highest level directory
      VDirectory chemistry =
        (VDirectory) VObject.forObjectAt( "localhost:8000/Chemistry" );

      // access various directories
      System.out.println( "symbols = " + chemistry.get( "symbol" ) );
      System.out.println( "numbers = " + chemistry.get( "number" ) );

      // access various elements
      System.out.println("symbol/calcium = "+chemistry.get("symbol/calcium")
);
      System.out.println("number/calcium = "+chemistry.get("number/calcium")
);

      // add an element and remove an element
      System.out.println( "add symbol/silver -> Ag" );
      chemistry.put( "symbol/silver", "Ag" );
      System.out.println( "symbols = " + chemistry.get( "symbol" ) );
     System.out.println( "symbol/silver = " + chemistry.get( "symbol/silver"
) );
      System.out.println( "remove symbol/silver" );
      chemistry.remove( "symbol/silver" );
     System.out.println( "symbol/silver = " + chemistry.get( "symbol/silver"
) );

      // display and then persist new symbols sub-directory
      System.out.println( "symbols = " + chemistry.get( "symbol" ) );
     ((VDirectory) chemistry.get( "symbol" )).saveNow(); // persist new entry

      Voyager.shutdown();
      }
    catch( VoyagerException exception )
      {
      System.err.println( exception );
      }
    }
  }
```