Voyager Core Developer's Guide

Version 1.1 for Voyager 8.0

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

Table of Contents

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

<This page intentionally left blank>

Introduction

Overview

Distributed application development has historically been difficult, complex and tedious. Stub and skeleton classes; binary incompatibilities; networking issues such as communication latency and bandwidth limitations; and many other issues combine to make distributed development a challenge for any development organization.

Advances in recent years have reduced many of these challenges. Common Object Request Broker Architecture (CORBA) and Web Services have given us standard APIs and messaging protocols; network reliability and bandwidth have improved dramatically; the Java Virtual Machine has ushered in runtime cross-platform binary compatibility for code and data; the Microsoft .NET Common Language Runtime (CLR) followed suit with similar benefits for C#, VB.NET, and C++/CLI; and advances in object-oriented design and design patterns for distributed computing have improved our understanding of distributed application design. **Recursion Software Voyager** builds on these developments, adds additional features, and ties them all together, setting a new standard for distributed application development.

The driving goal behind Voyager is to support the development of Java and .NET based distributed applications and make them easier to design, develop and deploy. Voyager provides a rich set of services and features for simplifying and speeding distributed application development, which have never before been available in a single product.

Some of Voyager's functionality – such as distributed garbage collection (DGC) – operates automatically in the background, completely transparent to you. Voyager's APIs are easy to learn and use. This ease of use means you can focus on your application logic instead of low-level "plumbing" work which other distributed development tools require you to handle manually. Finally, Voyager's advanced capabilities, flexibility, and extensibility give you the freedom to design applications based on your needs. You can fit Voyager to your architecture instead of contorting your architecture to fit Voyager.

Preface

This manual provides detailed information about the features available in Voyager. This guide assumes basic knowledge of distributed computing concepts and familiarity with the Java language or a .NET programming language.

This preface covers the following topics:

- Definitions

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

- Voyager installation and development requirements
- Voyager installation directories
- Deploying Voyager applications
- Contacting technical support

Definitions

JME — *Java Micro Edition.* In reference to running Voyager this term implies a supported version and configuration for the JME.

JSE — *Java Standard Edition.* In reference to running Voyager this term implies a supported version and configuration for the JSE.

.NET — *Microsoft .NET Framework.* In reference to running Voyager this term implies a supported version and configuration for a .NET Common Language Runtime environment.

VM — *Virtual Machine.* This term refers generically to a supported Voyager execution environment – either a Java Virtual Machine or a .NET Common Language Runtime environment.

Voyager Installation and Development Requirements

Before you install Voyager, ensure that you have:

- A Java Standard Edition runtime environment 1.4.2 or later for installation, and execution of the VOYAGER Wizard, and for Java development. You can download the JDK from java.sun.com free of charge.
- A Java Development Kit (JDK) 1.4.2 or later is required for Java development.
- The Microsoft .NET Common Language Runtime 2.0 is required for .NET development (C#, VB.NET, or C++/CLI). You can download the .NET Framework 2.0 from microsoft.com/downloads free of charge.
- A basic understanding of distributed computing concepts and a working knowledge of Java or C#.
- Approximately 200MB of available space on your hard drive to install Voyager.

Voyager Installation Directories

The directory structure of the Voyager install directory is shown in the following table.

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

`.\`	Voyager readme, install, changes, environment, copyright, and license text files.
`bin\`	Utilities and other binary files.
`doc\`	Developer guides and user guides.
`examples\`	Example files organized by programming language / environment.
`license\`	Licenses for 3rd-party products Voyager uses.
`platform\android\`	Android libraries
`platform\cdc\`	JME CDC API documentation and libraries
`platform\cldc\`	JME CLDC/MIDP 2.0 API documentation and libraries.
`platform\dotNET\`	.NET (via ikvm) API documentation and assemblies. (deprecated)
`platform\iphone\`	iPhone API documentation and assemblies.
`platform\jse\`	API documentation for JSE and `.jar` files for Java Standard Edition. Includes 3rd-party files.
`platform\windows-dotnet\`	.NET API documentation and assemblies.
`platform\windows-mobile\`	Compact Framework API documentation and assemblies.

Deploying Voyager Applications

Once you have written a Voyager application selected installed files will be needed for your deployment. See the detailed discussion in Deployment.

Contacting Technical Support

Recursion Software welcomes your problem reports and appreciates all comments and suggestions for improving Voyager. Please send all feedback to the Recursion Software Technical Support department.

Technical support for Voyager is available via email and phone. You can contact Technical Support by sending email to psupport@recursionsw.com or by calling (972) 731-8800.

Note: When submitting an issue via email, if you have a Customer Support ID be sure to include it on the first line of the message body.

Benefit and Feature Summary

Voyager Features

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

Voyager offers developers a complete set of features for distributed application development. The following sections describe the most important Voyager features.

Remote-Enabling of Classes

Java and .NET classes are automatically remote-enabled at runtime. A class does not have to be modified in any way, and no additional files are necessary to remote-enable a class. Thus, there is no difference between a "regular" Java/.NET class and a remote-enabled class. Classes may also be explicitly remote-enabled by declaring them to implement `com.recursionsw.ve.IRemote`.

Remote Object Construction

With a single method call, you can create a remote instance of any class on any Voyager VM and obtain a proxy to the newly created object. For some runtime environments the proxy class is generated dynamically if it does not exist, eliminating the need to manually generate this class. Since the proxy implements the same interfaces as the object, by using interface-based programming techniques you do not need to modify any code to work with remote objects.

Dynamic Class Loading

Classes can be dynamically loaded from one or more locations when necessary. This allows you to easily set up class repositories for Java and assembly repositories for .NET that serve your corporate applications, simplifying deployment and maintenance.

Remote Messaging

Method calls to a remote proxy are transparently forwarded to its object (referent). If the object is in a remote VM, the method arguments are serialized to the destination. The morphology of the arguments is maintained. If an object's class implements `com.recursionsw.ve.IRemote`, the object is passed by reference, meaning a proxy to the object is sent. If an object's class implements `com.recursionsw.ve.VSerializable` (or `java.io.Serializable`), it will be passed by value; the remote method will receive a copy of the original object. Objects that implement none of these interfaces are passed by reference by default.

Remote Exception Handling

If a remote exception occurs, it is caught at the remote site, returned to the caller, and rethrown locally. If the appropriate logging level is selected, a complete stack trace is displayed to the console.

Distributed Garbage Collection

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

The distributed garbage collector (DGC) automatically reclaims objects when there are no more remote references to them. This eliminates the need to explicitly track remote references to an object. The DGC mechanism uses an efficient "delta pinging" algorithm to minimize the traffic required for distributed garbage collection. You can also fine-tune the behavior of the distributed garbage collection mechanism and receive notification of DGC events.

Dynamic Aggregation™

Dynamic aggregation presents a fundamental step forward for object modeling and complements the traditional mechanisms of inheritance and polymorphism. This feature allows you to dynamically add secondary objects (termed facets) to a primary object at runtime. For example, you can dynamically add hobbies to an employee, a repair history to a car, or a payment record to a customer. The code for the primary object is decoupled from the code for its facets, simplifying your object model.

SOAP and WSDL Support

Voyager provides full support for exposing and accessing SOAP services. Voyager also provides dynamic WSDL generation for description of Web Services exposed, and a proxy generator for accessing remote WSDL described services.

Object Mobility

You can easily move any serializable object among Voyager VMs at runtime. Voyager automatically tracks the current location of the object. If a message is sent from a proxy to an object's old location, the proxy is automatically updated with the new location and the message is re-sent. Mobility is often useful when optimizing message traffic in a distributed system.

Autonomous Intelligent Mobile Agents

Voyager provides a framework for creating agent-oriented distributed applications. Voyager supports the creation of mobile, autonomous agents that can be deployed to a VM and execute on arrival. Agents can also move themselves between VMs and continue to execute upon arrival at a new location. Complex intelligent behavior can be written using the Voyager Wizard to construct rules that can run in a remote VM.

Task Management

Voyager uses a task management framework to balance workload and prevent the VM from being overloaded by threads.

Advanced Messaging

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

You can send one-way, synchronized, and future messages. One-way invocations return to the caller immediately after sending the message; any return value or exception is discarded. Future messages immediately return a placeholder to the result, which may then be polled or read in a blocking fashion.

Security

An enhanced Security Manager is provided, as well as hooks for installing custom sockets such as SSL.

Naming Service

Voyager's naming service provides a single, simple interface that unifies access to standard naming services. New naming services can be dynamically plugged into Voyager's naming service.

Yellow Pages Directory

Voyager's yellow pages directory complements the Naming Service. It supports lookup of a service based on one or more attributes or characteristics. The location and identity of the service does not need to be known at lookup time.

Multicasting

You can multicast a message to a distributed group of objects without requiring the sender or receiver to be modified in any way.

Publish-Subscribe

You can publish an event on a specified topic to a distributed group of subscribers. The publish-subscribe facility supports server-side filtering and wildcard matching of topics.

Timers

A `Stopwatch` and `Timer` class facilitate common timing chores. Timer events can be distributed and multicast if necessary.

Multi-home Support

Voyager supports multi-homed systems. A multi-homed system is one with multiple hostnames or addresses.

Connection Management

Connection management services allow you to manage the number of live and idle connections for a Voyager server to prevent server/client overload.

LDAP Authentication

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

Voyager supports authentication of clients to an LDAP Directory Server such as OpenLDAP, Apache Directory Server, or Microsoft Active Directory. Voyager clients provide a principal consisting of an identity (e.g. username) and security token (e.g. password), and the Voyager server uses these values to authenticate the client at connect time.

Voyager in .NET and Java

Note: Often Voyager is deployed to the same kind of VMs, e.g., a distributed application is run either on all Java VMs or all .NET VMs. For these homogenous deployments this section is not required reading. Architects and developers are encouraged to read this section if Voyager will be used in a mixed VM environment, i.e., both Java and .NET versions of Voyager being used.

Voyager maintains the same behavior across Java (JSE and JME) and .NET runtime environments. This is accomplished through a few key practices.

- Shared object model between Java and .NET. For example, Object, String, and Exception are mapped to their equivalents in the other environment.
- Common application programming interfaces. The Voyager APIs for Java and .NET are the same within the limits of the differing languages.

It is important to note that Voyager is native in both Java and .NET versions. The .NET version of Voyager does not require a Java VM since Voyager is built as a native CLR assembly.

The Interfaces Must Be Identical

For Voyager the interfaces used must be the same on all communicating Voyager programs.

Note: "Identical" means that the interfaces used have the same fully qualified name and the same contents – methods, method names and signatures. The fully qualified name on .NET consists of the namespace and the interface name while on Java the fully qualified name consists of the package name and the interface name.
Note: Some programming languages allow instance data within an interface. Interfaces used within a Voyager network must **not** use instance data.

This "same interface" requirement is true in a homogenous deployment – if an interface is changed all of its existing `.class` files must be updated in a Java Voyager network. It is also true in a mixed deployment consisting of both Java and .NET Voyager nodes.

Note: Voyager can be configured to load classes from a common location. See Serving Classes.

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

In a mixed .NET and Java environment there are two strategies for maintaining consistent interfaces for the whole Voyager:

1. Have two matching source code files for the interface, one written in Java and one written in C# (or another .NET language). In this case the Java package name must match the .NET namespace name. The interface names must match. The method count and method names and signatures must also match.

2. Have a single source code file in Java for the interface and use a utility tool to generate a .NET assembly. See the `cligen` tool in Appendix B – Utilities.

If the second approach is chosen, the generated assembly can be referenced and inherited from within a .NET language so the implementation of the interface can be native .NET.

Type Equivalence

The requirement of having the interface shared among Voyager nodes is one central issue. However, *all* serializable types (classes and interfaces) that are used within these shared interfaces must also be shared on both sides. This is because the receiving Voyager will deserialize the sent object into a new instance of that type in the native language for that environment.

Note: Voyager automatically handles equivalence between `java.lang.Object` and `System.Object`, `java.lang.String` and `System.String`, `java.lang.Exception` and `System.Exception`. As a result, Object, String, and Exception objects do not need special handling.

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

Voyager Features

Overview

This chapter describes the basic operation and usage of Voyager.

In this chapter, you will learn to:

- Start and stop a Voyager program
- Start a Voyager server from the command line
- Understand and use Voyager's dynamic class loading ability
- Serve classes to remote Voyager VM's
- Use threads
- Understand Voyager's support for object persistence
- Use Voyager on multi-homed systems
- Use Voyager's security manager

Starting and Stopping a Voyager Program

A program must invoke one of the following variations of `Voyager.startup()` before it can use any Voyager features.

```
startup()
```

Starts Voyager as a client that initially does not accept incoming connections from remote programs. The application is free to start one or more server contexts as needed.

```
startup( String serverName, String serverUrl )
```

Starts Voyager with a single server context that accepts incoming connections on the specified URL.

```
startup( Object object, String serverName, String serverUrl )
```

Start a Voyager applet or servlet as a server that accepts incoming messages on the specified URL

```
startup(ClassLoader loader)
```

Start Voyager as a client that doesn't initially accept incoming

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

messages, supplying a custom classloader

or

```
startup(ClassLoader loader, String serverName, String serverUrl)
```

This form is required when running Voyager in a Java environment under a custom classloader, such as a JEE EJB or servlet container.

All variations of startup return VoyagerContext, which is the context used by the application to reference Voyager.

```
startFromDirectory( String directoryUrl )
```

Starts a Voyager server from the information provided in the server profile specified in the URL. It is the API equivalent of starting Voyager with the `-d` command line option.. The URL specified is typically a Voyager server that uses the `-f` parameter to persist server profile information.

The general format of a URL (Universal Resource Locator) follows:

```
protocol://host:port/file;argument#reference
```

Each part of the URL is optional. For simplicity and readability, the Voyager documentation and examples typically use only the port (`//:8000`) or host:port (`//dallas:7000`). However, to minimize hostname resolution problems it is recommended to use the fully qualified hostname or IP address of the system. Be sure to consistently use the same identification for a system; do not use the hostname in one place and the IP address in another (`//dallas` versus `//10.2.2.250`), or the fully qualified hostname in one place and the hostname in another (`//dallas.recursionsw.com` versus `//dallas`). If you do this, Voyager may not be able to recognize that the different URLs are for the same Voyager process. This typically results in a `Connection refused` or `Address in use` exception message. A complete description of the URL format follows:

`protocol`	The `protocol` is the transport protocol. If unspecified, the default protocol (normally `tcp`) will be used.
`host`	The `host` is the hostname or IP address of the system. The hostname may be unqualified (`//dallas`) or fully qualified (`//dallas.recursionsw.com`) or `//localhost`. If `//localhost` is specified, VOYAGER attempts to resolve the system's hostname by calling `java.net.InetAddress.getLocalHost().getHostName()`. Because this may not return the system's fully qualified hostname, it is not recommended to use "`//localhost`" for the host.
`port`	The `port` specifies the port number of the startup server.
`File`, `;argument`, `#reference`	These parts of an URL are rarely used with Voyager, but are presented for completeness.

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

When Voyager is started as a server, it will begin listening on the URL specified on the command line or in the call to `Voyager.Startup( name, Url )`. A Voyager VM can accept connections on multiple URL's, however. The application simply creates a server context and tells the server context the URLs on which to listen. If the system is multi-homed with multiple hostnames, you can either explicitly specify the hostname or omit it and allow the operating system to determine the primary hostname.

Examples of starting Voyager programmatically follow:

```
VoyagerContext voyagerContext = Voyager.startup(); // startup as a
client
VoyagerContext voyagerContext = Voyager.startup( "my server", "//:8000"
); // startup as a server on port 8000
VoyagerContext voyagerContext = Voyager.startup( "my server",
"//dallas:7000" ); // startup as server on port dallas:7000
VoyagerContext voyagerContext = Voyager.startup( "my server",
"//10.2.2.20:7000" ); // startup as server on port 10.2.2.20:7000
```

To shut down Voyager, invoke voyagerContext`.shutdown()`. This method terminates the Voyager internal non-daemon threads, closes all open connections, and shuts down all server and client contexts. Daemon threads may continue to run, but the application can be terminated safely at this point.

You can use `voyagerContext.addSystemListener()` to listen to the events generated by the startup and shutdown.

Starting a Voyager Server from the Command Line

To start a Voyager server from the command line, use the `voyager` utility. This utility starts a Voyager server that accepts connections and messages on a given URL. To stop a Voyager server, press `Control-C`. Because this immediately terminates the process, it should not be used on a deployed system unless the application is in an unrecoverable state or can safely be terminated.

For example, to start a Voyager program running in a Java VM that accepts connections on port `8000`, enter the following:

```
>voyager //:8000
Voyager 8.0, Copyright 2009 Recursion Software, Inc. ...
```

The `voyager` utility has several options. For more information, refer to Utilities.

Loading Classes in Java

Note: This section applies to Voyager on a Java VM.

The Java VM loads and links classes dynamically at runtime. The VM's `bootstrap classloader` is typically capable of loading classes from the local filesystem through the

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

`classpath` environment variable. The JSE runtime also includes a classloader, which can load classes from `.jar` or `.zip` files in the `jre\lib\ext\` directory of the JSE installation.

A Voyager application usually requires additional classloading capability to provide dynamic proxy class generation and remote classloading capability. This is provided for and managed by Voyager's `com.recursionsw.ve.ClassManager` class, a custom classloader that extends `java.lang.ClassLoader`, and `resource loaders`, which are capable of providing classes and other resources to the Voyager classloader as a `java.io.InputStream`. The Voyager classloader manages all resource loaders, and `ClassManager` contains static methods that allow classes and any other resource to be loaded using Voyager's classloader.

Resource loaders implement the interface `com.recursionsw.ve.loader.IResourceLoader` and provide classes and other resources to the Voyager classloader. By default, a Voyager program has a single pre-installed `ProxyResourceLoader` at priority level 5. This loader supports dynamically generating and loading proxy classes. Additional resource loaders are available for resource loading from an URL, a directory in the local filesystem, or a .jar file. You may also create custom resource loaders. All resource loaders must be registered with the Voyager classloader (`com.recursionsw.ve.loader.VoyagerClassLoader`). For example, to enable class loading from a specific URL, add an `URLResourceLoader` constructed on an URL as follows:

```
VoyagerClassLoader.addURLResource("http://dallas:9000/");
```

The format for the URL is as follows:

- For classes in a directory not in the `CLASSPATH`, use `file:///full/directory/path/` (three forward slashes are intentional)
- For classes on a web server, use `http://host:port/root/`, where `root/` is the root directory for requests
- For classes on an HTTP-enabled Voyager VM, described in Serving Classes, use `http://host:port`

A Voyager program attempts to load resources according to the following sequence.

1. Search the `CLASSPATH`.
2. Search the `resource loaders` installed with the `VoyagerClassLoader` from highest priority to lowest priority.

Note that resource servers are not classloaders themselves. They only provide resources to the Voyager classloader.

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

Note: If working with Agents, do not use localhost as the host for a URL ResourceLoader (resource URL). Use the actual machine name or IP address.

To disable Voyager's use of resource loaders (including the dynamic proxy generator), and rely solely on Java's class loader, invoke `VoyagerClassLoader.setResourceLoadingEnabled( false )`. Note that this is the default in runtime environments unable to dynamically load classes, e.g., Android's Dalvic VM.

If Voyager is started from the command line, use `-c URL` to add an `URLResourceLoader` on the specified URL. The `-c` option may be specified many times on the same command line. For example, to start a Voyager server that can load classes from the directory `/bin/classes/` and from an HTTP server running on `//dallas:8000`, enter the following as a single line:

```
voyager //:8000 -c file:///bin/classes/ -c
http://dallas:8000
```

To manually load a class using Voyager's class loading machinery, use `ClassManager.getClass( classname )` instead of `Class.forName( classname )`.

Loading Classes in .NET

Note: This section and the `voyager_net.exe` utility are deprecated as are the DLLs referenced by `voyager_net.exe`. Instead, please use a Visual Studio project to launch your application. Refer to the examples solution that is provided with Voyager for examples.

Understanding Assembly Loading

For .NET development classes are built into *Assemblies*, either a .DLL or .EXE file, which are then loaded making the classes available to the .NET runtime. Here are some important steps in loading an assembly, `Basic2A.exe`, which was built with reference to `Stockmarket.dll`.

1. If a .config file with the same name as the executable, e.g. `Basics2A.exe.config`, is available in the same directory then it will be used for locating referenced assemblies

2. Any assemblies that are not signed must be in the same directory as `Basics2A.exe` but signed assemblies may be configured in `Basics2A.exe.config` with a location external to the executable's directory.

Note: The prebuilt assemblies for Voyager are signed. Example code that builds into assemblies, e.g. `Stockmarket.dll`, are not signed.

3. The operating system evaluates all *static* references beginning with `Basic2A.exe`, including `Stockmarket.dll` and eventually reaching `mscorlib,` which is the .NET runtime library.

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

4. Each of the assemblies may reference `mscorlib`, which is the .NET runtime library and some assemblies may have been built with reference to .NET 1.1 Framework while others were built with reference to .NET 2.0.

5. An installed .NET runtime CLR is launched which is capable of handling the highest version of `mscorlib` reference by any assembly. If all static references to `mscorlib` are for .NET 1.1 then the the .NET 1.1 CLR may be launched by the operating system even if the .NET 2.0 CLR is also installed on the machine.

6. Dynamically loading assemblies into a running CLR that are built for a later version of .NET will cause an error.

Note: Voyager assemblies reference the .NET Framework 2.0, which should always result in a 2.0 CLR (or later) being launched. Since dynamic class loading is used by Voyager it is important to note that assemblies loaded dynamically do not influence the CLR selected to run Voyager. For example,

```
> voyager_net 8000 -ad ..\examples\csharp\stockmarket
```

can fail if the .NET 2.0 CLR is chosen to launch Voyager (based on the static references of `voyager_net.exe`) and the .DLL assemblies in the stockmarket directory were built with .NET 3.0.

Loading Classes with `voyager_net`

If you use the `voyager_net` generic server it will not have any static reference to your application classes. Two command line options, `-ad` and `-a` are added to provided dynamic loading of assemblies. The `-ad` option will load all .DLL assemblies in the specified directory. For example,

```
> voyager_net 8000 -ad ..\examples\csharp\stockmarket
```

will start the `voyager_net` server and load all the .DLL assemblies in the `..\examples\csharp\stockmarket` directory. The `-a` option specifies an assembly by name that will be loaded. Use this option if you need to load an .EXE assembly. For example,

```
> voyager_net 8000 -a ..\examples\csharp\mytest\mytest.exe
```

Multiple `-ad` and `-a` options can be specified to load all of the required classes.

Serving Classes

A Voyager program can serve other programs with any resource that it can load. This is made possible by Voyager's built-in HTTP capability, which is disabled by default for security reasons. To HTTP-enable a Voyager program, invoke `ClassManager.enableResourceServer()`.

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

If Voyager is started from the command line, use the `-r` option to enable its HTTP server.

The security example demonstrates a Voyager client capable of loading classes from a remote Voyager server that is HTTP-enabled. It also demonstrates some of the security consequences that you may want to address when using this form of class loading.

Task and Thread Management

To reduce the significant overhead of creating and destroying threads, Voyager uses a task manager and thread pool. When Voyager needs to run a task in a different thread, Voyager schedules the task with its task manager. The task manager uses threads from an internal thread pool to run tasks. If the task manager needs a thread, it only creates a new Thread object when all threads in its thread pool are busy. When the thread finishes, it is added back to the pool. The task manager thread pool has a maximum size; if the pool is at its maximum size, and all threads are busy, then the task manager will queue up tasks until a thread becomes available.

The task manager thread pool is initially empty and has an infinite maximum size. To change the maximum pool size, invoke `TaskManagerConfigurator.setMaxUserThreads()`, or use the `-max_user_threads` option when using the `voyager` utility.

Voyager employs a task management framework to help balance workload and prevent the VM from being overrun by threads when the task list becomes large.

Snapshot

Facilities that save and restore objects need to access the complete state of an object, including its facets (as described in Dynamic Aggregation™) and Voyager-related properties, such as the object's export address. The `Snapshot` class provides this functionality.

To persist an object and all of its state to a database, obtain a `Snapshot` of the object and then either store the `Snapshot` directly into the database or store its parts individually. To obtain a `Snapshot` of an object, use `Snapshot.of( object )`. The fields are set to each part of the object's state and can be accessed using the following methods.

`getObject()` – Returns the object.
`getProperties()` – Returns an instance of `Properties` containing the object's Voyager-related properties.
`getFacets()` – Returns an array of the object's facets.

To load an object and its associated state back into memory, first recreate the original `Snapshot` either by loading it directly from the database or by loading the individual parts and then using `Snapshot.from( object, properties, facets )`. Then invoke

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

`Snapshot.restore()`, which recreates the original object state from the individual fields of the `Snapshot` and returns a proxy to the newly created object.

Multihome Support

Multihomed computers are machines that are configured with more than one host name or IP address. Voyager supports multihomed computers. When a Voyager server context's server is started, i.e., when a `ServerContext`'s `startServer(String url)` method is called on a `ServerContext` instance, it is running on a host name or IP address specified in the URL. To make Voyager aware of another host name or IP address, simply invoke the instance's `startServer(String url)` again with the additional URL. See the API documentation for `com.recursionsw.ve.ServerContext` for more details.

For example, suppose you have a machine with host name `abc.mycorp.com` and another host name `xyz.mycorp.com`. Voyager starts with `abc.mycorp.com`. To make Voyager aware of the host name `xyz.mycorp.com`, use `startServer()`.

Java

```
import com.recursionsw.ve.Voyager;

public class Example
  {
  public static void main(String[] args)
    {
    VoyagerContext voyagerContext = null;
    voyagerContext  = Voyager.startup( "server 1",
                      "//abc.mycorp.com:8000" );
    ServerContext s1 = voyagerContext.acquireServerContext("server 1");
    s1.startServer( "//xyz.mycorp.com:8000" );
    // ...
    }
  }
```

C#

```
using com.recursionsw.ve;

public class Example
  {
  public static void Main(String[] args)
    {
    VoyagerContext voyagerContext = null;

    voyagerContext = Voyager.startup( "server 1",
                     "//abc.mycorp.com:8000" );
    ServerContext s1 = voyagerContext.acquireServerContext("server 1");
    s1.startServer( "//xyz.mycorp.com:8000" );
    // ...
    }
  }
```

The Voyager server is running on port `8000` of `abc.mycorp.com`. It is also known to be running on the same port of `xyz.mycorp.com`.

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

Security

Voyager includes support for the standard Java security manager system. Java applications have no security manager by default, so objects may perform any type of operation. Voyager includes a security manager called `VoyagerSecurityManager`, which you can install at the start of a program to restrict operations.

Installing a Security Manager

You have the option of installing a security manager in a Voyager program. After it is installed, the security manager is active for the duration of the program, and 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. The following section lists legal operations by environment. 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 security exception is thrown.

The Voyager security manager distinguishes between native and foreign execution threads. Foreign execution threads are operating within the context of a remote invocation; that is, the method is being invoked by a remote proxy.

The Voyager security manager allows native execution threads to perform any operation but selectively restricts foreign threads by operation.

Note: You can modify or extend the Voyager security manager behavior by extending the `VoyagerSecurityManager` class.

You can install a Voyager security manager in one of two ways.

1. To start a Voyager server with a Voyager security manager, execute `voyager` with the `-s` (security) option.

2. To install a Voyager security manager in a Voyager program, call the `System.setSecurityManager( new VoyagerSecurityManager() )` method.

Note: When using the `VoyagerSecurityManager` in JSE, the `java.io.SerializablePermission enableSubstitution` must be granted to the Voyager codebase. For more information on the JSE security model, security permissions, and how to configure them, please refer to the JSE documentation.

The Security1 Example demonstrates the use of Voyager's security manager to restrict operations by foreign objects.

Identifying Object Authority

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

The following table lists the operations allowed by the JDK `SecurityManager` and indicates those that `VoyagerSecurityManager`, which extends `SecurityManager`, allows an object to perform as an object as a native thread and as a foreign execution thread.

Operation	Native	Foreign
Accept connections from any host	**x**	**x**
Connect to any host	**x**	**x**
Listen on any port	**x**	**x**
Perform multicast operations	**x**	**x**
Set factories	**x**	
Manipulate threads	**x**	
Manipulate thread groups	**x**	
Execute a process	**x**	
Exit the program	**x**	
Access AWT event queue	**x**	**x**
Access the system clipboard	**x**	
Create windows	**x**	**x**
Create class loader	**x**	
Delete files	**x**	
Read files, excluding socket file descriptors	**x**	
Write files, excluding socket file descriptors	**x**	
Access security APIs	**x**	
Link to a dynamic library	**x**	
Access private/protected data and methods	**x**	
Access packages	**x**	**x**
Define classes in packages	**x**	**x**
Print	**x**	
Manipulate properties	**x**	**x**

The `VoyagerSecurityManager` defines a `checkMethodAccess()` method not included in its parent JDK class, `SecurityManager` . The `checkMethodAccess()` method prevents a foreign object from calling the following Voyager methods.

Class	Method
`Voyager`	`shutdown()`

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

	`addSystemListener()` `removeSystemListener`

`ClassManager`	`setParentClassLoader()`
	`enableResourceServer()`
	`resetClassLoader()`

`VoyagerClassLoader`	`addResourceLoader()`
	`removeResourceLoader()`
	`setResourceLoadingEnabled()`

To change the methods disallowed by `checkMethodAccess()`, refer to the Voyager API documentation.

Audit Service

The **Audit** Service supports two business needs.

1. Holding users accountable for their use of the system; and
2. Detection of security violations, i.e., detecting attempts by unauthorized users to access the system and detecting attempts by authorized users to exceed the limits of their authorization.

The Audit subsystem supports these needs by enabling the reporting and recording of events with security implications regardless of which node in the distributed system detected the event.

Holding a user accountable involves recording user actions related to security-sensitive data items, i.e., recording creation of a new record, updating a field, changing a configuration item, etc. These are usually acceptable actions, and are recorded by the audit system to enable answering questions such as, “Who last changed that field?” or “Who looked at that field in a record in question over the past 30 days?”

Security violations of interest include the obvious, such as attempts to break into a system by guessing a password, or an attempt to connect to an application from an unauthorized location, and the less obvious. Less obvious violations include an authorized user’s attempt to retrieve a information the user is not authorized to see, e.g., an employee attempting to review the employee’s supervisor’s salary history even though the employee is authorized to see the employee’s subordinates’ salary histories.

The Voyager Audit Service draws specifications and design elements from the Open Group’s Preliminary Specification, Distributed Audit Service (XDAS)[1] but is not a full implementation of the XDAS specification.

[1] The Open Group. Preliminary Specification, Distributed Audit Service (XDAS). The Open Group, 1997.

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

In Voyager's distributed implementation, the audit system at each node is configured to forward audit events to neighboring nodes, as well as (optionally) recording the audit events locally. Neighboring nodes can be in a parent-child relationship or a peer-to-peer relationship.

Motivation

The current Voyager logging capability is limited to writing to the console of the host node, and lacks the regular record structure necessary to easily parse the logged records. The audit system provides both the consistent event record format needed to support straightforward record parsing and processing, and the ability to configure the audit subsystem to forward audit events to local or remote audit event handling services. The audit event handlers are capable of writing the event records to the console, an audit event file, or in any other manner needed by the organization.

To provide flexible audit event processing, the audit subsystem on a Voyager node can be configured to deal with an audit event in any combination of the following ways.

- Forward the event to a "parent" audit handling node. A "parent" node is one to which the "child" sends audit events, but which never sends audit events to the child. An edge device will typically be configured as a child to one or more parent devices.

- Forward the event to a "peer" audit handling node, i.e., a node to which this node sends audit events and from which this node receives audit events. Peers might be hosts acting as management consoles, where all active consoles are to receive all the audit event records.

- Handle the event locally. The event can be sent to local audit event handlers, which are implemented as Voyager services. A handler service that writes audit event records to the local console is installed by default. An optional handler service writes audit records to the local file system. If routed through the console log handler, event display can be filtered to eliminate uninteresting events. Obviously a single Voyager instance must locally handle audit events, but local handling also applies to any Voyager node that needs to write the stream of audit events to permanent storage.

Structure

The Audit subsystem consists of one required service, the Audit Service, and zero or more optional services. All code for the Audit subsystem lives in the `com.recursionsw.ve.audit` package. The Audit Service, realized by `AuditService`, receives all audit events, regardless of their source, and hands each event to zero or more local audit event record services, zero or more remote parents, or zero or more remote

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

peers. The local services, which register listeners with the Audit Service, implement audit event handling for the local Voyager node. Additional optional services can be easily written and dynamically installed, e.g., one that writes to a Unix systems syslog.

The current local audit handler services are the Audit Console Logger and the Audit File Logger.

1. The Audit Console Logger (`AuditServiceConsoleLogService`) writes audit event records to the Voyager console. This service is, by default, installed and started, but can be disabled before or after Voyager starts. This service implements user-configurable filters to control which events get displayed.

2. The Audit File Logger (`AuditServiceFileLoggerService`) writes audit event records to a local log file. This service is capable of changing to a new log file without losing any audit records, but doing so requires a call from an external source to initiate the change. The Audit File Logger service is not normally loaded. Simply touching the class, e.g., setting a configuration option, will install and initialize it.

Both of these services extend `AbstractAuditLogger`, a public abstract base class for all audit record logging services. `AbstractAuditLogger` implements the audit record event listener that is registered with `AbstractService`, a count of records seen by the listener, and methods to enable and disable the listener. `AbstractAuditLogger` also defines `logRecord(AuditRecord)`, the abstract method subclasses implement to handle audit record events. Subclasses can, if necessary, override `getPrerequisites()`, the method declared in `com.recursionsw.ve.system.IService` that returns a list of services on which this service depends, and `boolean isActive()`, a method that returns `true` if the service is active.

Clients access the Audit service through the `Audit` class, which provides three factory methods, four methods for configuration, and one query method.

One of the factory methods returns an implementation of `IAuditSession`, where a session describes an entity authorized to insert records to the audit record stream. A session is generally created and retained for repeated use.

Two factory methods return an implementation of `IAuditRecord`, which contains the fields that describe the audit event. At creation time a record is associated with a session. One of the `IAuditRecord` factory methods takes a session argument and returns an empty audit record. The other `IAuditRecord` factory method takes arguments that completely fill in the audit record fields.

Three of the configuration methods deal with connecting and disconnecting parent and peer neighbors. The fourth provides the audit subsystem with an the name of a class implementing `IAuditPolicy`, i.e., an object describing audit service policies.

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

The following paragraphs describe audit system configuration, and creation and use of the session and record classes.

Using the Audit Service

The example program `Audit1` contains a richly commented example of using the Audit subsystem. The file is in `core/java/examples-src/examples/audit`.

The example program suite consisting of `Audit2a, Audit2aPeer,` and `Audit2b` illustrates parent, child, and peer Audit Service relationships. These files are also in `core/java/examples-src/examples/audit`.

Configuration and Initialization

The Audit Service and the optional logging services support configuration from a Voyager property file as well as by calling configuration methods.

Audit Service

Configuring the Audit Service is optional. Reasonable defaults exist for all the configurable elements.

Connection Policy

Realizations of the interface `IAuditPolicy` describe how to deal with missing connections to audit neighbors, i.e., parents, children, or peers. The class that realizes `IAuditPolicy` can be specified in the Voyager property file using either of the following lines. The class name must be fully qualified.

```
Voyager.audit.Audit.setConnectionPolicy=ClassName

com.recursionsw.ve.audit.Audit.setConnectionPolicy=ClassName
```

Java code to install the connection policy looks like the following line.

```
com.recursionsw.ve.audit.Audit.setConnectionPolicy(aClassName);
```

Parent Neighbors

An Audit Service parent receives all audit events originating at the child Voyager node, or originating at any Audit peer or Audit child of the child Voyager node. A parent node never sends an audit event to a child node. A parent of the current Voyager node is specified using a normal Voyager client context that points to the parent Voyager. A parent-child relationship is always established by the child. Specifying a parent node is optional.

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

Specify a parent using the following Java code. Configuring parents using the following call supports multiple parents.

```
com.recursionsw.ve.audit.Audit.connectParent(ClientContext context)
```

Peer Neighbors

An Audit Service peer receives all audit events originating at the Voyager node, or originating at any Audit peer Voyager node. A peer node receives audit events from neighboring peer nodes, as well as sending audit events to neighboring peer nodes. A peer of the current Voyager node is specified using a normal Voyager URL that points to the peer Voyager. Either peer can ask for creation of the peer relationship. Specifying a peer node is optional.

Specify a peer using the following Java code. Configuring peers using the following call supports multiple peers.

```
com.recursionsw.ve.audit.Audit.connectPeer(ClientContext context);
```

Console Audit Log Service

All configuration of this service occurs using static methods of the class `ConfigureConsoleLogger`.

The console logger supports filtering, i.e., selecting which audit records to display based on the audit record's outcome code.

The console logger filters operate at two levels. At the coarse level, output of audit records can be turned off and on based on the outcome code "class." The three outcome code classes are *success*, *failure*, and *denial*. A method exists in `ConfigureConsoleLogger` for each outcome class to turns on or off output of audit records having that outcome. For example, the following line of Java turns on output of records with successful outcome codes.

```
com.recursionsw.ve.audit.ConfigureConsoleLogger.printSuccessRecords(
        true);
```

The default configuration is equivalent to executing the following lines of Java code.

```
com.recursionsw.ve.audit.ConfigureConsoleLogger.printDenialRecords(
        true);
com.recursionsw.ve.audit.ConfigureConsoleLogger.printFailureRecords(
        true);
com.recursionsw.ve.audit.ConfigureConsoleLogger.printSuccessRecords(
        false);
```

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

The second, finer, filtering happens on individual outcome codes. Specification of a finer filter overrides or has higher priority than the coarse outcome filter. For example, with the default configuration a record with an outcome code of IAuditOutcome.AUDIT_OUTCOME_PRIV_GRANTED would not print, but turning on output of this code using the following line will cause the record to print.

```
com.recursionsw.ve.audit.ConfigureConsoleLogger.addFilter(
        IAuditOutcome.AUDIT_OUTCOME_PRIV_GRANTED, true);
```

Finally, the following line of Java will restore the filters to their default configuration.

```
com.recursionsw.ve.audit.ConfigureConsoleLogger.resetFilters();
```

File Audit Log Service

The class `ConfigureFileLogger` configures the File Audit Log Service. The File Audit Log Service supports configuration of the directory into which the audit log files are written, left or prefix portion of the file name, and the right or suffix portion of the file name.

Directory Name

The `ConfigureFileLogger` method `setDirectory()` configures the directory that will contains all the audit log files. The following line sets the directory to the default value, which is “temp”.

```
com.recursionsw.ve.audit.ConfigureFileLogger.setDirectory(
        IAuditServiceFileLoggerService.DEFAULT_DIRECTORY_NAME );
```

File Name Prefix

The `ConfigureFileLogger` method `setDefaultFileNamePrefix()`. The following line sets the prefix to the default value, which is “VEAuditLog”.

```
com.recursionsw.ve.audit.ConfigureFileLogger.setDefaultFileNamePrefix(
        IAuditServiceFileLoggerService.DEFAULT_FILE_NAME_PREFIX );
```

File Name Suffix

The `ConfigureFileLogger` method `setDefaultFileNameSuffix()` sets the trailing portion of the log file name. The following line sets the prefix to the default value, which is ”.txt”.

```
com.recursionsw.ve.audit.ConfigureFileLogger.setDefaultFileNameSuffix(
        IAuditServiceFileLoggerService.DEFAULT_FILE_NAME_SUFFIX );
```

Change Log File

The `ConfigureFileLogger` method `startNewLogFile()` causes the logger service to close the current file, if any, and create a new output file. The file will be created in the the directory named in the most recent call to `setDirectory()` using the file name

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

elements specified by the most recent calls to `setDefaultFileNamePrefix()` and `setDefaultFileNameSuffix()`. If the new file name already exists, including if the new name matches the current name, the file change request will be ignored. If the file open fails the current log file will continue to be used.

Create Session

As mention earlier, a session describes an entity that holds the right to add a record to the stream of audit records, as well as holding information common to multiple audit event records. Once created, an `IAuditSession` cannot be changed.

The class `AuditSession` is the realization of `IAuditSession`. A user does not have access to `AuditSession`. The factory method in `Audit` builds the instances.

The factory method `Audit.acquireSession()`, which produces realizations of `IAuditSession`, requires the following arguments.

- Authority: The originating authentication authority, usually the name of a server, or domain and realm that provides the identities involved in the associated events (required; `null` not allowed).
- Location Address: The communication service end point address, e.g., a URL.
- Location Name: The name of the host or service reporting the event.
- Principal Id: The user ID of the principal, relative to the authentication authority (required; `null` not allowed).
- Principal Name: The user name of the principal, relative to the authentication authority (optional; `null` allowed).
- Service Type: The protocol used when connecting to the service at Location Address.
- Time Source: The source of the clock setting, e.g., an NTP or SNTP server such as tick.usno.navy.mil.

The following code creates an Audit session record.

```
final IAuditSession aSession = Audit.acquireSession(
 /* authenticaion authority, in this case the Windows domain name */
    "recursionsw",
/* domain controller */"dal-exch.recursionsw.com",
/* reporting host name */"joedeveloper-xp",
/* user id of the principal */"joedeveloper",
/* name of the principal */"Joe Developer",
/* the service type */"" );
```

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

Create and write a Record

Building an audit event record requires first creating the empty record using a factory method, then populating the record using one or more setter methods. Adding the record to the stream of audit events happens when a record's `commit()` method is called. Exception handling is omitted in the following examples.

```
final IAuditRecord aRecord = Audit.acquireRecord(aSession);
```

Populating the new record can be accomplished in a single call, multiple calls to individual set methods, or a single call followed by calls to individual set methods. Here's an example of setting the fields using a single call, followed by an example setting the same values using individual setter calls. Note that in a `putEventInfo()` call, a colon delimits the subfields of the initiator and target information fields, so embedded colons, such as are found in a URL, must be escaped using % as the escape character. However, when set as individual subfields, such as seen below in the `setTargetInfo()` call, such escape coding is not required.

```
aRecord.putEventInfo(
/* event number */IAuditEvents.AUDIT_EVENT_START_SYS,
/* outcome code */IAuditOutcome.AUDIT_OUTCOME_SUCCESS,
/* time source */"tick.usno.navy.mil",
/* initiator info */"auth service:name in auth svc:id in auth svc",
/* target info */"Voyager:localhost%:8000:",
/* unspecified event data */
        "Event specific information, e.g., command line params");

aRecord.setEventInfo(
        "Event specific information, e.g., command line params" );
aRecord.setEventNumber( IAuditEvents.AUDIT_EVENT_START_SYS );
aRecord.setTimeSource( "tick.usno.navy.mil" );
aRecord.setInitiatorInfo( "auth service", "name in auth svc",
        "id in auth svc" );
aRecord.setOutcome( IAuditOutcome.AUDIT_OUTCOME_SUCCESS );
aRecord.setTargetInfo( "Voyager", "localhost:8000",
        "" , "" , "" , "" );
aRecord.setTimeStamp();
```

And finally, the code below illustrates adding the record to the audit event stream.

```
aRecord.commit();
```

If a required field other than the time stamp is not set, `commit()` will throw an `AuditException` exception. If the time stamp is not already set when `commit()` is called, `commit()` will set the record's time stamp to the current time.

Differences between Voyager and Open XDAS

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

The Voyager Audit subsystem is not a complete implementation of XDAS. Voyager's implementation omits, for example, the Audit Read API. Voyager derives the values of various constants and bit maps from the Open XDAS[2] implementation of XDAS, but differs from the Open XDAS audit record format in the following two ways.

1. The Open XDAS implementation expresses the record length field of an audit record as a four character hexadecimal number, while Voyager's implementation formats the field as a variable length decimal number. The XDAS specification on page 37 specifies the length field as containing "Digits 0-9", suggesting the length should be a decimal number.

2. The Open XDAS implementation expresses the audit record's version number as "OX1" ("O" is the letter, not the digit zero), while Voyager's implementation specifies the version number as an integer whose value is specified in `IAuditRecord.AUDIT_SERVICE_VERSION`. The XDAS specification on page 37 specifies the version number as containing "Digits 0-9", suggesting the version identifier should be numeric.

Basic Features

Overview

This chapter covers all the features of `Voyager` that are required to build a simple distributed application.

In this chapter, you will learn to:

- Use interfaces for distributed computing
- Create contexts describing client and server connections
- Create a remote object
- Send messages and handle exceptions
- Log information to the console
- Understand distributed garbage collection
- Use the naming service
- Work with proxies
- Export objects
- Remote-enable a class that has no interface

[2] See http://openxdas.sourceforge.net/

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

- Use the federated directory service

Using Interfaces for Distributed Computing

The Java and .NET languages support interfaces. An interface contains no code. It defines a set of method signatures that must be defined by the class that implements the interface. A variable whose type is an interface may refer to any object whose class implements the interface. By convention, `Voyager` interfaces begin with `I`. Your code does not need to follow this convention. An example of an interface follows:

Java, C#

```
public interface IStockmarket
  {
  int quote( String symbol );
  int buy( int shares, String symbol );
  int sell( int shares, String symbol );
  void news( String announcement );
  }
```

If the class `Stockmarket` implements `IStockmarket`, it is legal to write:

```
IStockmarket market = new Stockmarket(); // market refers to local
object
```

A remote object is represented by a special proxy object that implements the same interfaces as its remote counterpart. A variable whose type is an interface may refer to a remote object via a proxy, because both the remote object and its proxy implement the same interfaces. Consequently, as long as you use interface-based programming, the code for a remote method invocation through a proxy is coded exactly like a local method invocation directly to an object.

See the section Remote-Enabling a Class that has No Interface for information about remote-enabling a class that does not implement an interface.

Retrieving a `VoyagerContext`

Most references to Voyager services happen using an instance of a `VoyagerContext`, usually retrieved by calling `Voyager.startup()`. (or other variations of startup)

The `VoyagerContext` can access naming (`getNaming()`), the context manager (`getContextManager()`), client contexts (`acquireClientContext(String name)`), the default server context (`acquireDefaultServerContext()`), etc.

Creating or Retrieving a `ClientContext`

Voyager references a remote Voyager instance (process) through a `ClientContext`. An application creates a `ClientContext` using one of several methods implemented in

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

VoyagerContext. The methods acquireClientContext(Guid) and acquireClientContext(String) both retrieve or create a ClientContext. The first variant refers to a remote Voyager server with the indicated Guid. The second variant refers to a remote Voyager server with the indicated name. If the ClientContext already exists the existing instance is returned, but if the ClientContext doesn't exist a new ClientContext instance is created and returned.

The network address of a remote Voyager instance is set using the ClientContext's openEndpoint(url) method. Note that this method fails with a runtime exception if called on the ClientContext referencing the local Voyager. Creating the actual connection to the remote Voyager may be deferred until the connection is actually needed.

Creating or Retrieving a ServerContext

A ServerContext receives incoming Voyager messages and dispatches them for processing. A ServerContext also contains the collection of objects exported through that ServerContext. Voyager will not automatically create a ServerContext. The first ServerContext created is used as the default ServerContext unless a different one is explicitly identified using VoyagerContext's setDefaultServerContext(ServerContext) method.

Configuring a ServerContext is a two step sequence: the first step is creating the ServerContext and the second step is providing the ServerContext with the URL on which to listen for incoming messages. As with the ClientContext, the VoyagerContext provides several methods for retrieving or creating a ServerContext, including acquireServerContext(Guid) and acquireServerContext(String). Both methods return an existing ServerContext if one already exists, or create and return a new one. The second step calls the ServerContext startServer(String) method to provide the ServerContext an address on which to listen.

Creating a Remote Object

A remote object is represented by a special object called a *proxy* that implements the same interfaces as its remote counterpart. The proxy exists in the local VM and implements an interface that is also visible in the local VM. A variable declaration whose type is an interface may refer to a remote object via a proxy, because both the remote object and its proxy implement the same interfaces. Consequently, as long as you use interface-based programming, the code for a remote method invocation through a proxy is coded exactly like a local method invocation directly to an object.

To create an object at a location referenced by a ClientContext, call getFactory() to retrieve the ClientContext's Factory instance, then invoke one of Factory's create() methods. All the create() methods return a proxy to the newly created object,

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

creating the proxy class dynamically if it does not already exist and the runtime environment supports dynamic proxy generation.

There are several variations of `create()`, depending on whether the class constructor takes arguments. You must always fully qualify the name of the class. For example, use `examples.stockmarket.Stockmarket` instead of `Stockmarket`. To create a default instance of `Stockmarket` in the local program and another in the program running on port `8000` of the machine `Dallas`, create the following code.

Java, C#

```
String className = "examples.stockmarket.Stockmarket";
// create locally ...
Factory aFactory =
voyagerContext.getLocalClientContext().getFactory();
IStockmarket market1 = (IStockmarket) aFactory.create(className);
//create remotely ...
ClientContext cc = voyagerContext.acquireClientContext("Dallas");
cc.openEndpoint("//dallas:8000");
aFactory = cc.getFactory();
IStockmarket market2 = (IStockmarket)aFactory.create(className);
```

Both `market1` and `market2` will be proxy objects. The `market1` proxy refers to a local instance of `Stockmarket`, and the `market2` proxy refers to a remote instance.

To create an instance of `Stockmarket` and use the constructor that takes a String and an integer, type:

Java

```
Object[] args = new Object[] { "NASDAQ", new Integer( 42 ) };
IStockmarket market3 =
    (IStockmarket) aFactory.create( className, args);
```

C#

```
Object[] args = new Object[] { "NASDAQ", 42 };
IStockmarket market3 =
    (IStockmarket) Factory.create(className, args);
```

Note that primitive arguments must be wrapped in their `Object` equivalents if the compiler does not automatically handle that process, which is called *boxing* or *autoboxing*.

Sending Messages and Handling Exceptions

A message sent via a proxy is handled according to the following rules.

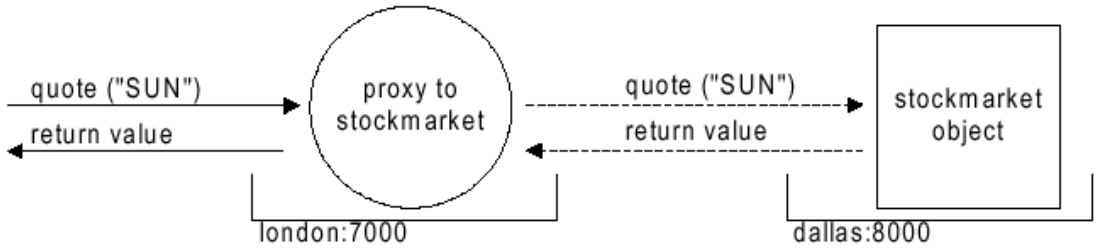
If the destination object is in a different virtual machine, the arguments and return value must be sent across the network. If an argument implements `com.recursionsw.ve.IRemote`, a proxy to the argument is sent (pass by reference). If the argument implements `com.recursionsw.ve.VSerializable` or `java.io.Serializable`, a copy of the argument is sent using standard Java-style

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

serialization (pass by value). Morphology of the arguments is maintained – an object that is an argument or part of an argument is copied exactly once, and an argument or part of an argument that shares an object in the local virtual machine also shares a copy of the object in the remote virtual machine. Rules for an argument also apply to a return value.

If the destination object is in the same virtual machine, arguments passed by reference will pass the original object instead of a proxy to the object. Serializable objects will still be serialized even though they are already in the same VM. This maintains the same semantics for a method invocation: regardless of whether the calling object and called object are on the same VM, the called method will get a copy of the serializable object which it can safely modify. Without this behavior, when the method was invoked locally it would modify the original object and when the method was invoked remotely it would modify a copy of the object.

The following figure shows how a remote message is processed.



If a remote method throws an exception, it is caught and re-thrown in the local program. If a `Voyager`-related exception, such as a network error, argument serialization error, etc., occurs the exception is wrapped in a `com.recursionsw.ve.RuntimeRemoteException`. `RuntimeRemoteException` is also thrown if a method invoked on an object via a proxy throws an exception that is not declared in the throws clause for that method. In each case, the public `getDetail()` method returns the original exception. `Voyager`'s exception handling policy allows you to select between checked and unchecked exceptions.

The Basics1 Example demonstrates basic messaging and remote construction.

Logging Information to the Console

The `com.recursionsw.lib.util.Console` class allows you to log information, including stack traces of remote exceptions, to the console or a `PrintStream`. Use `Console.enableTopic()` or `Console.addEnableTopics()` to select enabled topics. Use `Console.disableTopic()` to turn off a previously selected topic. Pre-defined constants used by `Voyager` include:

```
LogConst.SILENT
```

Disables logging of messages at the EXCEPTIONS and VERBOSE levels.

```
LogConst.EXCEPTIONS
```

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

Displays stack traces of remote exceptions and unhandled exceptions to the console.

`LogConst.VERBOSE`

Displays stack traces of remote exceptions, unhandled exceptions, and internal debug information and stack traces to the console.

Enabled topics can be set via the system property `console.enabledTopics`. This is a comma-separated list of topics that may include “silent”, “exceptions” and “verbose” as well as user-defined topics. To set the enabled topics for `Voyager` from the command line, use the `voyager -l` option with the list of topics to enabled. The `-exceptions` and `-verbose` command-line options will also set `Voyager`'s enabled topics.

Understanding Distributed Garbage Collection

`Voyager`'s distributed garbage collector (DGC) reclaims objects when they are no longer pointed to by any local or remote references. Just as with the native VM’s garbage collector, distributed garbage collection happens automatically and transparently.

`Voyager` uses an efficient "delta pinging" scheme to reduce DGC network traffic. Each program notes when references to remote objects are created and destroyed. In each DGC cycle, which is 2 minutes by default, the program sends each referenced remote program a single message containing a summary of the references to its objects that were added/removed since the last DGC cycle. By tracking this information as it changes over time, each program can tell when no remote references exist to an exported object. At this time, the DGC mechanism on that VM releases its reference to the object, permitting the VM's garbage collection mechanism to reclaim the object. The DGC mechanism will also release its reference to an object if the remote VM(s) that have proxy references to the object cannot be reached for three consecutive cycles. This keeps the `Voyager` VM from using an increasing amount of memory as remote VM's are started and shut down.

Objects which have been bound in `Voyager`'s naming service are referenced permanently, as are mobile Agents which have their `setAutonomous()` flag set to `true`.

DGC Notification

If a class is interested in being notified when a remote reference to an instance of the class is about to be discarded by DGC, it can implement the `com.recursionsw.ve.messageprotocol.vrmp.dgc.IDGCListener` interface. The callback function `discardingReference()` is invoked when a remote reference to the object is about to be discarded. The object has the option to allow or delay discarding the reference. See the API documentation for `com.recursionsw.ve.messageprotocol.vrmp.dgc.IDGCListener` for more details.

DGC Discard Delay Configuration

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

DGC reference discard delay configuration support, provided via the `DGC.setDiscardDelay(long)` method, sets the delay between the time a remote reference is last used and the time the reference is discarded by DGC. See the API documentation for `com.recursionsw.ve.messageprotocol.vrmp.dgc.DGC` for more details.

Using Naming Services

The `Voyager Namespace` service provides unified access to a variety of naming services. This section shows how to use the `Namespace` class to bind names to objects and look them up.

The class `com.recursionsw.ve.Namespace` is a façade, which unifies binding and lookup operations to any naming service implementation. `Voyager` provides the following naming service implementations:

- `Voyager` federated directory service

The `Namespace` class differentiates among various naming service implementations by using a unique prefix for each implementation. For example, the `Voyager` federated directory service uses the prefix `vdir:`. Binding and lookup operations use the name's prefix to determine which underlying naming service implementation to access for the operation. Once an object has been bound, it can be looked up by any type of client using any lookup prefix whose protocol is supported by Voyager.

To bind a name to an object, retrieve the `Namespace` instance from the `VoyagerContext` and invoke `bind()` with the name expressed as an URL. The following code segment creates a `Stockmarket` on the host `//dallas:8000` and then binds it to the name `NASDAQ` for later lookup:

Java, C#

```
String className = "examples.stockmarket.Stockmarket";
VoyagerContext voyagerContext = null;
VoyagerContext = Voyager.startup();
ClientContext cc = VoyagerContext.acquireClientContext("Dallas");
cc.openEndpoint("//dallas:8000");
aFactory = cc.getFactory();
IStockmarket market = (IStockmarket)aFactory.create(className);
cc.getNamespace().bind("/NASDAQ", market );
```

To obtain a proxy to a named object, invoke the `Namespace lookup()` method. The following example obtains a proxy to the object that was created and named by the previous code segment, and where `cc` is the `ClientContext`.

Java, C#

```
IStockmarket market = (IStockmarket)
     cc.getNamespace().lookup( "/NASDAQ" );
```

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

The default naming service is the `Voyager` federated directory service (prefix `vdir:`). If a prefix is missing from a name, it is assumed to be `vdir:`. `Voyager` provides naming service implementation that is installed automatically.

Voyager

Service	Prefix
Voyager federated directory service	`vdir:`

The Naming2 Example illustrates the default naming service.

Working with Proxies

Voyager's proxy classes provide the network communications capabilities to perform remote invocations and work with remote references to objects. All Voyager proxy classes extend `com.recursionsw.ve.Proxy` and implement the interface(s) of their referent. In some runtime environments the Voyager classloader generates required proxy classes at runtime automatically the first time Voyager requires an instance of that proxy class (typically, the first time a remote reference is acquired by the VM). For any runtime environment the Voyaer `pgen` tool can create at build time the necessary proxy classes. Use any of the following to obtain a proxy to an object.

```
Factory's create( String classname )
```

All of the `Factory create()` methods return a proxy to a newly created remote object, where `classname` is the name of the class for which you are creating an instance.

```
Namespace's lookup( String name )
```

Returns a proxy to the object bound to a particular name. This is true for all of the `Namespace lookup()` methods.

```
Proxy.of( Object object )
```

As with the other classes above, all of `Proxy's of()` methods return a proxy. If the specified object is already a proxy, returns the object; otherwise returns a proxy to the object.

Special Methods

A method call on a proxy is forwarded to its associated object unless it is one of the following special methods:

```
getClass(), notify(), notifyAll(), wait()
```

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

These methods are all final methods in `Object` and are executed directly by the proxy.

`hashCode()`

Returns the hash code of the proxy itself. Use `remoteHashCode()` to obtain the hash code of a proxy's associated object. Two proxies return the same hash code if they refer to the same object.

`equals()`

Returns true if the argument is a proxy that refers to the same object as the receiver. Use `remoteEquals()` to compare the proxy's associated object with another object.

Additional methods in `Proxy` follow.

`isLocal()`

Returns true if the proxy is in the same VM as its associated object.

`getLocal()`

If the proxy is in the same VM as its associated object, returns a direct reference to the object; otherwise returns null.

`getClientContext()`

Returns the `ClientContext` of the proxy's associated object.

`toExternalForm()`

Returns a string that, when passed to `Namespace's lookup()`, returns a proxy to this proxy's object. This method is useful if you need to transmit a reference to an object via an external medium or for persisting a reference to an object.

To pass an object by reference, either explicitly pass a proxy obtained using `Proxy.of()`, or implicitly pass a proxy by ensuring that the object class implements `com.recursionsw.ve.IRemote` (Voyager will also pass a proxy reference if the object does not implement `java.io.Serializable` or `com.recursionsw.ve.VSerializable`).

Serializing Proxies

`Proxy` classes are typically generated at runtime. When reading a `Proxy` class from an `ObjectInputStream` it may be necessary to generate the appropriate `Proxy` subclass. In these cases, a subclass of `ObjectInputStream` should be used whose `resolveClass()` method requests the Proxy subclass from Voyager's class manager. For example:

Java

```
class VObjectInputStream extends ObjectInputStream
  {
  protected Class resolveClass (ObjectStreamClass v ) throws
ClassNotFoundException
```

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

```
        {
        return ClassManager.loadClass(v.getName());
        }
    }
```

You may also use the `com.recursionsw.ve.directory.VObjectInputStream` class for this purpose.

If your intent in serializing proxies is to be able to obtain a reference to a remote object after a restart of the VM, you may prefer `Proxy.toExternalForm()`. This method returns a `String` which may be passed to `Namespace.lookup()` to obtain a proxy to the object.

Exporting Objects

To receive remote messages, an object must be exported to exactly one local `ServerContext`. After it is exported, all remote messages to an object arrive via its export `ServerContext`.

If a proxy to an unexported object is passed to a remote program, Voyager automatically exports the object to the default `ServerContext`. If Voyager was started on an explicit URL, the default `ServerContext` is the one listening on the startup URL, otherwise the default `ServerContext` is the first one created or the one selected using `VoyagerContext's setDefaultServerContext(ServerContext)` method. Note that Voyager never automatically creates a `ServerContext`, and if an implicit export happens before a `ServerContext` is created, the export will fail with an exception.

The automatic export mechanism is sufficient for most applications. However, there are times where it is useful to partition objects among more than one `ServerContext`. For example, security reasons might dictate associating one group of objects with a `ServerContext` whose URL that is connected to an intranet, while associating another group of objects with a `ServerContext` whose URL connects to the Internet via SSL. Because programs on the Internet can only communicate via the server using SSL connections, they can only send messages to the group of objects that are exported on that `ServerContext`.

To explicitly export an object, use the `export()` method on the appropriate `ServerContext` instance.

```
Proxy export( Proxy aProxy )
```

Alternately, call `Proxy's` static `export()` method and provide the appropriate `ServerContext` as the second argument.

```
Proxy export( Object object, ServerContext serverContext )
```

Exports the object on the `ServerContext`.

```
unexport( Object object )
```

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

The static `Proxy` method `unexport()` removes the object from the `ServerContext`'s collection of exported objects. The `ServerContext` instance method `unexport()` does the same thing.

Note: An exported object can receive messages on exactly one `ServerContext`.

The Basics2 Example binds a name to an object exported on an explicit `ServerContext`.

Remote-Enabling a Class that has No Interface

Voyager allows an object to be constructed remotely and sent messages even if its class does not implement an appropriate interface.

The igen utility generates a default interface from a class. A default interface has the public methods of the original class and is named using `I` followed by the name of the original class. If the original class is in the `java.*` package, the default interface is placed in the `com.recursionsw.java.*` package, otherwise it is placed in the same package as the original class.

For example, to generate the default interface `com.recursionsw.java.util.IVector` from a `java.util.Vector`, type:

```
igen java.util.Vector
```

When a proxy class is dynamically generated from a class that does not implement `com.recursionsw.ve.IRemote`, or its default interface, the default interface is automatically added to the list of classes that the proxy class implements. An instance of the proxy class implements the default interface even though its associated object does not.

To construct a remote `java.util.Vector` and send it messages, write:

Java

```
import com.recursionsw.java.util.IVector;

ClientContext cc = voyagerContext.acquireClientContext("Dallas");
cc.openEndpoint("//dallas:8000");
aFactory = cc.getFactory();
IVector vector = (IVector) aFactory.create("java.util.Vector");

// note that the proxy implements IVector even though Vector does not

vector.addElement( "hi" );
```

For more information about `igen`, see Utilities.

Working with Federated Directory Services

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

The Voyager federated directory service 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`. The building block of the directory service is a `Directory`, which has the following interface:

`put( String key, Object value )`

Associates a key with a value. If key is a simple string, associates it with the specified value in the local directory. If key is a path, looks up the `Directory` associated with the head of the path name and then forwards the `put()` message with the remaining tail of the path name. Returns the value previously associated with the key or `null` when there was none.

`get( String key )`

Returns the value associated with a particular key. If key is a simple string, return its associated value in the local directory or `null` when there is none. If key is a path, looks up the `Directory` associated with the head of the path name and then forwards the `get()` message with the remaining tail of the path name.

`remove( String key )`

Removes the directory entry with the specified key. If key is a simple string, removes its entry from the local directory. If key is a path, looks up the `Directory` associated with the head of the path name and then forwards the `remove()` message with the remaining tail of the path name. Returns the value that was associated with the key or `null` when there was none.

`getValues()`

Returns an array of the values in the local directory.

`getKeys()`

Returns an array of the keys in the local directory.

`clear()`

Removes every entry from the local directory. Removing the entries has no effect on the directories that the local directory used to reference.

`size()`

Returns the number of keys in the local `Directory`.

To create a simple directory of local objects, create a Directory object and send it the *put()* message with a string key and a local object.

```
Directory symbols = new Directory();
symbols.put( "CA", "calcium" );
symbols.put( "AU", "gold" );
// symbols.get( "CA" ) would return "calcium"
```

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

To create a chained directory structure, a Directory that refers to another Directory , send put() to a Directory object with another directory or a proxy to a remote Directory as the second parameter.

```
Directory root = new Directory();
root.put( "symbols", symbols ); // associate "symbols" with
the symbols directory
// root.get( "symbols/CA" ) would return "calcium"
```

Because `Directory` implements `IRemote`, you can pass a local directory as a parameter to a remote directory and it is automatically sent as a proxy.

The Naming1 Example sets up a simple federated directory service.

Advanced Features

Dynamic Aggregation™

Voyager supports dynamic aggregation, which allows you to attach new code and data to an object at runtime.

This feature resolves the following problems that commonly occur during the construction of an object-oriented system:

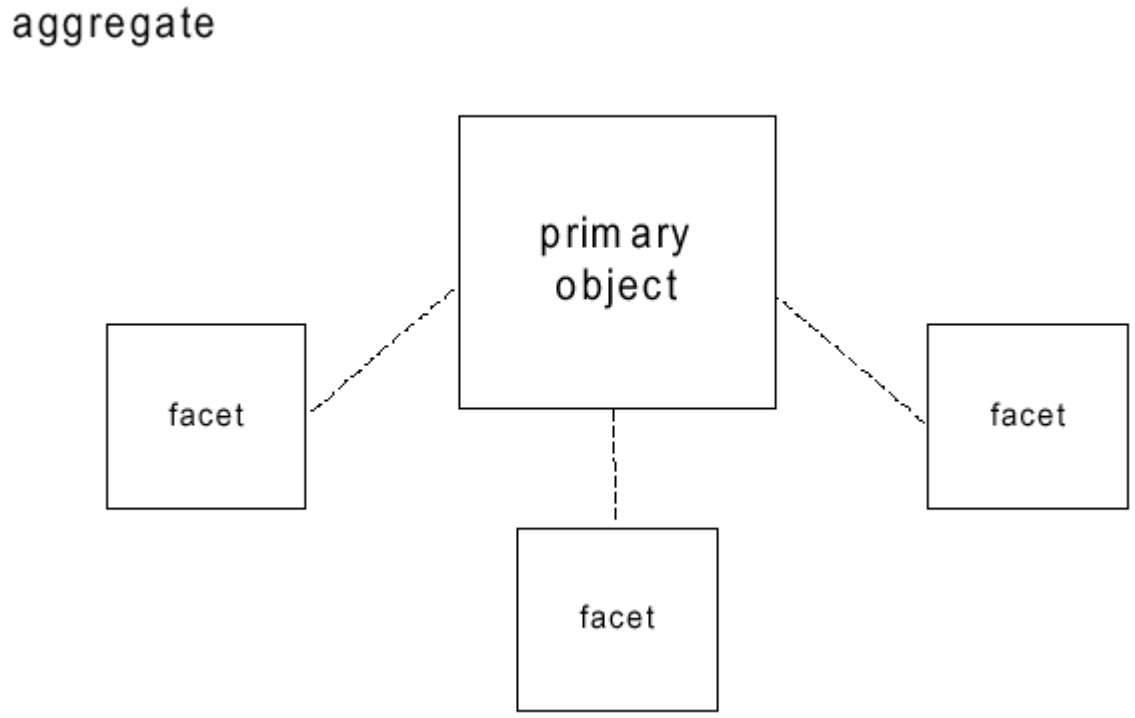
- Adding behavior to a third-party component whose source is not available.
- Customizing an object in a subsystem-specific way, so it can be used by multiple subsystems.
- Extending an object's behavior at runtime, perhaps in unforeseen ways.
- Dynamic aggregation represents a fundamental step forward for object modeling and complements the mechanisms of inheritance and polymorphism.
- In this chapter you will learn to:
- Work with dynamic aggregation
- Access and add facets
- Select a facet implementation
- Create facet-aware classes

Working with Dynamic Aggregation

Dynamic aggregation allows you to attach secondary objects, or facets, to a primary object at runtime. A primary object and its facets form an aggregate that is typically persisted, moved, and garbage-collected as a single unit.

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

The following diagram illustrates a primary object and its facets.



There are several rules associated with facets.

- A class does not have to be modified in any way for its instances to play the role of a primary object and/or facet.
- The class of a facet does not have to be related in any way to the class of a primary object. An instance of a class can be added as a facet to any kind of primary object.
- Facets cannot be nested. In other words, a facet cannot have a facet.
- Facets cannot be removed. After a facet is added, it remains for the life span of the aggregate.
- A primary object and its facets have the same life span and are garbage-collected only when there are no references to either the primary object or any of its facets.

There are many uses for dynamic aggregation. For example, you can dynamically add a bonus plan facet to an employee, a repair history facet to a car, a payment record facet to a customer, or a hyperlinks facet to a generic object.

Accessing and Adding Facets

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

A primary object's facets are represented by an instance of `Facets` that is initially set to `null`. To access an object's `Facets`, use one of the following static `Facets` methods:

- `get( Object object )`

Returns the object's `Facets`, which may be `null` .

- `of( Object object )`

Returns the object's `Facets`, setting it to an initialized instance of `Facets` when it is currently equal to `null` .

Because a facet is part of an aggregation, invoking `Facets.get()` or `Facets.of()` on a facet returns the `Facets` instance of the facet's primary object.

To manipulate an object's facets, use the following instance methods defined in `Facets`:

- `get( String interfacename )`

Returns a proxy to a facet that implements the specified interface or `null` if no match is found. The interface name must be fully qualified.

- `of( String interfacename )`

See Selecting a Facet Implementation below.

- `getPrimary()`

Returns a proxy to the primary object.

- `getFacets()`

Returns an array of proxies to the primary object's facets.

The Aggregation1 Example demonstrates facets by adding an `Account` facet to an `Employee` and later accessing the facet from a remote program.

Two additional static helper methods in Facets that simplify facet manipulation follow:

- `get( Object object, Class type )`

Returns the objects when the specified object is an instance of the specified interface. Otherwise, returns a proxy to the facet when the specified object has a facet that is an instance of the specified interface. Returns `null` when neither rule applies.

- `of( Object object, Class type )`

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

Returns the object when the specified object is an instance of the specified interface. Otherwise, returns a proxy to the facet when the specified object has a facet that is an instance of the specified interface. Adds and returns a proxy to a facet that implements the specified type when neither rule applies.

Provide static `get()` and `of()` methods to further simplify access to facets. For example, assuming that the name of the facet interface is `IRepairHistory` , add static `get()` and `of()` helpers methods to `RepairHistory`. For example:

Java

```
static public IRepairHistory get( Object object )
  {
  return ( IRepairHistory )
com.recursionsw.ve.Facets.get( object, IRepairHistory.class );
  }

static public IRepairHistory of( Object object ) throws
ClassCastException
  {
  return ( IRepairHistory)
com.recursionsw.ve.Facets.of( object, IRepairHistory.class );
  }
```

These methods then allow you to write code like the following example.

Java

```
// return the car's repair history facet or null if it does
not have one
IRepairHistory history1 = RepairHistory.get( car1 );

// return the car's repair history facet, adding one if it
does not already exist
IRepairHistory history2 = RepairHistory.of( car2 );
```

The Aggregation2 Exampledemonstrate use of the `of()` and `get()` methods to add and access a `Security` facet on an `Employee`.

Selecting a Facet Implementation

A facet implementation varies based on the class of primary object. For example, `BonusPlan.of()` may need to attach a different kind of bonus plan facet to a Programmer than to a Manager. Voyager uses a simple scheme for selecting the class of facet that is added during an `of( object )` operation.

Assuming that the class of `object` is `MyClass`, `MyFacet.of( object )` attempts to attach a facet that implements `IMyFacet` and is called `xxxMyFacet`, where the search starts with `xxx="MyClass"` and moves up `MyClass` 's superclass chain to `xxx="Object"`. Classes that match the name without implementing `IMyFacet` are ignored. If the head of

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

the superclass chain is reached and there is no match for `ObjectMyFacet`, a last chance match is attempted using `xxx=""`.

During each search cycle, the candidate class is first looked for in the package of `MyClass` and then in the package of `MyFacet`.

For example, assume that `company.Programmer` extends `company.Employee` which in turn extends `java.lang.Object`. Assume also that the full path of `BonusPlan` is `incentive.BonusPlan`. If the statement `BonusPlan.of( object )` is executed where object is an instance of Programmer, the search process picks the first class in the following series that exists and implements `IBonusPlan`.

1. `company.ProgrammerBonusPlan`
2. `incentive.ProgrammerBonusPlan`
3. `company.EmployeeBonusPlan`
4. `incentive.EmployeeBonusPlan`
5. `company.ObjectBonusPlan`
6. `incentive.ObjectBonusPlan`
7. `company.BonusPlan`
8. `incentive.BonusPlan`

After the class is selected, the facet is instantiated using the default constructor.

The Aggregation3 Example illustrates facet selection

Packaging Facets

Use the following guidelines when packaging facet classes:

- Begin the name of your facet interface with `I`.
- Create a class whose name corresponds to the interface name and omits the `I` prefix, and populate it with the `of()` and `get()` static helper methods.
- Place the default facet implementation in the same package as the facet interface, and prefix its name with the name of the most general class to which the facet

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

applies. For example, if the default facet applies to all objects, use the `Object` prefix.

- Place specialized facet implementations into the same package as their associated class.

Creating Facet-Aware Classes

To make a class facet-aware, implement `com.recursionsw.lib.facets.IFacet`, which declares the following method:

- `isTransient()` - If this method returns true, override the regular rule for garbage collection of facets and reclaim the facet immediately when there are no more references to it. Choose this feature when a facet is stateless and does not need to be associated with the primary object after its work is complete. The Voyager `Mobility` facet is an example of a transient facet. See the Mobility and Agents chapter for more information.

In addition, any class that implements `IFacet` can provide a constructor that takes a single `IFacets` parameter. If provided, this constructor is invoked instead of the default constructor whenever an instance of the class is added as a facet. The `IFacets` argument is set to the Facets instance of the primary object.

The Aggregation4 Example illustrates facet-aware classes by defining a transient `BonusPlan` facet for a `Manager`.

Timers

Voyager's timer Services include the `Stopwatch` and `Timer` classes. You can use a `Stopwatch` object to clock time intervals and print time measurement statistics. You can use a `Timer` object to generate timer events and add listeners to timers.

In this chapter, you will learn to:

- Clock time intervals
- Use timers and timer events

Clocking Time Intervals

Use Voyager's `Stopwatch` class to clock time intervals. You can start and stop a `Stopwatch` object an unlimited number of times before resetting it; every start/stop cycle is called a lap. You can access the cumulative lap time, average lap time, and last lap time, and you can record individual lap times.

see the following methods defined in `Stopwatch` to clock time intervals:

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

- `getDate()`

Returns the current date.

- `getMilliseconds()`

Returns the current time in milliseconds since January 1, 1970, 00:00:00 GMT.

- `reset()`

Resets the stopwatch, clears lap times, and sets the lap count to zero.

- `start()`

Starts a stopwatch.

- `stop()`

Stops a stopwatch, increments the lap count, and, when enabled, records the lap time.

- `lap()`

Stops the stopwatch temporarily to record the lap time and immediately restart it.

- `setRecordLapTimes( boolean flag )`

Enables or disables the recording of lap times.

- `isRecordLapTimes()`

Returns a boolean indicating whether lap-time recording is enabled.

- `getLapCount()`

Returns the current completed lap count.

- `getLapTime()`

Returns the last completed lap time.

- `getLapTimes()`

Returns a long array of recorded lap times. If lap-time recording is disabled, an empty array is returned.

- `getTotalTime()`

Returns the sum of all completed lap times.

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

- `getAverageLapTime()`

Returns the average lap time.

The Stopwatch1 Example starts and stops a `Stopwatch` object and prints various time measurement statistics.

Using Timers and TimerEvents

Voyager's `Timer` class acts like an alarm clock. You can set a `Timer` object to send a `TimerEvent` to one or more listeners. Upon receiving an event, a listener performs an action. When the action is complete, the timer can continue by sending a `TimerEvent` to its next listener. To set up a timer and listeners, follow these steps:

1. Construct a timer and one or more listeners.
2. Set the timer to generate one-shot or periodic events.
3. Add the listeners to the timer.

Constructing a Timer

When you construct a timer, it is placed in a `TimerGroup` . Each `TimerGroup` has its own thread, and all timers in a `TimerGroup` share its thread to generate events. Unless specified otherwise, a timer is placed in the default `TimerGroup` and its thread priority is set to normal (`Thread.NORM_PRIORITY`).

You can make a group of timers use a separate thread by assigning the timers to a discrete `TimerGroup` at construction. First, construct a new `TimerGroup` , optionally supplying a thread priority as a parameter, and then construct timers with the new `TimerGroup` as a parameter:

```
TimerGroup newgroup = new TimerGroup( Thread.MIN_PRIORITY );
Timer timer1 = new Timer( newgroup );
Timer timer2 = new Timer( newgroup );
```

Setting a Timer

You can set a timer to generate an event at a particular point in time, after a specified period of time, or periodically with the following methods defined in `Timer`:

- `alarmAt( Date date )`

Sets the timer to generate an event at the specified time.

- `alarmAfter( long milliseconds )`

Sets the timer to generate an event after the specified number of milliseconds.

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

- `alarmEvery( long period )`

Sets the timer to generate an event every time the specified period of time (in milliseconds) elapses.

Other `Timer` methods used to work with timer events include:

- `clearAlarm()`

Cancels the generation of the timer's event.

- `getAlarm()`

Returns the time that the timer is scheduled to generate its next event.

- `getPeriodicity()`

Returns the number of milliseconds between the timer's events.

Adding a Listener to a Timer

A timer generates an event only if it has a listener. Add an object to a timer as a listener using these steps:

1. Ensure that the object's class implements the `TimerListener` interface.

2. Send `addTimerListener()` to the timer with an instance of the object as a parameter.

To remove a listener from a timer, call `removeTimerListener( TimerListener listener )` on the timer.

Multiple listeners to a timer use a single thread, the timer's `TimerGroup` thread, to perform actions upon receiving events. You can override this default behavior by wrapping a listener with a `TimerListenerThread` ; that is, you can construct a `TimerListenerThread` object with an instance of the listener as a parameter. `TimerListenerThread` implements `TimerListener` .

For example, suppose a `listener1` object listens to a `timer1` timer. The following code wraps `listener1` with a `TimerListenerThread` and then adds the wrapped listener to `timer1` .

```
TimerListener timerListener1 = new
TimerListenerThread( listener1 );
timer1.addTimerListener( timerListener1 );
```

A listener wrapped with a `TimerListenerThread` is dynamically allocated a new thread from a thread pool when it receives an event. In this way, the timer can use its

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

`TimerGroup` thread to continue delivering events to other listeners without waiting for the wrapped listener to perform its action.

By default, the priority of a new thread allocated by `TimerListenerThread` is equal to the priority of the current thread. To override the default, specify the desired priority when you construct the `TimerListenerThread` object:

```
new TimerListenerThread( listener1, Thread.MAX_PRIORITY )
```

The Timer1 Example demonstrates a ramification of Voyager's default thread behavior, that is, sharing a `TimerGroup` thread. Two listeners receive `TimerEvent` events via the same thread, so the second listener does not receive a `TimerEvent` until the first listener completes its `timerExpired()` method.

The Timer2 Example demonstrates creating a new `TimerGroup` . A `timer1` listener receives an event from the default `TimerGroup's` thread, and a `timer2` listener receives an event from the new `TimerGroup's` thread.

The Timer3 Example demonstrates allocating listeners separate threads to perform actions upon receiving `TimerEvent` events. The second listener receives a `TimerEvent` before the first listener's `timerExpired()` method completes.

Advanced Messaging

You can send synchronous messages in Voyager using regular Java and .NET syntax. However, many applications need greater flexibility, so Voyager provides a message abstraction layer that supports more sophisticated messaging features.

In this chapter, you will learn to:

- Invoke messages dynamically
- Retrieve remote results by reference
- Use multicast and publish/subscribe

Invoking Messages Dynamically

You can dynamically invoke messages either synchronously or asynchronously.

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, if any, is received. For example, the following line of code sends a synchronous `buy()` message to an instance of `Stockmarket`.

```
int price = market.buy( 42, "SUN" );
```

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

You can send a synchronous message dynamically using `Sync`'s `invoke()` method, which returns a `Result` object when the message has completed. You can then query the `Result` object to get the return value/exception. To send a synchronous message, retrieve the synchronous invoker from the appropriate ClientContext (), then call `invoke()`. The simplest version requires passing the following parameters.

- Target object
- Name of the method you want to call on the target object
- Parameters to the dynamically invoked method in an object array

For example, the following code uses a `Sync` invoker to dynamically invoke a `buy()` message on an instance of `Stockmarket`.

Java

```
ClientContext cc =
voyagerContext.acquireClientContext("Server8000");
Result result = cc.getSyncInvoker().invoke( market, "buy", new
Object[] { new Integer( 42 ), "SUN" } );
int price = result.readInt();
```

C#

```
ClientContext cc =
voyagerContext.acquireClientContext("Server8000");
Result result = cc.getSyncInvoker().invoke( market, "buy", new
Object[] { 42, "SUN" } );
int price = result.readInt();
```

Primitive arguments must be sent as their object equivalents if the compiler does not autobox primitives.

In most cases, the simple name of the method suffices. However, if there is more than one method with the same name in the target object, the method name must be specified with argument types using the syntax `method( type1, type2 )`. Spaces in the signature are ignored, and the return type must not be specified. A version of the previous example that uses the longer version of the signature follows:

```
ClientContext cc =
voyagerContext.acquireClientContext("Server8000");
Result result = cc.getSyncInvoker().invoke( market,
        "buy", new Object[] { new Integer( 42 ), "SUN" } );
int price = result.readInt();
```

You can query a `Result` object using the following methods. In the case of synchronous methods, the reply value is always available by the time these methods are called. `Future` messages allow the methods to be called before the reply value is received.

- `isAvailable()`

Returns `true` if the `Result` received its return value.

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

- `readXXX(), where XXX = Boolean , Byte, Char, Short, Int, Long, Float, Double, Object`

Returns the value of `Result`, blocking until either the value is received or the timeout period of `Result` elapses. If the value is not received within the timeout period, a `TimeoutException` is thrown. See the Future Messages section for information about timeouts. The timeout countdown starts when `readXXX()` is called, not when the message is actually sent. If a remote exception occurs during a future message invocation and you attempt to call `readXXX()` on `Result`, the exception is automatically rethrown. See the Voyager Basics chapter for information about exceptions.

- `isException()`

Waits for a reply and then returns true if Result contains an exception.

- `getException()`

Waits for a reply and then returns the exception contained in `Result` or `null` when no exception occurred.

The Message1 Example demonstrates invoking a synchronous instance method using Voyager's dynamic invocation feature.

One-Way Messages

A one-way message does not return a result. When a caller sends a one-way message, the caller does not block while the message completes, so sending a one-way message is fast. You can send a one-way message dynamically using `OneWay`, which performs "fire-and-forget" messaging.

To send a one-way message dynamically, call the `OneWay invoke()` method, passing the following parameters.

- Target object
- Name of the method you want to call on the target object
- Parameters to the dynamically invoked method in an object array

For example, the following line of code uses `OneWay` to dynamically invoke a one-way `buy()` message on an instance of `Stockmarket`.

Java
```
ClientContext cc =
voyagerContext.acquireClientContext("Server8000");
Result result = cc.getOneWayInvoker().invoke( market,
       "buy", new Object[] { new Integer( 42 ), "SUN" } );
```
C#

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

```
ClientContext cc =
voyagerContext.acquireClientContext("Server8000");
Result result = cc.getOneWayInvoker().invoke( market,
       "buy", new Object[] { 42, "SUN" } );
```

The Message2 Example demonstrates sending a one-way message.

Future Messages

A future message immediately returns a `Result` object, which is a placeholder to the return value. When a caller sends a future message, the caller does not block while the message completes. You can use `Result` to retrieve the return value at any time by polling, blocking, or waiting for a callback.

To send a future message, call `Future`'s `invoke()` method, passing the following parameters:

- Target object
- Name of the method you want to call on the target object
- Parameters to the dynamically invoked method in an object array

For example, the following code uses `Future` to dynamically invoke a `quote()` message on a `Stockmarket` object and then reads the return value at a later time.

```
ClientContext cc =
voyagerContext.acquireClientContext("Server8000");
Result result = cc.getFutureInvoker().invoke( market,
       "quote", new Object[] { "SUN" } );
// perform other operations here
result.readInt(); // block for price, if necessary
```

The Message3 Example demonstrates sending a future message and reading the return value with a blocking call. This example also demonstrates blocking reads when the placeholder result of the future invocation is a thrown exception.

You can be notified when a future return value arrives through the standard Java event/listener mechanism. When a return value arrives, `Result` sends `resultReceived()` with a `ResultEvent` object to every `ResultListener` that either was specified in the full version of `invoke()` or was added to the `Result` object after the message was sent.

The Message4 Example demonstrates receiving an event notification of the arrival of the return value to a future invocation.

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

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

The Message5 Example demonstrates Voyager's ability for multiple threads to block while waiting for the return value to a single future invocation.

By default, Voyager messages are synchronous and never time out. However, you can set a timeout for a future message by using the full version of `Future's invoke()`. For example, the following line of code creates a `Result` with a timeout period of 10,000 milliseconds.

```
Result result = cc.getFutureInvoker().invoke( market,
   "quote", new Object[] { "SUN" }, false, 10000, null );
```

The timeout period does not begin until `Result` is read.

Voyager also allows you to change the timeout value for a `Result` generated by a future message. Use the following `Result` methods to work with timeouts:

- `setTimeout( long timeout )`

Changes the timeout value for a `Result`. When `Result` is read, the timeout period begins. Reads that take longer to complete than the specified timeout period cause a `TimeoutException` to be thrown. See the Voyager Basics chapter for information about exceptions.

- `getTimeout()`

Returns the current timeout value for a `Result`. The default value, zero, indicates the `Result` never times out.

The Message6 Example demonstrates Voyager's support of method invocations that time out.

Retrieving Remote Results by Reference

By default, `Future's invoke()` and `Sync's invoke()` return a copy of a remote method result. If a result is large, undesirable network traffic occurs. With Voyager, you can tell `Future` or `Sync` to return a proxy to a result instead, thereby greatly reducing network traffic. If the result is not serializable, returning a proxy eliminates the need for serialization and allows the method to be invoked successfully. As expected, a proxy to a result keeps the remote result alive. To request that `Future` or `Sync` return a proxy to a result, use the full version of `invoke()` and set the `returnProxy` parameter to `true`.

The Message7 Example demonstrates Voyager's support for remote method invocations that return results by reference.

Dynamic Discovery

Finding or discovering other systems of interest remains a central issue for distributed systems. Voyager defines a collection of interfaces and abstract classes that define a

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

generic application-programming interface for finding other Voyagers. The next section describes the generic API. The following section describes an implementation that uses UDP multicast packets to advertise and listen for other Voyagers without prior knowledge of their identity or address.

Generic Application Programming Interface

The discovery code lives in Java package `com.recursionsw.ve.discovery`, and is composed of the following interfaces.

- **`IDiscoveryManager`, the methods implemented on the container for all available discovery implementations. The implementation instance is available from the default Voyager context by calling the `getDiscoveryManager()` method.**
- **`IDiscoveryService`, the methods for managing an implementation of a dynamic discovery service, including retrieving the name of the service, starting and stopping discovery announcement sending and receiving, managing listeners for dynamic discovery events.**
- **`IAnnouncement`, the methods implemented by a Voyager's announcement.**
- **`IServerDescription`, the methods describing a Voyager's identity and available `ServerContext`s.**
- **`IAnnouncementMarshaller`, the methods implemented by the class that builds `IAnnouncement` instances from the Voyager `ServerContext`s.**
- **`IDiscoveryAnnouncementListener`, the methods implemented by a listener registered with `IDiscoveryService`, and which is notified of each `IAnnouncement` received.**
- **`IDiscovered`, the methods implemented by an implementation class that maintains a collection of recent announcements. The implementation of this interface relieves the application of the need to manage a collection of available Voyager systems. The default instance is available by calling `IDiscoveryManager`'s `getDiscovered()` method.**
- **`IDiscoveredListener`, the methods that must be implemented by a listener registered with `IDiscovered`. A method in this interface is called when the `IDiscovered` implementation adds a new Voyager, and a different method is called when an existing Voyager's announcement is removed.**

Using the Generic API

Voyager creates the `IDiscoveryManager` implementation during startup. An application intending to use a discovery service should first check the result returned by `IDiscoveryManager`'s `getDiscoveryServices()` or `getDiscoveryService(String)` to see if the desired implementation is already available. If not, the application should construct the implementation and call `IDiscoveryManager`'s `registerDiscoveryService(IDiscoveryService)` to tell `IDiscoveryManager` about it.

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

Once the `IDiscoveryService` implementation is available, the application can interact with dynamic discovery in any of the following ways.

- **The application can create a listener for discovery announcements and register the listener with the discovery service by calling `IDiscoveryService`'s `registerAnnouncementListener(IDiscoveryAnnouncementListener)` method. This results in a notification each and every time a discovery announcement is received. This approach requires the application to manage knowledge of discovered Voyagers, since `IDiscoveryService` implementations maintain no history or discovery state.**
- **The application can create a listener for discovered Voyager adds and deletes and register it with `registerListener(IDiscoveredListener)`, found in `IDiscovered`. This approach relies on the `IDiscovered` implementation to maintain a collection of discovered Voyagers, and to purge announcements that exceed a specified age.**
- **The application can call `IDiscovered`'s `getListOfDiscoveredVoyagers()` when it needs to look for another Voyager. Again, this approach relies on the `IDiscovered` implementation to maintain a collection of discovered Voyagers.**
- **The application can manage announcing its Voyager's `ServerContext`s by calling `IDiscoveryService`'s `startAnnouncementSenders()` and `stopAnnouncementSenders()` methods.**

Registering or unregistering a discovery service with the `IDiscoveryManager` implementation also does the same operation on the default implementation of `IDiscovered`. This results in the `IDiscovered` implementation maintaining the announcement state of all known discovery services. Running an `IDiscoveryService` implementation without registering it `IDiscoveryManager` works, but is not recommended.
Methods in the dynamic discovery subsystem throw a `DiscoveryException` exception when they encounter a fault directly related to dynamic discovery.

Implementing Dynamic Discovery

Classes found in the `com.recursionsw.ve.discovery.impl` package provide a starting point for new realizations of the dynamic discovery application programming interfaces. The API documentation for the following classes describes usage details.

- `AbstractDiscoveryService` is an abstract base class implementing `the IDiscoveryService interface. This class knows nothing of the mechanism used by the discovery implementation, other than providing the mechanisms for periodically sending an announcement and managing listeners.`
- `AbstractDiscoveryServiceSenderReceiver` is an abstract base class that extends `AbstractDiscoveryService`. This class assumes the discovery implementation requires separate activities to send and receive announcements. The implementation manages collections of sender and receiver configurations

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

- `Announcement` implements `IAnnouncement` and is the concrete class delivered to the `IDiscoveryService` listeners.
- `ServerDescription` implements `IServerDescription` and is the concrete class used by `Announcement` to describe a single Voyager server, i.e., a `ServerContext`.

While a new dynamic discovery implementation could start from the interfaces, most will extend some or all of the classes described above.

Using UDP Dynamic Discovery Implementation

This dynamic discovery implementation, which is contained in the core .jar file and .NET assembly, sends and receives announcements using multicast UDP packets. The primary class, and the only class an application must explicitly construct, is `UDPDiscoveryService`, found in the package `com.recursionsw.ve.discovery.impl.udp`. The default no-argument constructor builds an instance using the defaults defined in the class as constants. The `UDPAnnouncementMarshaller` class, an implementation of `IAnnouncementMarshaller`, builds the content of each announced server.

The serializable class `UDPAnnouncement` extends `Announcement`, and is the class serialized to create an announcement that can be transmitted using a UDP packet. Due to limits imposed by UDP, a serialized UDP announcement, including all packet overhead, cannot exceed 65,535 bytes.

The `UDPDiscovery1` and `UDPDiscovery2` examples, found in the `examples.discovery` package, illustrate how to set up and use UDP dynamic discovery.

The `UDPDiscoveryService` class offers a public static method named `startDefaultUDPDiscoveryService()` that is suitable for invocation during an application's startup initialization. This parameter-less method constructs an `UDPDiscoveryService`, registers it with the `IDiscoveryManager`, and instructs the service to start sending and receiving UDP-based discovery announcements. Adding the following line to a Voyager property file will result in UDP discovery starting when Voyager starts, using the same default configuration calling `startDefaultUDPDiscoveryService()` starts.

```
Voyager.discovery.impl.udp.UDPDiscoveryServiceInstaller.install=true
```

Note that while UDP discovery has been implemented on all platforms on which Voyager runs, some platforms, especially Java ME devices, do not support UDP multicast. As a result, Voyager's UDP-based discovery implementation fails on some platforms because of the platform's limitations.

Using Multicast and Publish/Subscribe

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.

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

- Voting systems use a distributed messaging feature (multicast) to poll voters around the world for their views on a particular matter.

- News services use a distributed publish/subscribe feature to send news events only to readers who are interested in the broadcast topic.

Most traditional publish-subscribe systems use a single repeater to replicate a message or event to each subscriber. This approach is appropriate for smaller systems, but does not scale well, especially in distributed applications. Voyager uses a scalable architecture for message/event propagation called Space.

Understanding the Space Architecture

A `Space` is a logical container that can span multiple virtual machines across the network. A `Subspace` is the basic element of a distributed Space. A `Space` is created by linking one or more `Subspaces` together, and the content of a Space is the union of the content of its linked `Subspaces`.

A message/event is sent into a `Space` by publishing it to a `Subspace` in that `Space`. The message is cloned to each of the `Space's Subspaces` and delivered to every object (subscriber) in the local `Subspace`, resulting in a rapid, parallel fan-out of the message to every member of the `Space`. As the message propagates, it leaves behind a marker unique to that message which prevents the message from being re-propagated if it re-enters a `Subspace` it has already visited, that is, a message is delivered exactly once. This mechanism allows you to connect `Subspaces` to form arbitrary topologies without the possibility of multiple message delivery.

Understanding the Space Implementation

Four interfaces describe behaviors of `Spaces`. Two different message transport implementations offer differing quality of service properties.

Methods found in `ISubspaceMessaging` support messaging and `Subspace` contents. The `ISubspace` interface extends `ISubspaceMessaging` and contains methods supporting maintenance of `Subspace` listeners. Finally, `ITcpSubspaceConnections`, which extends `ISubspace`, contains methods for managing the topology of `Subspaces`, using TCP as the implementation transport. The class `TcpSubspace` implements `ITcpSubspaceConnections` and communicates using the TCP transport `TcpTransport`. Alternatively, `IJmsSubspaceConnections`, which also extends `ISubspace`, contains methods for managing the topology of `Subspaces` using a Java Messaging System implementation. The `IJmsSubspaceConnections` interface is realized by `JmsSubspace`. In the following paragraphs `Subspace` refers to a conceptual part of a `Space`, while `TcpSubspace` or `JmsSubspace` refers to the concrete realization of the concept.

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

Compared to `TcpSubspace`, `JmsSubspace` offers a higher quality of service in terms of message delivery, somewhat slower message propogation, and in some cases, integrates with an organization's existing messaging middleware. Using `JmsSubspace` a also requires some administration, e.g., creating a JMS topic for each JMS-based `Subspace`.

Using TCP Spaces

Space Topologies

The topology of a `Space` depends on the needs of the application and the environment in which it will run. Factors that influence this include:

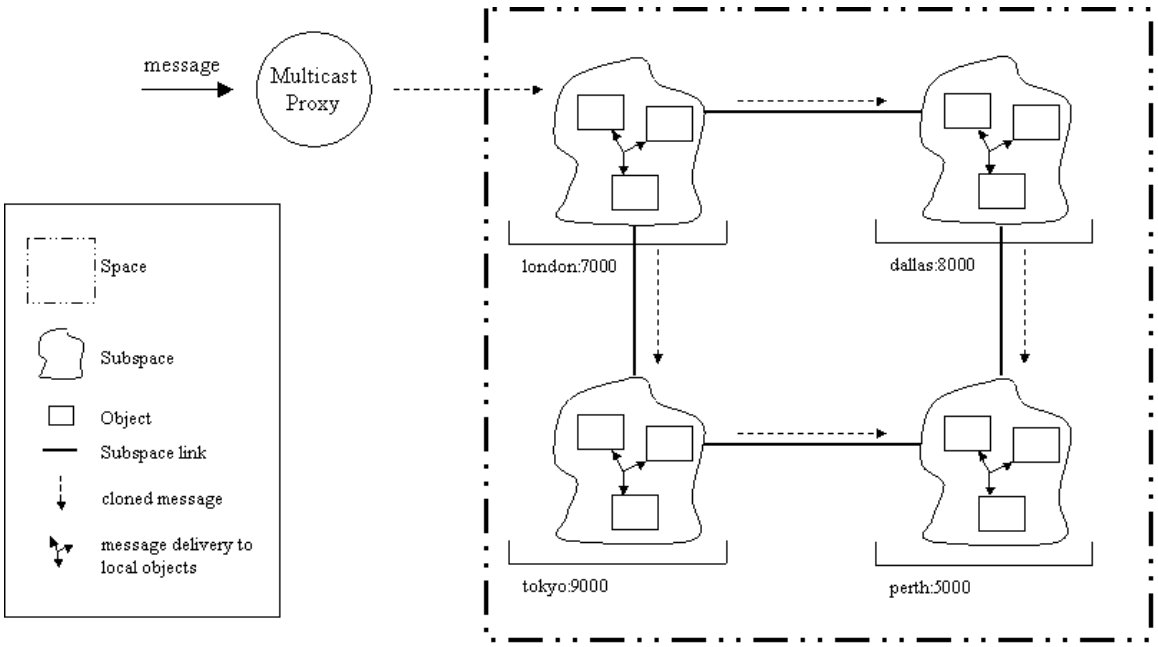
- Where messages or events are generated.
- Network reliability.
- The impact of a `Subspace` becoming unavailable.
- The rate at which events or messages are generated.
- The size of the events or messages published.

In most applications, a star or double-star topology is the most effective topology, providing effective message propagation while minimizing excessive use of network bandwidth. In this configuration, the server's `TcpSubspace` is connected to each client's `TcpSubspace`, but client `TcpSubspace` are not interconnected. If there are multiple servers, their `TcpSubspace` are connected. Events or messages are typically created on the server and are efficiently propagated to each client unless the client becomes disconnected from the server.

In a peer-to-peer application, a more effective topology is for each peer's `Subspace` to be connected to a small number of other peers. In this topology, messages can be created by any peer. Efficient and reliable propagation of messages through the `Space` is ensured through multiple connections.

The following diagram illustrates sending a message to a `TcpSubspace` in a `Space`.

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved



Creating and Populating a Space

To create a logical `Space` and populate it with objects, follow these steps:

1. Construct one or more `TcpSubspace` objects. Each `TcpSubspace` can reside anywhere in the network, allowing a single `Space` to span multiple programs.

   ```
   ITcpSubspaceConnections subspace1 = new TcpSubspace();
   ITcpSubspaceConnections subspace2 = new TcpSubspace();
   ```

2. Use the `subspace1.connect(subspace2)` method to connect the `TcpSubspaces` in a logical `Space`. Each connection is bi-directional; that is, if you connect `subspace1` to `subspace2`, you need not connect `subspace2` to `subspace1`. (If you do, the second connection attempt will be ignored.) Because the `connect` method requires an argument of type `TcpSubspace`, a `JmsSubspace` cannot connect to a `TcpSubspace`.

3. Use the `subspace1.add(object)` method to add one or more objects to each `Subspace`.

4. You can add different types of objects, including proxies, `JmsSubspaces`, and `TcpSubspaces`, into a `Space`.

Note: Creation and connection of `TcpSubspaces` can be done in any sequence.

You can manipulate `Subspaces` using additional methods defined in `ITcpSubspaceConnections`, including:

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

1. `disconnect(ITcpSubspaceConnections subspace )`

Disconnects two `TcpSubspace`'s. Like the `connect()` method, `disconnect()` is symmetric.

2. `getNeighbors()`

Returns an array of proxies to all neighboring `TcpSubspaces`.

3. `isNeighbor( ISubspace subspace )`

Returns `true` when the specified `ISubspace` is a neighboring `ISubspace`.

Refer to the API documentation for the `com.recursionsw.ve.space.ISubspaceMessaging`, `com.recursionsw.ve.space.ISubspace`, and `com.recursionsw.ve.space.ITcpSubspaceConnections` interfaces for the complete list of features available.

Using JMS Spaces

Space Topology

Unlike TCP-based `Spaces`, the topology of JMS-based `Spaces` depends entirely on the topology of the JMS servers. From the perspective of a `Space` the topology appears flat in the sense that every `JmsSpace` is an immediate neighbor of every other `JmsSpace` connected using the same URL.

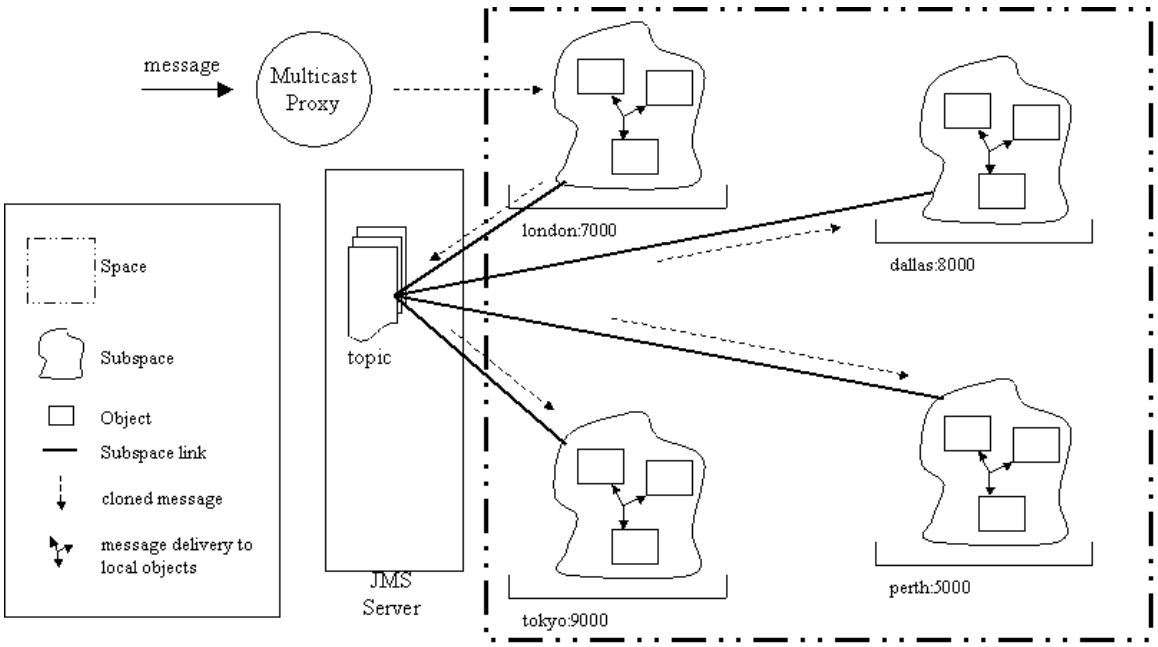
A JMS-based `Space` is defined by a URL of the following form.

```
jms:/topic/topicpath;topic
```

- `jms` is the transport protocol
- `/topic/topicpath` is the name of the JMS topic associated with the `Space`, i.e., the name of the Space is the JMS topic name. This example shows a simple single level JMS topic name, where a company's JMS topic names are very likely to be hierarchical.
- `;topic` marks the URL as referring to a JMS topic rather than a JMS queue.

The following diagram illustrates sending a message to a `JmsSubspace` in a `Space`.

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved



Creating and Populating a Space

To create a logical `Space` using JMS as the transport mechanism and populate it with objects, follow these steps:

1. Construct one or more `JmsSubspace` objects. Each `JmsSubspace` can reside anywhere in the network, allowing a single `Space` to span multiple programs.

2. Use the `subspace1.connect("jms:/topic/testTopic;topic")` method, found in `IjmsSubspaceConnections`, to connect each `JmsSubspace` to the `Space`. The URL names the `Space` the `JmsSubspace` is joining. Because the `connect` method requires an argument of type `String`, a `TcpSubspace` cannot connect to a `JmsSubspace`.

3. Add members to the `JmsSubspace`, exactly as is done for `TcpSpaces`.

4. Exactly as is done when using `TcpSpaces`, you can add different types of objects, including proxies, `JmsSubspaces`, and `TcpSubspaces`, into a `Space`.

You can manipulate `JmsSubspaces` using additional methods defined in `IJmsSubspaceConnections`, including:

```
disconnect(String subspaceURL  )
```

Disconnects a `JmsSubspace`.

Configuring JMS Spaces

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

The JMS transport is not automatically loaded when Voyager starts. As an optional transport module, the JMS transport implementation is usually loaded by adding to the Voyager configuration file the following lines. (See the Specifying a Properties File section for configuration file details.)

```
Voyager.transport.Transport.register=#Voyager.jmstransport.JmsTransport
com.recursionsw.ve.jmstransport.JmsTransport.setInitialContextPropertyF
ile = initialcontext.properties
```

The first line tell Voyager to dynamically load the JMS transport module. The second line points to another property file describing how to locate the initial JMS naming context, and contains three lines such as seen below. Alternatively, the following lines can be directly substituted for the above line specifying the second property file.

```
java.naming.factory.initial=org.jnp.interfaces.NamingContextFactory
java.naming.provider.url=jnp://localhost:1099
java.naming.factory.url.pkgs=org.jboss.naming:org.jnp.interfaces
```

These lines, which work for JBoss' JMS implementation, enable the JMS transport to resolve names found in jms: URLs, e.g., `/topic/testTopic` as seen in the previous discusson of `JmsSubspace` URLs.

Nested Spaces

You can nest `Spaces` by adding a (possibly remote) `ISubspace` as an element of another `Subspace`, instead of connecting them. Operations on the containing `Space`, such as multicasting and publish/subscribe, are propagated automatically to the contained `Spaces`, allowing you to group smaller `Spaces` into a single logical `Space`. Multicasts and publications originating in the contained `Space` are not propagated to the containing `Space`, i.e., the connection is one-way only. This one-way connection provides an additional level of flexibility when designing `Space` topologies.

The Space1 Example demonstrates creating and populating a distributed `Space`.

Subspace Event Listeners

A `Subspace` generates a `SubspaceEvent` when neighbors are connected or disconnected and when objects are added to or removed from the `Subspace`. You can listen for these events with a `SubspaceListener`. The `SubspaceListener` interface declares one method that your listener must implement:

```
void subspaceEvent( SubspaceEvent event );
```

`Subspace` events, defined as contstants in `SubspaceEvent`, are:

`ADDING`: An object was added to the `Subspace`.

`REMOVING`: An object was removed from the `Subspace`.

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

`CONNECTING`: The `Subspace` is being connected to another `Subspace`.

`CONNECTED`: The `Subspace` was successfully connected to another `Subspace`.

`DISCONNECTING`: The `Subspace` is being disconnected from another `Subspace`.

`DISCONNECTED`: The `Subspace` was successfully disconnected from another `Subspace`.

There are two events generated for each connection or disconnection. Because connects and disconnects are symmetric, both `ISubspaces` must successfully perform the action before it is considered complete. The `CONNECTING/DISCONNECTING` events are generated at the beginning of the action, and the `CONNECTED/DISCONNECTED` events are generated only if the action successfully completes.

Multicasting

You can multicast a message to a group of objects in a `Space` using a multicast proxy provided by a method found in `ISubspaceMessaging`.

- `getMulticastProxy( String classname )`

  Returns a multicast proxy that is type-compatible with the specified class or interface. Messages sent to this proxy are multicast to every object in the `Space` that is an instance of the specified class or interface. Multicast messages return `false` , `\0`, `0` or `null` depending on the return type. You can create any number of multicast proxies with different types to the same logical Space, even to the same `Subspace` within a `Space`.

Multicast messages are always automatically propagated to nested `Subspaces`.

The Space2 Example demonstrates typesafe multicasting of messages and JavaBeans events to objects in a `Space`.

Publishing and Subscribing Events

To publish an event associated with a topic to every object that implements `PublishedEventListener` in a Space, use `Publish.invoke( ISubspace subspace, EventObject event, Topic topic ). PublishedEventListener` defines a single method `publishedEvent( EventObject event, Topic topic)` that receives every published event in the `Space`. The listener must handle the event in the appropriate manner.

A topic is specified hierarchically with fields separated by periods, like `sports.bulls` and `books.fiction.mystery`. The asterisk (`*`) wild card matches the next field, and the left angle bracket (<) matches all remaining fields. For example, `games.soccer.goals`

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

matches `games.soccer.*`, `games.*.goals` and `games.<`. Both publishers and subscribers can use wildcards to match against a range of topics.

An object can subscribe to events in three ways.

1. An object can implement `PublishedEventListener` and add itself to a `Space`. It then receives every event that is published to the `Space` and must perform additional filtering and processing as necessary.

2. An object can use an instance of `Subscriber` to listen to the `Space` on its behalf and perform event filtering/forwarding. A `Subscriber` implements `PublishedEventListener` and has methods for subscribing/unsubscribing to topics. It also contains a reference to another `PublishedEventListener`. When a `Subscriber` is added to a `Space`, it forwards any published event that matches a topic to its associated `PublishedEventListener`. The `PublishedEventListener` does not have to be in the same program as the `Subscriber`. For example, to perform server-side filtering, set the `Subscriber`'s `PublishedEventListener` to a local intermediary object that performs additional processing and then forwards the event, if appropriate, to its final remote destination.

3. An object can use dynamic aggregation, add a `Subscriber` facet, and then add the facet to the `Space`. The `Subscriber` facet forwards all selected events to the primary object, which must implement `PublishedEventListener`.

Published events are always automatically propagated to nested `Subspaces`.

Note: Subscriber objects must be manually removed from a subspace when the client disconnects, otherwise they will be orphaned on the server and never garbage collected.

The Space3 Example demonstrates publishing events to subscribers in a `Space`.

Administering a Space

By default, an `ISubspaceMessaging` instance does nothing when its objects and neighbors are disconnected or killed. You can instruct an `ISubspaceMessaging` instance to purge itself of disconnected or dead objects and neighbors by using the following `ISubspaceMessaging` methods.

- `setPurgePolicy( byte policy )`

Sets a `Subspace's` purge policy. Four policies are available.

1. `ISubspaceMessaging.DIED` removes proxies to objects and neighboring `Subspaces` that have been garbage-collected. A `Subspace` knows an object is

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

dead when an `ObjectNotFoundException` is thrown as a result of sending a message to the object.

2. `ISubspaceMessaging.DISCONNECTED` removes proxies to objects and neighboring `Subspaces` that are not reachable. A `Subspace` knows an object is disconnected when an `IOException` is thrown as a result of sending a message to the object.

3. `ISubspaceMessaging.ALL` removes proxies to dead and disconnected objects and neighbors.

4. `ISubspaceMessaging.NONE`, the default policy, ignores dead and disconnected proxies.

- `getPurgePolicy()`

Returns the purge policy assigned to a `Subspace`.

- `purge( byte policy )`

Forces a `Subspace` to be purged immediately.

A `Subspace` automatically purges itself according to its purge policy each time a message is delivered.

A `TcpSubspace` propagates events to remote `TcpSubspace` in a separate thread. This propagation mechanism is designed for a high degree of scalability and fault tolerance. There are several parameters that can be used to fine-tune the propagation mechanism. These parameters can be supplied as standard properties and read on startup, or set through methods in the `com.recursionsw.ve.space.PropertyHelper` class. Note that changed parameters only apply to newly created `TcpSubspaces`.

- `subspaceConnectorMaxQueueSize` = 0+ events (default: 0)

Each `TcpSubspace` has a queue to hold events for delivery to a neighboring `TcpSubspace`. This parameter configures the maximum size of the queue. Setting this to a non-zero value N will force events to be discarded in the event that the queue reaches a size of N. This prevents the queue from unbounded growth in the case of overwhelming event publication, at the cost of losing events. If you require a more advanced queue management strategy, use the `getQueueSize()` method found in `ISubspaceMessaging`.

- `subspaceConnectorLogging` = {true|false} (default: false)

This parameter controls whether informational/debug messages are logged to the Voyager console. Enable this to obtain detailed information about the behavior of the queue.

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

- `rescheduleSubspaceConnector` = {true|false} (default: true)

The delivery of the queue associated with a connected `TcpSubspace` requires a separate thread that is allocated from Voyager's thread pool. This parameter determines what happens when the queue is empty (all events have been delivered). If false, the thread will block indefinitely until at least one new event is added to the queue. If true, the thread will block for a configurable amount of time for new event(s) to be added to the queue. If the specified time elapses with no new events to deliver, the thread will be returned to Voyager's thread pool. The advantage of rescheduling is that the VM will typically require fewer threads to operate. This can be important if a `TcpSubspace` has a large number of neighbors, because propagation to each neighbor requires a separate thread. The advantage of not rescheduling is that events added to the queue will be delivered immediately, instead of waiting for a thread to be acquired from the thread pool.

- `subspaceConnectorDeliveryThreadWaitTime` = 0+ ms (default: 1000)

This parameter is only effective if `rescheduleSubspaceConnector` is enabled (true). It determines how long the queue delivery thread will wait for new events before returning to the Voyager thread pool. When setting this value, consider the rate of event publication: if there are short delays between event publication, and this property is set to a low value, it is likely that threads will return to the thread pool only to be immediately called on to deliver new events. Conversely, if there are long delays between event publications, and this property is set to a high value, threads will probably be idle for a long period of time instead of being returned to the thread pool. It is recommended that this property be set to between 500ms and 10000ms.

- `enableSubspaceConnectorMonitor` = {true|false} (default: false)

If this parameter is set, a thread delivering events to a neighboring `TcpSubspace` is monitored for network/connection problems. If there are problems with the delivery (excessive delays or exceptions), the queue is first disabled. In this state it will no longer accept new events for delivery. If there are further problems, the connection between the `TcpSubspace` is broken. If the delivery thread recovers, the queue is re-enabled and will begin accepting new events. The monitoring is performed by a thread that is notified on a periodic interval.

- `subspaceConnectorMonitorTimerDelay` = 0+ ms (default: 1000)
- `subspaceConnectorDisableDelay` = 0+ ms (default: 10000)
- `subspaceConnectorDeactivateDelay` = 0+ ms (default: 10000)

These three parameters determine the behavior of the thread monitoring the event propagation threads. First, the `subspaceConnectorMonitorTimerDelay` property determines the time interval at which the delivery threads are checked for a "hang-during-delivery" condition. The other two properties set the timeout delays for disabling and deactivating the event queue. If the delivery thread is in the "delivering" state for more

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

time than specified in `subspaceConnectorDisableDelay`, it will be disabled. The queue will no longer accept new events. If the `subspaceConnectorDeactivateDelay` time then expires, the queue will be deactivated: the connection between the two `Subspaces` is broken. However, if the delivery thread successfully recovers before the deactivation timeout, the event queue is re-enabled.

Mobility and Agents

Mobility allows you to move objects that exchange large numbers of messages 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 locality optimization. In addition, 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.

In addition to standard mobility support, Voyager also supports mobile autonomous agents, which are objects that move themselves in order to achieve their goals. Finally, Voyager's VM can be grouped together to form a cluster of distributed processes that can receive agents and execute them. This gives you the capability to perform distributed load balancing, task scheduling and load management.

In this section, you will learn to:

- Move an object to a new location
- Obtain move notification
- Understand uses for mobile agents
- Create mobile agents
- Deploy mobile agents

Moving an Object to a New Location

To move an object to new location, use `Mobility.of()` to obtain the object's mobility facet (see the Dynamic Aggregation™ chapter) and then use the methods defined in `IMobility`.

- `moveTo( ClientContext destination )`

Moves to the program referenced by the specified `ClientContext`.

- `moveTo( Object destination )`

Moves to the program that contains the specified object. The object is usually specified as a proxy.

For example, the following code creates a `StockMarket` at `//dallas:8000` and then moves it to `//tokyo:9000` :

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

```
String className = "examples.stockmarket.Stockmarket";
VoyagerContext voyagerContext = null;
VoyagerContext = Voyager.starup();
ClientContext ccDallas =
voyagerContext.acquireClientContext("Dallas");
ccDallas.openEndpoint("//dallas:8000");
ClientContext ccTokyo =
voyagerContext.acquireClientContext("Tokyo");
ccTokyo.openEndpoint("//tokyo:9000");

aFactory = ccDallas.getFactory();
// send message to initial location
IStockmarket market = (Istockmarket)aFactory.create(className);
market.news( "at first location" );
// obtain mobility facet
IMobility mobility = Mobility.of( market );
mobility.moveTo( ccTokyo ); // move the object to a new location

// the last two lines could be written in a single line as
Mobility.of( market ).moveTo( ccTokyo )
// message is delivered to new location
market.news( "at second location" );
```

The `moveTo()` method causes the following sequence of events to occur.

1. Any messages that the object is currently processing are allowed to complete and any new messages that arrive at the object are suspended. Mobility can only detect method calls that are made through Voyager `Proxy` objects, so do not attempt to move an object that might be executing methods that were invoked directly.

2. The object and all of its non-transient parts are copied to the new location using serialization, ignoring pass-by-reference tags like `com.recursionsw.ve.IRemote`. An exception is thrown when any part of the object is not serializable or when a network error occurs. To avoid copying a particular part as an object, store a proxy to the part instead.

3. The new addresses of the object and all of its non-transient parts are cached at the old location.

4. The old object is destroyed.

5. Suspended messages sent to the old object are resumed.

6. When a message sent via a proxy arrives at the old address of a moved object, a special exception containing the object's new address is thrown back to the stale proxy. The proxy traps this exception, rebinds to the new address, and then

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

resends the message to the updated address. If the program at the old location crashes before a stale proxy is updated, the stale proxy is unable to successfully rebind and a message sent via the proxy generates an `ObjectNotFoundException`.

7. The `moveTo()` returns after the object is successfully moved or when a mobility exception occurs. If an exception occurs, the old object is restored to its original condition, suspended messages are resumed, and the exception is rethrown wrapped in a `MobilityException` .

The rules for garbage collection are not affected by mobility. A moved object is reclaimed when there are no more local or remote references to it. The new addresses cached at the old location are not treated as references by the garbage collection system. **Note:** It is unsafe to move an object when local references point to it from outside the context of Voyager or when the object has one or more threads not associated with a remote message.

It is also necessary that Voyager know the proper name of the local machine as it is known to itself and all other Voyager servers to which any mobile agents may be moved. If you experience problems with Voyager not detecting the machine's hostname correctly, the machine's hostname should be explicitly stated when starting Voyager.

The Mobility1 Example creates a `Drone` object and then moves it between programs.

Obtaining Move Notification

Sometimes an object needs to know that it is about to move or has just been moved. For example, a persistent mobile object may need to remove itself from the origin's persistent store and add itself to the destination's persistent store. Voyager provides this capability through the `IMobile` interface. If an object or any of its parts implements the `IMobile` interface, they will receive callbacks during a move in the following order.

- `preDeparture( String source, String destination )`

This method executes on the original object at the source. If the method throws a `MobilityException` , the move aborts and no more `IMobile` callbacks occur.

- `preArrival()`

This method is executed on the copy of the object at the destination. If the method throws a `MobilityException` , the move is aborted and no more `IMobile` callbacks occur.

- `postArrival()`

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

At this point, the copy of the object becomes the real object, the object at the source becomes the stale object, and the move is deemed successful and cannot be aborted. This method executes on the copy of object at the destination immediately prior to the user-supplied callback. It is typically defined to perform activities such as adding the new object into persistent storage.

- `postDeparture()`

This method executes on the original stale object at the source. It is typically defined to perform activities such as removing the stale object from persistence. Messages sent to the stale object via a proxy are redirected to the new object, so `postDeparture()` should not use proxies to the original object or any of its facets. Because the user-supplied callback on the new object is executed using a fresh thread, it is possible for this `postDeparture()` to be executing concurrently with the user-supplied callback.

The Mobility2 Example creates a mobility-aware `Drone2` object and then moves it between programs. Also see the Understanding the Uses for Mobile Agents section for another example.

Understanding the Uses for Mobile Agents

A mobile autonomous agent is an object that moves itself around the network in order to achieve its goals. You can use mobile agents as follows:

- 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 VM.

- If a program needs to send a large number of messages to objects in remote VMs, an agent can be constructed to visit each VM in turn and send the messages locally. Local messages are often between 1,000 and 100,000 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 VMs and communicate with each other to achieve the final goal.

- If periodic monitoring of a remote object is required, creating an agent that moves to the remote object and monitors it locally is more efficient that monitoring the object 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 cell phone or PDA, then an agent can move into the device, perform its task, and move back into the network only when necessary.

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

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, you may find many ways to agent-enhance your current and future programs.

Creating Mobile Agents

To make an object a mobile autonomous agent, use `Agent.of()` to obtain the object's agent facet (see the Dynamic Aggregation™ chapter) and then use the methods defined in `IAgent`. NOTE: The Agent facet is distinct from the Agent class discussed below in the AgentSpaces section.

- `moveTo( ClientContext clientContext, String callback [, Object[] args ] )`

Moves to the program with the specified context and then restarts by executing an oneway callback with optional arguments. A `MobilityException` is thrown when the callback method is not found or is not public.

- `moveTo( Object object, String callback [, Object[] args ] )`

Moves to the program containing the specified object and then restarts by executing a oneway callback with a proxy to the object as the first argument and the optional arguments as the remaining arguments. A `MobilityException` is thrown when the callback method is not found or is not public.

- `setAutonomous( boolean flag )`

If the flag is true, become autonomous. An autonomous agent is not reclaimed by the garbage collector even if there are no more local or remote references to it. An agent is initially autonomous by default, and typically executes `setAutonomous( false )` when it has achieved its goal and wishes to be garbage collected.

- `isAutonomous()`

Return true if this agent is autonomous.

- `getHome()`

Return the home of this agent, which is defined to be the URL of the agent when its agent facet was first accessed.

For example, an object can move itself to `//dallas:8000` and restart using `atDallas()` by executing `Agent.of( this ).moveTo( dallasClientContext, "atDallas" );`

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 method in the agent after it moves. Therefore, only exception-handling code should follow a `moveTo()`.

The Agents1 Example constructs a `Trader` agent that works on the stockmarket from a remote location and then moves itself to the stockmarket to work locally.

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

Code Mobility

There are three ways to make an agent's class files available to a host to which the agent may be traveling:

- Pre-install all the class files in the remote host's `CLASSPATH`. In a large system, this method requires dealing with maintenance issues. Most mobile platforms require this solution, since they typically cannot dynamically load object code.
- Keep all system classes in a single repository. This method requires that all remote hosts bootstrap the location of the resource repository at startup. See the Loading Classes and Serving Classes sections in the Voyager Basics chapter for additional information on resource loaders. In most cases, this option is preferred when managing a homogeneous system.
- Have an agent register a resource loader before it arrives. This method allows an agent to carry its class files and resources as it moves through the network.

The Voyager class library ships with two `IResourceLoader` implementations.

- `URLResourceLoader`

This resource loader takes a `java.net.URL` class on the constructor. The `URL` instance may reference the host that the agent is being launched from or a simple repository. For example, `http://classes.home.com:8000/` .

- `ArchiveResourceLoader`

This resource loader is similar to the `URLResourceLoader`, except the URL expected on the constructor must point to a .jar or `.zip` file. For example, `http://classes.home.com:8000/jars/networkagent.jar`. The `ArchiveResourceLoader` does not retrieve the remote jar until the jar itself or its resources holder is requested. This reduces network traffic in case an instance of this resource loader is already installed on the remote host.

To set and retrieve an agent's resource loader, invoke the following methods on `IAgent` :

- `setResourceLoader( IResourceLoader resourceLoader )`

Indicates to the agent to use the given `IResourceLoader` instance when loading its resources.

- `getResourceLoader()`

Returns the registered `IResourceLoader` instance used by the agent. If additional resources are stored in a resource loader other than class files, such as certificates or sound files, the agent can access them directly using the `IResourceLoader` interface.

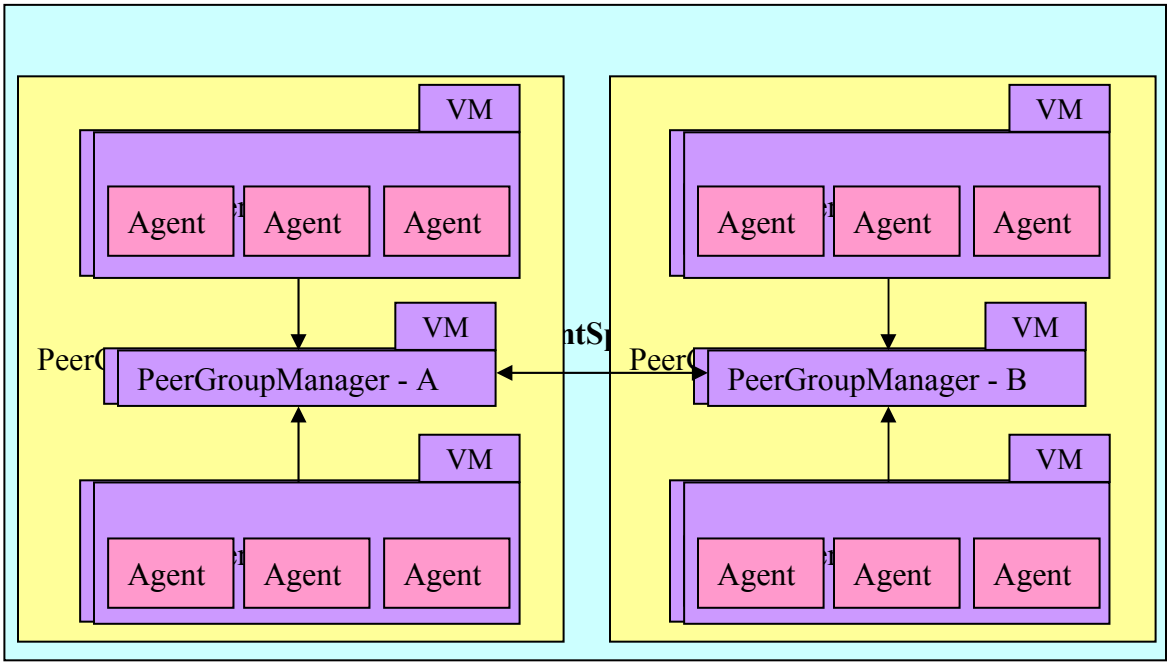
When an agent leaves a host, it always removes the resource loader it installed before it arrived. The `Voyager ClassLoader` maintains a reference count on each

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

`IResourceLoader` instance installed at a given priority. When the count reaches zero, the given resource loader is removed from the system.

AgentSpace

`Voyager` supports the creation and deployment of Agents into a cluster of `Voyager` VM's. This cluster is called an AgentSpace. An AgentSpace is a virtual, distributed group of VM's that provide their services to executing agents. An AgentSpace is composed of one or more PeerGroups. A PeerGroup is managed by a PeerGroupManager and is in turn composed of AgentPeer's. Each AgentPeer hosts and executes agents. The following diagram illustrates these containment relationships:



Creating and Using AgentSpaces

The primary interface for working with AgentSpaces is the class `com.recursionsw.ve.agentspace.AgentSpace`. The `AgentSpace` class contains methods to create `IPeerGroupManagers`, `IAgentPeers` and `Agents`, and deploy `Agents`. **Note:** The Agent class in AgentSpace is distinct from the Agent facet.

```
public AgentSpace(String agentSpaceName)
public AgentSpace(String agentSpaceName, Object securityPrincipal)
public AgentSpace(String agentSpaceName, Object securityPrincipal,
      AgentPrototype defaultAgentPrototype)
```

These constructors create an instance of `AgentSpace`. An `AgentSpace` always has a name. If it has a security principal, that principal is used when connecting to (or creating)

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

a `PeerGroupManager` for security authentication. If a default `AgentPrototype` is provided, that prototype is used when creating an Agent (see below).

```
public IPeerGroupManager createPeerGroupManager(String groupName)
```

This method creates a new `PeerGroupManager` to manage a `PeerGroup`. Use this method when creating the first `PeerGroup` of an `AgentSpace`.

```
public IPeerGroupManager createPeerGroupManager(String groupName,
      String remotePeerGroupManagerUrl)
public IPeerGroupManager createPeerGroupManager(String groupName,
      IPeerGroupManager remotePeerGroupManager)
```

These methods create a new `PeerGroupManager` that will be connected to the specified `PeerGroupManager`. The first method attempts to look up the `PeerGroupManager` using the provided URL, and the second method uses the provided `PeerGroupManager` directly. Use these methods to create a new `PeerGroup` that is connected to the `PeerGroup` managed by the provided `PeerGroupManager`.

```
public IAgentPeer createPeer(IPeerGroupManager peerGroupManager)
public IAgentPeer createPeer(String peerName, IPeerGroupManager
      peerGroupManager)
```

Use these methods to start a new `AgentPeer` that is a member of a `PeerGroup`. The `AgentPeer` will join the `PeerGroup` managed by the specified `PeerGroupManager`.

```
public Agent createAgent(IAgentAction agentAction)
public Agent createAgent(AgentPrototype agentPrototype)
public Agent createAgent(AgentPrototype agentPrototype, IAgentAction
agentAction)
```

These methods provide several ways to create a new `Agent`. An `Agent` always has an action associated with it (an implementation of `IAgentAction`). The action implements the logic that will be executed when the Agent arrives at an `AgentPeer`. If provided, the `AgentPrototype` contains additional settings for the agent. If none is provided, the default `AgentPrototype` will be used.

```
public void deployAgent(Agent agent, IPeerGroupManager
      peerGroupManager)
```

The `deployAgent()` method takes the specified agent and deploys it to an `AgentSpace` using the specified `PeerGroupManager`. On deployment, the agent is serialized and passed to (first) the `PeerGroupManager` and (second) from the `PeerGroupManager` to an `AgentPeer` in the `PeerGroup`. Once it arrives at the `AgentPeer` it will be executed.

Agent Action Execution

An Agent's action (implementation of `IAgentAction`) is executed when the agent is deployed to an AgentPeer. Each action is executed in its own thread. Actions must implement `com.recursionsw.ve.VSerializable` or `java.io.Serializable`.

The result of executing an action is an `Object`. This object may be any serializable object.

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

AgentSpace Events

Events are a primary mechanism for communication in an `AgentSpace`. You can register an event listener by calling `public void attachListener(PublishedEventListener listener, IPeerGroupManager peerGroupManager)`. This listener will receive all AgentSpace events, and can publish events to the AgentSpace.

An agent can also listen for and publish events. To set an event listener on an agent, either set it in the `AgentPrototype` prior to creating the agent or call `Agent.setEventListener(PublishedEventListener listener)`. Agent listeners will receive all `AgentSpace` events and can publish events to the `AgentSpace`.

AgentSpace Topologies

The topology of an `AgentSpace` is determined by the connections between `PeerGroupManagers` and the arrangement of `AgentPeers` into `PeerGroups`. The calls you make to `createPeerGroupManager` and `createPeer` determine the topology.

Agent Execution and Survivability

Each `PeerGroupManager` monitors the `AgentPeers` registered with it. A loss of connection to an `AgentPeer` (due to either network failure or failure of the `AgentPeer` VM) is detected. All `Agents` deployed to the failed `AgentPeer` are automatically re-deployed to another `AgentPeer` in the `PeerGroup`. Take this into consideration when implementing an agent action. When the action starts execution it should clean up external effects from any previous attempted execution of the action. For long-running agent actions, consider saving intermediate results to a database or file.

Each `AgentPeer` monitors execution of the agents deployed to it. If an agent action throws a `RuntimeException` or executes longer than its maximum allowed execution time the `AgentPeer` will re-execute the action. Note that in the case of a timeout, the thread executing the original action may still be running.

AgentSpace Security

Security considerations are often a key consideration for distributed applications. `AgentSpace` is designed to support security.

AgentSpace security centers on the implementation and usage of the `IAgentSpaceSecurityManager` interface. To install a security manager in a `PeerGroupManager`, implement this interface and call the `IPeerGroupManager.setAgentSpaceSecurityManager(IAgentSpaceSecurityManager agentSpaceSecurityManager)` method. This interface has two methods:

```
SecurityContext createSecurityContext(Object securityPrincipal) throws
PeerGroupException
```

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

```
void authorize(String action, SecurityContext securityContext) throws
PeerGroupException
```

The `createSecurityContext()` method is called when a new `PeerGroupManager` joins an `AgentSpace`. It takes a security principal (passed to the `AgentSpace` when it was created) and returns a `SecurityContext`. (Any serializable object can be a security principal.) Extend `SecurityContext` to provide the necessary information to authorize actions.

The `authorize()` method is called by the `PeerGroupManager` when the following actions are attempted:

- Joining an `AgentSpace`
- Registering an `AgentPeer`
- Deploying an `Agent`

The `SecurityContext` provided is the one the security manager created when `createSecurityContext()` was called.

To summarize, here are the steps for securing an `AgentSpace`:

1. Implement `IAgentSpaceSecurityManager`.
2. Extend `SecurityContext` to provide authorization information the security manager will use when authorizing actions.
3. Set each PeerGroupManager's security manager.
4. Use a security principal when creating an AgentSpace instance.

Yellow Pages Directory

When using the Naming Service, a well-known name is used to acquire a reference to a service. A client performing a Naming Service lookup is asking for the single service associated with a unique well-known name.

Voyager's Yellow Pages Directory provides another mechanism for acquiring a reference to a service. The Yellow Pages Directory provides a mapping between a *service description*, consisting of one or more name-value *service attribute*s, and a service. A Yellow Pages lookup is performed using a *discovery request* containing an expression to match against service descriptions. The Yellow Pages Directory returns all service descriptions that match the expression in the discovery request. A client performing a Yellow Pages lookup is asking for all the services that match a filter: the discovery request expression.

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

The building block of Voyager's Yellow Pages Directory is the com.recursionsw.ve.yp.YellowPages class, which implements the interface com.recursionsw.ve.yp.IYellowPages and provides the central API for most Yellow Pages features. Instances of the YellowPages class host a VM-local registry for services. A Yellow Pages Directory can be a single Yellow Pages instance or a distributed federation of inter-connected Yellow Pages instances.

The methods in the IYellowPages interface are described below:

- `connect(IYellowPages yellowPages)`

The connect() method connects two Yellow Pages instances. Each instance has a *service registry* that provides local storage for service descriptions. Connections are bi-directional: when `yp1.connect(yp2)` is called, `yp1` is connected to `yp2` and `yp2` is connected to `yp1`. Service descriptions are registered only in a single instance: they are not propagated to connected instance. Only discovery requests are propagated to the federation of instances.

- `disconnect(IYellowPages yellowPages)`
- `disconnect()`

The disconnect() methods disconnect a Yellow Pages instance from another Yellow Pages instance or from all instances it is connected to.

- `registerService(ServiceDescription serviceDescription)`
- `deregisterService(ServiceDescription serviceDescription)`

These methods register or deregister a service. The `ServiceDescription` provided includes a set of name-value service attributes and an `IServiceResolver` used by a client to obtain a reference to the service.

- `ServiceDescription[] lookup(DiscoveryRequest discoveryRequest) throws YellowPagesTimeToLiveException`
- `void lookup(DiscoveryRequest discoveryRequest, IDiscoveryListener discoveryListener) throws YellowPagesTimeToLiveException`

Perform a lookup. A lookup begins when a `lookup()` method is called with a *discovery request*. A discovery request contains a discovery request expression. Each term in the expression is a conditional test of an attribute in the service description, such as "equals" or "exists". The discovery request is propagated to the federation of Yellow Pages instances. Each instance applies the discovery request's expression to the service descriptions registered, and returns any matches to the client. The two `lookup()` methods provide, respectively, synchronous and asynchronous lookups. The

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

synchronous `lookup()` method returns matches as an array of `ServiceDescriptions`; the asynchronous `lookup()` method returns `ServiceDescriptions` to the `IDiscoveryListener` in a separate thread. (Note that all lookups are internally performed asynchronously; the first `lookup()` method internally simulates a synchronous lookup.)

If the time-to-live parameter, defined in the DiscoveryRequest, is less than or equal to 0, the YellowPagesTimeToLiveException is thrown. Please see the section titled, Performing a Yellow Pages Lookup, for more information.

Creating a Yellow Pages Directory

To create a Yellow Pages Directory, create or acquire one or more Yellow Pages instances and connect them using the `connect()` method:

```
String classname = YellowPages.class.getName();
Factory f8000 =
voyagerContext.acquireClientContext("Server8000").getFactory();
Factory f9000 =
voyagerContext.acquireClientContext("Server9000").getFactory();
IYellowPages yp1 = (IYellowPages) f8000.create(classname);
IYellowPages yp2 = (IYellowPages) f9000.create(classname);
yp1.connect(yp2);
```

It is common to use one Yellow Pages instance per VM. The `YellowPages` class provides several static methods to simplify acquiring and connecting instances in separate VMs based on the Singleton design pattern:

- `static IYellowPages getInstance()`

Acquire the default (singleton) Yellow Pages instance for this VM.

- `static IYellowPages getInstance(ClientContext cc)`

Acquire the default (singleton) Yellow Pages instance for the VM at the given `ClientContext`. This will call the static `getInstance()` method for the `YellowPages` class in that VM.

- `static void connect(ClientContext cc)`

Connect the default (singleton) Yellow Pages instance for this VM to the instance for the VM at the given `ClientContext`.

The `getInstance()` methods delegate to an implementation of `IYellowPagesFactory` to provide the actual `IYellowPages` instance. Use the static get/set methods provided in the `YellowPages` class to get or set this factory.

Registering a Service

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

Services are registered in a Yellow Pages Directory using a *service description*, implemented in the class `com.recursionsw.ve.yp.registry.ServiceDescription`. The service description contains a list of service attributes (name-value pairs) and a service resolver. The service resolver is used by the client that performing a lookup to obtain a reference to the service. The standard service resolver creates a proxy for the service and returns this proxy to the client.

`ServiceDescription` provides several constructors:

- `ServiceDescription()`

The default constructor, generally not used.

- `ServiceDescription(String name, Object service)`

Create a service description. The *name* parameter will be the name of the service. The *service* parameter is the service itself. The default service resolver will be used for resolving the service.

- `ServiceDescription(String name, IServiceResolver serviceResolver)`

Create a service description. The *name* is as above. The *serviceResolver* provides a reference to the service when its `resolve()` method is called (typically, by the client performing a lookup).

After creating a `ServiceDescription` it must be registered with a Yellow Pages instance. A service may be registered using multiple service descriptions; however, each service description must be unique.

Performing a Yellow Pages Lookup

A lookup in the Naming Service returns a single object (service) for a unique name. A Yellow Pages lookup returns zero or more `ServiceDescriptions` for a `DiscoveryRequest` containing a `DiscoveryRequestExpression`. A Yellow Pages lookup begins with creating the `DiscoveryRequest` and its associated `DiscoveryRequestExpression`:

```
DiscoveryRequest request = new DiscoveryRequest();
DiscoveryRequestExpression expr = new DiscoveryRequestExpression();
```

The next step is to add one or more conditional sub-expressions to the `DiscoveryRequestExpression`, typically using the `ExpressionFactory` helper class. The below example adds an “equals” sub-expression to test for an attribute named “myAttributeName” with a (String) value of “myAttributeValue”.

```
expr.add(ExpressionFactory.eq("myAttrName", "myAttrValue"));
```

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

Each Yellow Pages instance in the Yellow Pages Directory will test its registered `ServiceDescriptions` against this expression and return the matching `ServiceDescriptions`.

In addition to specifying sub-expressions you can also optionally specify a time-to-live and a minimum number of matching ServiceDescriptions to be returned:

```
request.setTimeToLive(50);
request.setMinMatches(4);
```

If the time to live is set to a value less than or equal to zero, the YellowPagesTimetoLiveException is thrown.

Finally, set the request expression in the DiscoveryRequest and ask the Yellow Pages Directory to perform the lookup. This example uses the synchronous lookup, which returns matches in the requesting thread:

```
request.setRequestExpression(expr);
ServiceDescription[] matches = yellowPages.lookup(request);
```

Once a list of matches has been returned, you can resolve the service itself by calling `resolveService()`:

```
IMyService service = (IMyService) matches[0].resolveService();
```

The YellowPages1 Example demonstrates the Yellow Pages Directory.

Using a Discovery Listener

Each Yellow Pages instance in a Yellow Pages Directory responds individually to a discovery request. In some situations it is preferable to receive these responses asynchronously. The `IDiscoveryListener` interface provides a callback mechanism to receive responses to a discovery request. `IDiscoveryListener` has two methods:

```
void receiveServiceDescriptions(ServiceDescription[] descriptions);
```

The `receiveServiceDescriptions()` method is called when a Yellow Pages instance returns zero or more `ServiceDescriptions` in response to a discovery request. There is no guarantee on how many times this method is called or how many `ServiceDescriptions` will be passed to the listener.

```
void lookupComplete();
```

This method is called when the discovery request is considered complete due to an expiring time-to-live, receiving the minimum number of `ServiceDescriptions`, or receiving a response from all Yellow Pages instances in the Yellow Pages Directory.

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

To use a discovery listener, implement the `IDiscoveryListener` interface and provide the implementation to the asynchronous version of `lookup()`:

```
IDiscoveryListener myListener = new MyDiscoveryListener();
yellowPages.lookup(myDiscoveryRequest, myListener);
```

Because results are returned asynchronously, the call to `lookup()` returns immediately.

Using UDP as a messaging transport

Oneway, asynchronous, unreliable invocations can be made via UDP (unicast, multicast, and broadcast). To use this transport, specify the "udp" protocol in the URL for `ServerContext`'s `startServer(String url)` method. Also within the URL, a "well-known", unique integer ID must be supplied, and additionally for a client, the full class name for the interface or implementation class of the server object. For example:

```
// unicast (object ID is 99 for these examples, and
// client ID must match server ID)
ServerContext sc1 = voyagerContext.acquireServerContext("sc 9000");
sc1.startServer("udp://localhost:9000/99");
sc1.export(new ex.ServerObject(), "/99");
ClientContext cc1 = voyagerContext.acquireClientContet("sc 9000");
cc1.openEndpoint("udp://localhost:9000/99");

//broadcast
cc1.getNamespace().lookup("udp:/99;proxyClass=ex.ServerObject");

//multicast
ServerContext sc2 = voyagerContext.acquireServerContext("multi
9000");
sc2.startServer("udp://230.0.0.1:9000/99");
sc2.export(new ex.ServerObject(), "/99");
cc1.lookup("udp:/99;proxyClass=ex.ServerObject")
Proxy.export(new ex.ServerObject(), "udp://230.0.0.1:9000/99");
ClientContext cc2 = voyagerContext.acquireClientContet("multi 9000");
cc2.openEndpoint("udp://230.0.0.1:9000/99");
exServerObject proxy =
cc2.getNamespace().lookup("udp:/99;proxyClass=ex.ServerObject");
```

Using custom object streamers

Data marshaling for a remote invocation parameter can be controlled by using a custom object streamer. Custom object streamers implement the `com.recursionsw.ve.messageprotocol.vrmp.IStreamer` interface and are registered via `Vrmp.getMessageStreamerRegistry().registerStreamer(<class>, <streamer>)`

For a complete example of use, please see `examples.udp.MessageStreamerExample` in the java examples directory of the install.

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

Web Services: SOAP

Web Services Integration with Voyager

Note: The web services integration described here is not currently supported in Voyager on .NET.
To enable Web Services support, either use the "-web" option when running the `voyager` utility, or programmatically call com.recursionsw.ve.web.services.WebServices.enableWebServices(). This enables Voyager to act as both a Web Services server and client.

Server Support for Web Services

Voyager can expose an existing agent or POJO as a Web Service. Any object bound into the Voyager Naming Service can be accessed as a Web Service if Web Services support is enabled. The name used to bind the object is used as the URL when the Web Service is accessed. To obtain the WSDL for the services, append "?wsdl" to the URL.

For example, assuming a Voyager server has been started on //dallas.recursionsw.com:8000 and an object is bound into /services/Stockmarket in this server, you can access the Stockmarket as a Web Service at the URL http://dallas.recursionsw.com:8000/services/Stockmarket. Following is an example snippet of code using XFire to access this Web Service:

```
// Create a service model for the client
ServiceFactory serviceFactory = new ObjectServiceFactory();
Service serviceModel = serviceFactory.create(IStockmarket.class);

// Create a client proxy
XFireProxyFactory proxyFactory = new XFireProxyFactory();
String hostName = InetAddress.getLocalHost().getHostName();
IStockmarket market =
(IStockmarket)proxyFactory.create(serviceModel,
"http://dallas.recursionsw.com:8000/services/Stockmarket");

market.getQuote("sun");
```

Client Support for Web Services

Voyager supports invocation of remote Web Services via HTTP. To access a remote Web Service you must first generate a Java interface from WSDL using the webgen utility. Assuming the service is located at http://dallas.recursionsw.com:8000/services/Stockmarket, execute the following command:

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

```
java com.recursionsw.ve.tools.webgen.WebGen –serviceUri
http://dallas.recursionsw.com:8000/services/Stockmarket
```

This generates the requires client-side Java interfaces and classes necessary to access the service. These .java files will have a package based on the service's URI. In this example, the package will be "services". If the service URI does not have a filepath associated with it (for example, "http://dallas.recursionsw.com:8000/Stockmarket") a package of "webgen" will be used for generation.

Once the .java files are generated, you can look up and invoke the Web Service using Voyager's standard mechanism, the Namespace class:

```
ClientContext clientContext =
voyagerContext.acquireClientContext("Server8000");
services.Stockmarket market = clientContext.
getNamespace().lookup("ws:http://dallas.recursionsw.com:8000/services/S
tockmarket");
int price = market.quote("SUN");
```

Note the "ws:" prefix in the name. This prefix is required when looking up any Web Service using Voyager. Also, the URL provided is used to resolve the interface ("services.Stockmarket") the proxy will implement. If the location of the Web Service changes or you specified a different package for the generated classes when running WebGen, you must add a mapping to tell Voyager how to resolve the URL to the interface. For example, if you execute

```
java com.recursionsw.ve.tools.webgen.WebGen –serviceUri
http://dallas.recursionsw.com:8000/services/Stockmarket -targetPackage
com.recursionsw.generated
```

You will need to add the following mapping prior to calling Namespace's lookup():

```
com.recursionsw.ve.web.services.WebServices.addInterfaceMapping
("http://dallas.recursionsw.com:8000/services/Stockmarket",
"com.recursionsw.generated.Stockmarket");
```

Then, you can look up the Web Service:

```
ClientContext clientContext =
voyagerContext.acquireClientContext("Server8000");
com.recursionsw.generated.Stockmarket market = clientContext.
getNamespace().lookup("ws:http://dallas.recursionsw.com:8000/services/S
tockmarket");
int price = market.quote("SUN");
```

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

The WebGen utility has two additional parameters: use –targetDir to specify the directory where classes will be generated, and –overwrite to specify whether existing classes will be overwritten:

```
java com.recursionsw.ve.tools.webgen.WebGen –serviceUri
http://dallas.recursionsw.com:8000/services/Stockmarket -targetPackage
com.recursionsw.generated –targetDir src/generated –overwrite true
```

LDAP Authentication Service

Using Voyager to Authenticate Clients

LDAP, or Lightweight Directory Access Protocol, is a standard employed by many organizations to provide user security. User information is stored in the LDAP Directory Server, and programs authenticate users by matching provided client principals against the LDAP Directory Server. Voyager's LDAP authentication service allows Voyager clients to be automatically authenticated against an LDAP server at connect time.

Installing and Configuring the Voyager LDAP Authentication Service

To install the LDAP authentication service, set the property:

`com.recursionsw.ve.auth.enableLDAPAuthenticationService` (Voyager JSE) or
`recursionsw.voyager.auth.enableLDAPAuthenticationService` (Voyager .NET);

or, call:

Voyager JSE:

```
com.recursionsw.ve.auth.ldap.LDAPHandshakePropertiesHandler.
  getInstance().install()
```

or

Voyager .NET:

```
recursionsw.voyager.auth.ldap.LDAPHandshakePropertiesHandler.
  getInstance().install()
```

In addition to installing the LDAP authentication service, it will need to be configured with the LDAP Directory Server's network address and the LDAP domain to be used for authentication. Use the following properties or API's to configure the Voyager server's LDAP authentication service:

Voyager JSE:

`com.recursionsw.ve.auth.LDAPDomain`=*servercontext=domain*
`com.recursionsw.ve.auth.LDAPHost`=*servercontext=host*

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

Voyager .NET:
`recursionsw.voyager.auth.LDAPDomain`=*servercontext=domain*
`recursionsw.voyager.auth.LDAPHost`=*servercontext=host*

or, call the following methods on `LDAPHandshakePropertiesHandler`:

Voyager JSE:
```
LDAPHandshakePropertiesHandler.getInstance().setLDAPHost(String
  serverContextName, String host)
LDAPHandshakePropertiesHandler.getInstance().setLDAPDomain(String
  serverContextName, String domain)
```
or

Voyager .NET:
```
LDAPHandshakePropertiesHandler.getInstance().setLDAPHost(String
  serverContextName, String
host)LDAPHandshakePropertiesHandler.getInstance().setLDAPDomain(String
  serverContextName, String domain)
```

The LDAP domain property is required only if the LDAP Directory Service being used for authentication requires it to be present as part of the distinguished name when performing a bind to the server.

The server context name refers to the name of a `ServerContext` as obtained through the call to `VoyagerContext.acquireServerContext(String name)`. For example, if a Voyager JSE server calls `VoyagerContext.acquireServerContext("my_server")` it would set the properties:
```
com.recursionsw.ve.auth.LDAPDomain=my_server=some_ldap_domain
com.recursionsw.ve.auth.LDAPHost=my_server=ldap://ldapserverhost
```

or call:
```
LDAPHandshakePropertiesHandler.getInstance().setLDAPDomain("my_server",
"some_ldap_domain");
LDAPHandshakePropertiesHandler.getInstance().setLDAPHost("my_server",
"ldap://ldapserverhost");
```

When configured in this fashion, Voyager clients connecting to a server started in the ServerContext with the specified name **must** specify a client principal (e.g. username and password). Failure to provide a valid principal will cause connection attempts to be aborted by the server. When this occurs, the server will close the connection immediately without sending a response to the client.

If a ServerContext with a *different* name is started, connections established to servers started in that ServerContext *will not* be authenticated. It is legal to start multiple ServerContext's in a Voyager server application; take care that when doing so and using the LDAP authentication service that you specify which one or ones must use authentication. If multiple ServerContext's must be authenticated, separate configuration

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

properties using a semi-colon (";") or call the LDAP configuration API's separately for each ServerContext name to be configured. For example, if the ServerContext's "server1" and "server2" are both obtained and need to be authenticated, use:

```
com.recursionsw.ve.auth.LDAPHost=server1=ldap://ldaphost1;server2=ldap:
//ldaphost2
```

Note that installation of the LDAP authentication service should be done *before* any clients connect to the server. If using properties to install the authentication service, Voyager ensures installation occurs during the Voyager startup sequence. If using the Voyager API to programmatically install the authentication service, it is advisable to install the client after calling `Voyager.startup()` and before calling `ServerContext.startServer()`.

Installing and Configuring the Voyager LDAP Client

To configure a Voyager client to authenticate to a Voyager server using LDAP, enable the LDAP authentication client. Set the property:

`com.recursionsw.voyager.auth.useLDAPClientAuthentication` (Voyager JSE and CLDC) or
`recursionsw.voyager.auth.useLDAPAuthentication` (Voyager .NET and CF);

or, call:

Voyager JSE and CLDC:

```
com.recursionsw.voyager.auth.ldap.LDAPHandshakeContributor().
  getInstance().install()
```

or

Voyager .NET and CF:

```
recursionsw.voyager.auth.LDAPHandshakeContributor().
  getInstance().install()
```

The LDAP authentication client must be configured with a user principal. Use a format similar to the properties used to configure the LDAP authentication service:

Voyager JSE and CLDC:

`com.recursionsw.voyager.auth.LDAPClientIdentity`=*clientcontext*=*identity*
`com.recursionsw.voyager.auth.LDAPClientToken`=*clientcontext*=*token*

Voyager .NET and CF:

`recursionsw.voyager.auth.LDAPClientIdentity`=*clientcontext*=*identity*
`recursionsw.voyager.auth.LDAPClientToken`=*clientcontext*=*token*

or call:

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

Voyager JSE and CLDC:

```
com.recursionsw.ve.auth.ldap.LDAPHandshakeContributor.getInstance().
  setLDAPAuthenticationIdentity(String clientContextName, String
identity)
com.recursionsw.ve.auth.ldap.LDAPHandshakeContributor.getInstance().
  setLDAPAuthenticationToken(String clientContextName, String token)
```

Voyager .NET and CF:

```
recursionsw.voyager.auth.ldap.LDAPHandshakeContributor.getInstance().
  setLDAPAuthenticationIdentity(String clientContextName, String
identity)
recursionsw.voyager.auth.ldap.LDAPHandshakeContributor.getInstance().
  setLDAPAuthenticationToken(String clientContextName, String token)
```

The client context name is the name used when calling `VoyagerContext.acquireClientContext(String name)`. For example, if a Voyager JSE client needs to authenticate against a Voyager server with LDAP authentication enabled and calls `VoyagerContext.acquireClientContext("my_server")`, it would set the properties (assuming the client user is `joe.user` and password is `joe.password`):

```
com.recursionsw.voyager.auth.LDAPClientIdentity=my_server=joe.user
com.recursionsw.voyager.auth.LDAPClientToken=my_server=joe.password
```

or call:

```
com.recursionsw.ve.auth.ldap.LDAPHandshakeContributor.getInstance().
  setLDAPAuthenticationIdentity("my_server", "joe.user");
com.recursionsw.ve.auth.ldap.LDAPHandshakeContributor.getInstance().
  setLDAPAuthenticationToken("my_server", "joe.password");
```

If multiple ClientContext's need to use LDAP authentication, you can configure the authentication client properties using a semi-colon (";") to separate values or call the LDAPHandshakeContributor configuration methods for each ClientContext. For example, if two ClientContext's named "app1" and "app2" both need to use LDAP authentication, specify:

```
com.recursionsw.ve.auth.LDAPClientIdentity=app1=joe.user;app2=jane.user
```

Note that installation of the LDAP authentication client must be performed *before* any remote operations are performed. If using properties to install the authentication client, Voyager ensures installation occurs during the Voyager startup sequence. If using the Voyager API to programmatically install the authentication client, it is advisable to install the client after calling `Voyager.startup()` and before calling `ClientContext.openEndpoint()`.

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

Voyager LDAP Authentication Providers

The Voyager LDAP authentication service uses an LDAP authentication provider to perform the actual authentication of the client principal against the LDAP Directory Server. This provider is an implementation of the interface `com.recursionsw.ve.auth.ldap.ILDAPAuthenticationProvider` (Voyager JSE) or `recursionsw.voyager.auth.ldap.ILDAPAuthenticationProvider` (Voyager .NET). This interface declares a single method:

```
boolean authenticate(String ldapDomain, String ldapHost, String
identity, String token);
```

Voyager provides an implementation of the provider which does the following:

1. Creates an LDAP user dn (distinguished name) from the client-provided identity and server-configured ldapDomain;
2. Connects to the LDAP Directory Server at the server-configured ldapHost;
3. Performs an LDAP login ("bind") using the user dn and client-provided token (password).

For example, assume a Voyager JSE server application is configured with the following properties:

```
com.recursionsw.ve.auth.enableLDAPAuthenticationService
com.recursionsw.ve.auth.LDAPDomain=my_server=ldap_domain
com.recursionsw.ve.auth.LDAPHost=my_server=ldap://ldapserver
```

Assume a Voyager client attempts to connect with the following properties:

```
com.recursionsw.ve.auth.useLDAPClientAuthentication
com.recursionsw.ve.auth.LDAPClientIdentity=server=joe.user
com.recursionsw.ve.auth.LDAPClientSecurityToken=server=joe.password
```

When the Voyager JSE server's LDAP authentication service receives the client identity and token from the Voyager client, it will call the configured LDAP authentication provider with

```
authenticate("ldap_domain", "ldap://ldapserver", "joe.user",
"joe.password")
```

If the call to authenticate returns "true", the Voyager LDAP authentication service allows the connect attempt to succeed. If it returns "false", it closes the connection attempt.

Custom Voyager LDAP Authentication Providers

A custom authentication provider may be implemented and the Voyager LDAP authentication service can be configured to use the custom provider. To implement and

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

use a custom provider, create a class that implements the `ILDAPAuthenticationProvider` interface and then configure the authentication service to use the custom provider, either via a property or API call:

Voyager JSE:

`com.recursionsw.ve.auth.LDAPAuthenticationProvider`=*fullyqualifiedclassname*

or

```
LDAPHandshakePropertiesHandler.getInstance().
  setLDAPAuthenticationProvider(ILDAPAuthenticationProvider provider);
```

Voyager .NET:

`recursionsw.voyager.auth.LDAPAuthenticationProvider`=*fullyqualifiedtypename*

or

```
LDAPHandshakePropertiesHandler.getInstance().
  setLDAPAuthenticationProvider(ILDAPAuthenticationProvider provider);
```

The custom provider will be called with the configured LDAP domain and host, and client-provided identity and security token. It must return either “true” to indicate authentication succeeded or “false” to indicate authentication failed.

Note that the LDAP authentication service will call the authentication provider only once for each client program instance that connects. Once the client has been successfully authenticated, further connections established by the client will not cause authentication to occur. If the client program is stopped and restarted, however, it is treated as a new program instance and the first connection from that new program instance will be authenticated.

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

Voyager Administration

Configuration and Management

Several of Voyager's internal settings can be modified at runtime using static methods. For example, you can change the maximum thread pool size at runtime by using `TaskManagerConfigurator.setMaxUserThreads()`.

The same settings can also be set using standard Java property files, which are read at startup. This approach allows values to be set and changed without modification of the application source code.

In this chapter, you will learn to:

- Understand Voyager properties
- Specify a properties file
- Specify multiple values

Understanding Voyager Properties

The following table summarizes Voyager's user-customizable properties. Each property is case sensitive.

Property	Value
`Voyager.ClassManager.enableResourceServer`	`true | false`
`Voyager .taskmanagement.TaskManagerConfigurator.setMaxUserThreads`	`<int>`
`Voyager.loader.VoyagerClassLoader.addResourceLoader`	`#<classname>`
`Voyager.loader.VoyagerClassLoader.addURLResource`	`<URL>`
`Voyager.loader.VoyagerClassLoader.setResourceLoadingEnabled`	`true | false`
`Voyager.router.Routing.setRouterAddress`	`<XURL>`
`Voyager.tcp.TcpTransport.setServerListenBacklog`	`<int>`
`Voyager.transport.Transport.register`	`#<classname>`
`Voyager.transport.Transport.setDefaultTransport`	`<transport id>`
`Voyager.tcp.use_ip_addressing`	`true | false`
`com.recursionsw.ve.tcp.latency`	`<int>`

- `Voyager.ClassManager.enableResourceServer`

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

This property allows a Voyager server to be able to serve resources, typically classes, via HTTP. It is equivalent to the `ClassManager.enableResourceServer()` method. The default value is `false`. For example:

```
Voyager.ClassManager.enableResourceServer=true
```

- `Voyager.taskmanagement.TaskManagerConfigurator.setMaxUserThreads`

This property allows users to specify the maximum number of threads Voyager will create. Higher numbers are useful for servers with more memory, while lower numbers are often useful when memory resources are limited or must be shared with other applications. This property is equivalent to the `TaskManagerConfigurator.setMaxUserThreads()` method. The default value is `Integer.MAX_VALUE`. For example:

```
Voyager.taskmanagement.TaskManagerConfigurator.setMaxUserTh
reads=50
```

- `Voyager.loader.VoyagerClassLoader.addResourceLoader`

This property allows resource loaders to be added to Voyager's classloading mechanism. A custom resource loader allows Voyager to retrieve resources, including Java classes, from custom sources. The property is equivalent to the `VoyagerClassLoader.addResourceLoader()` method. The classname provided must resolve to a public class that implements `IResourceLoader` and has a public no-argument constructor. For example:

```
Voyager.loader.VoyagerClassLoader.addResourceLoader=#com.ac
me.AcmeLoader
```

- `Voyager.loader.VoyagerClassLoader.addURLResource`

This property allows the Voyager server to load classes from the specified URL. If the URL is an HTTP URL, it must map to an HTTP server, such as a remote Voyager that has resource serving enabled. This property is equivalent to the `VoyagerClassLoader.addURLResource ()` method. For example:

```
Voyager.loader.VoyagerClassLoader.addURLResource=http://acm
e.com:8000
```

- `Voyager.loader.VoyagerClassLoader.setResourceLoadingEnabled`

This property enables or disables resource loading. It is primarily used to disable resource loading, which includes dynamic proxy generation, loading from a remote URL, etc. It is equivalent to the `VoyagerClassLoader.setResourceLoadingEnabled()` method and has a default value of `true`. All proxies, including those for select Voyager classes, must be generated manually with the `pgen` utility. For example:

```
Voyager.loader.VoyagerClassLoader.setResourceLoadingEnabled
=false
```

- `Voyager.router.Routing.setRouterAddress`

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

This property allows a routing address to be specified. If an application specifies a router, all remote messages are automatically sent to the router for redirection to their final destination. It is equivalent to the `Routing.setRouterAddress()` method. For example:

```
Voyager.router.Routing.setRouterAddress=//10.2.15.194:8000
```

- `Voyager.tcp.TcpTransport.setServerListenBacklog`

This property can be used to set the listen backlog for all TCP server sockets used by Voyager. It is equivalent to the `TcpTransport.setServerListenBacklog()` method. The default value is 50. For example:

```
Voyager.tcp.TcpTransport.setServerListenBacklog=20
```

- `Voyager.transport.Transport.register`

This property allows a custom transport mechanism to be registered with Voyager's pluggable transport system. It is equivalent to the `Transport.register()` method. By default, a TCP transport mechanism is registered with Voyager. The classname provided must resolve to a public class that implements `ITransport` and has a public no-argument constructor. For example:

```
Voyager.transport.Transport.register=#com.acme.SSLTransport
```

- `Voyager.transport.Transport.setDefaultTransport`

This property allows users to set the default transport used by Voyager. It is equivalent to the `Transport.setDefaultTransport()` mechanism. By default, all remote messaging is handled by the TCP transport. If a custom transport is set to the default, an instance of the transport must be registered with `Transport`. For example:

```
Voyager.transport.Transport.setDefaultTransport=ssl
```

- `lib.util.Console.setEnabledTopics`

This property allows the Console log level to be set. It is equivalent to the `Console.setEnabledTopics()` method which, unlike `Console.enableTopic()`, removes all enabled topics before enabling the requested topic. Available options are `silent`, `exceptions` and `verbose`. For example:

```
lib.util.Console.setEnabledTopics=exceptions
```

`Voyager.tcp.use_ip_addressing`

When this property is set to `true`, Voyager will use only IP addresses when sending a proxy to a remote process. This is required when the remote process might not be able to resolve the local process hostname. For example:

```
Voyager.tcp.use_ip_addressing=true
```

`com.recursionsw.ve.tcp.latency`

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

This value determines the maximum latency when Voyager attempts to create a connection to a remote process. If the connection cannot be established within this time, an exception is thrown. For example:

```
com.recursionsw.ve.tcp.latency=10000
```

Specifying a Properties File

Voyager servers are started with the `voyager` utility for running in a Java VM and `voyager_net` utility for running in a .NET VM. The `voyager` utility has an extra flag, the `-p` flag, for specifying a properties file. For example, the following code will start a Voyager server on port `8000` with the properties file `acme.properties`. Once all flags have been processed, Voyager will read in the properties file and configure itself as specified in the properties file. Voyager will then complete the startup process.

```
voyager //:8000 -p acme.properties
```

The Configuration1 Example demonstrates custom configuration of a Voyager server.

For custom applications in which the `voyager` utility is not used to start up a server, a Voyager properties file can be loaded using a `PropertyLoader`.

The Configuration2 Example demonstrates custom configuration of a user application.

Specifying Multiple Values

Some Voyager properties can be specified more than once. For instance, an application may need to be able to load classes from several URLs. Therefore, Voyager allows the `addURLResource` property to be specified more than once using an indexing syntax. For example, the following code will install URL resource loaders for two different remote HTTP servers.

```
Voyager.loader.VoyagerClassLoader.addURLResource[1]=http://
acme.com:8000
Voyager.loader.VoyagerClassLoader.addURLResource[2]=http://
acme2.com:8000
```

The Configuration3 Example demonstrates specifying multiple Voyager properties.

Connection Management

Voyager provides the ability to manage the connections underlying Voyager-to-Voyager communications. The `com.recursionsw.ve.transport.IConnectionManagementPolicy` interface is implemented to create a connection management policy. An instance of the policy is created and registered with `Transport.registerConnectionManagementPolicy( String protocol, IConnectionManagementPolicy policy )`. Once registered, the policy is queried for the following actions:

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

- Creation of client connection to remote URL
- Creation of server connection to remote URL
- Idle of client connection

There is a cost associated with managing connections. Every time a new connection is desired, Voyager must examine the current policy to determine if the new connection is allowed. Furthermore, the more complicated the policy restrictions are, the longer it will take to analyze. Although in most cases this will not be noticeable, high-volume Voyager networks may wish to carefully tune connection management parameters or even revert to non-managed behavior.

A client connection in Voyager is used when one ORB is communicating a function invocation to a remote object. A server connection is involved whenever an exported Proxy receives a remote invocation request for a local object.

In this section, you will learn to:

- Understand connection management policies.
- Understand case policies.
- Establish case policies.
- Define policy listeners.

Understanding Connection Management Policies

Voyager connections are segregated into client connections and server connections. A client connection, associated with a Proxy to a remote object, sends invocation requests. A server connection, associated with a local object exported to remote Voyager servers, receives invocation requests.

Voyager provides two default connection management policies: `com.recursionsw.ve.transport.impl.tcp.BasicConnectionManagementPolicy` and `com.recursionsw.ve.transport.impl.tcp.RangeConnectionManagementPolicy`. `BasicConnectionManagementPolicy` applies a single `CasePolicy` to limit connections. `RangeConnectionManagementPolicy` provides the capability to associate a `CasePolicy` with a `HostAddressRange`, a range of addresses and ports. When the policy is queried, the applicable `CasePolicy`'s are used to determine whether the operation will be allowed. An instance of `ManagedTcpPolicy` contains a collection of `CasePolicy` objects describing the restrictions on connections between different Voyager servers according to their IP addresses and ports. If a new connection would violate the set limits, then the requesting thread will block until the new connection is allowed. Note that this could

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

cause deadlock problems if distributed objects recursively call methods upon one another such that they use up all allowed connections.

Understanding Case Policies

A `CasePolicy` consists of several characteristics describing how connections should be limited or disconnected.

Maximum Number of Server Connections

Server connections may be capped at a particular number. Server connections include currently active connections accepting invocation requests as well as pending connections awaiting a client to connect.

Maximum Number of Client Connections

Client connections may also be limited. Client connections deliver invocation requests to remote objects.

Maximum Number of Idle Client Connections

A client connection is idle if it is not sending an invocation request or awaiting a response from an invocation request. Idle client connections are pooled, allowing a small number of connections to handle many proxies, as long as invocations are relatively infrequent.

A server connection is idle if it is awaiting an invocation request.

Client Connection Idle Time

A client connection can be given an idle time limit. If a client connection idles longer than this limit, it will be removed from use and closed.

Establishing Case Policies for RangeConnectionManagementPolicy

Case policies must be added to a `ManagedTcpPolicy` object, which is then set as the policy for a given ORB. The easiest way is often to obtain the current policy, modify it, and then establish the modified policy as the new ruling policy.

`CasePolicy` objects are managed by three methods on the `ManagedTcpPolicy` class.

- `public void setCasePolicy( HostAddressRange range, CasePolicy casePolicy );`

Sets the ruling `CasePolicy` for the given range of addresses. All connections that fall within the given range will be subject to the restrictions of the new `CasePolicy`.

- `public CasePolicy getCasePolicy( HostAddresRange range );`

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

Retrieves the `CasePolicy` established for the given range of addresses. If no `CasePolicy` is explicitly established, then the least-restrictive `CasePolicy` is returned, no connection or idle time limits.

- `public void removeCasePolicy( HostAddresRange range );`

Removes any established `CasePolicy` for the given range of addresses.

Two functions also provide access to the global case policy ruling any and all connections for the current Voyager server:

```
public void setGlobalCasePolicy( CasePolicy casePolicy );

public CasePolicy getGlobalCasePolicy();
```

About HostAddressRange

A `HostAddressRange` represents a set of connection endpoints. The `HostAddressRange` constructor takes a String value describing the host and port ranges for the set. Host ranges may include asterisks as wildcards to indicate all matching values. Port ranges may use a dash to indicate an inclusive range. For example:

`10.1.0.1:2000`	Specifies the endpoint at port `2000` on the machine with the IP address `10.1.0.1`
`www.recursionsw.com:5000`	Specifies the endpoint at port `5000` on the machine with the IP address `www.recursionsw.com`
`host.org:-`	Specifies all endpoints at any port on the machine with the IP address `host.org`
`*.recursionsw.com:1024-5000`	Specifies all endpoints with a port number between `1024` and `5000` (inclusive) on any machine whose IP address ends with `recursionsw.com`
`10.*.-`	Specifies all endpoints located at any port on any machine whose IP address begins with `10`
`*.-`	All endpoints

Note that a machine always has an IP address in number format and usually has one in name format. If you use HostAddressRanges with the name formats, then you may experience delays when Voyager queries your system's Domain Name Service (DNS) to resolve machine names. If this delay is too large, either use a faster DNS server or use the `#####number####` format for all HostAddressRange entries.

Secure Sockets

Voyager supports use of encrypted connections by using Transport Layer Security (TLS; also referred to as Secure Sockets Layer or SSL). For .Net version of Voyager, SSL is achieved by wrapping around the socket using SslStream. Policies are available that apply by name to client and server contexts to specify cleartext or encrypted connections

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

and, for encrypted connections, the connection setup parameters. To successfully connect the contexts on either end of a connection must apply compatilble policies, e.g., both must apply the default policy that establishes an unencrypted connection, or both must apply a policy that establishes an encrypted connection with the appropriate parameters.

The context policies are typically decleared in the Voyager properties file, or the property values are set before Voyager is started. Once started, Voyager will not change the context policy configuration.

The **Voyager Security Developer's Guide** documents how to configure client and server context policies.

Examples

Setting the Global CasePolicy

To set a Voyager server to limit the number of client connections to 25 and the idle time limit to 10 seconds.

```
IConnectionManagementPolicy policy = new
BasicConnectionManagementPolicy(25, CasePolicy.NO_LIMIT,
CasePolicy.NO_LIMIT, 10000 );
Transport.registerConnectionManagementPolicy( "tcp", policy
);
```

Setting Case Policies

To limit the number of client connections to the `recursionsw.com` space to 10 with 5-second idle limits.

```
int NO_LIMIT = CasePolicy.NO_LIMIT;
RangeTcpPolicy rangePolicy = new RangeTcpPolicy();
IConnectionManagementPolicy managementPolicy = new
RangeConnectionManagementPolicy( rangePolicy );
rangePolicy.setCasePolicy( new
HostAddressRange( "*.recursionsw.com" ), new
CasePolicy( 10, NO_LIMIT, NO_LIMIT, 5000 ) );
Transport.registerConnectionManagementPolicy( "tcp",
managementPolicy );
```

To prevent idle connections to the `10.2.10.*` subnet.

```
rangePolicy.setCasePolicy( new
HostAddressRange( "10.2.10.*" ), new CasePolicy( 0, 0, 0, 0
) );
```

To limit the number of server connections that will be accepted on port `8000` to 7.

```
rangePolicy.setCasePolicy( new
HostAddressRange( "//:8000" ), new CasePolicy( NO_LIMIT, 7,
NO_LIMIT, 0 ) );
```

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

Sockets are generated by classes that implement the `com.recursionsw.ve.transport.impl.tcp.socket.SocketFactory` interface. That interface has a single method with the following signature.

```
public Socket createClientSocket( SocketPolicy policy,
String host, int port, InetAddress bindHost, int bindPort )
throws IOException;
```

A `SocketPolicy` provides the necessary configuration parameters for the `SocketFactory`. Its definition provides only the minimum required for TCP sockets, but can easily be extended for custom implementations.

Java

```
public abstract class SocketPolicy implements Serializable
  {
  private long timeout = -1;

  public abstract String getShortName();

  public abstract String getSocketFactoryClassName();

  public abstract String getSocketPolicyConfigurator();

  public final long getTimeout()
    {
    return timeout;
    }

   public final void setTimeout( long timeout )
    {
    this.timeout = timeout;
    }
  }
```

- `getShortName()` returns a String that will be displayed in the Socket Policies panel's Socket Type list.
- `getSocketFactoryClassName()` returns the name of the `SocketFactory` class used to create sockets governed by this policy type.
- `getSocketPolicyConfigurator()` returns the name of a class providing the socket policy's GUI configuration and implementing the `SocketPolicyConfigurator` interface.

ServerSocket Policies

`ServerSocket` Policies also use the `SocketPolicy` class. In this case, the class names that the `SocketPolicy` provides will refer to `ServerSocket` factories and configurations. The one method in the `ServerSocketFactory` interface constructs `ServerSocket` instances to be used by Voyager when accepting requests on certain ports:

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

```
public ServerSocket createServerSocket( SocketPolicy
policy, InetAddress bindInterface, int bindPort, int
backlog ) throws IOException;
```

Note: Socket timeouts are not currently used by Voyager.

Adding Custom Sockets to Voyager

Voyager provides the `com.recursionsw.ve.transport.impl.tcp.TCPSocketPolicy` and the `com.recursionsw.ve.transport.impl.tcp.TCPServerSocketPolicy` classes. These classes implement basic TCP Sockets behavior.

Configuring Socket Policies Programmatically

Socket policies are managed by a singleton of the `com.recursionsw.ve.transport.impl.tcp.socket.SocketPolicyManager` class. This class manages the association between a `HostAddressRange` and a `SocketPolicy`. To add or remove a policy, you must first obtain the singleton instance of `SocketPolicyManager`, and then call the appropriate method.

VRMP Configuration

Voyager's native VRMP protocol can be configured through the following properties:

1. `com.recursionsw.ve.messageprotocol.vrmp.enableObjectStreamCache = {true|false} {default:true}`

Enabling this property increases the performance of the VRMP layer, at the cost of increased memory usage. The percent improvement depends on the size of your method invocations and network bandwidth. Smaller invocations will see a greater improvement, and high-bandwidth networks will see a greater improvement. NOTE: If you enable this property, it must be enabled on all Voyager VM, which will be communicating with each other.

2. `com.recursionsw.ve.messageprotocol.vrmp.streamCacheLimit = 0+ (default: 100)`

This property is only effective if enableObjectStreamCache=true. This controls the size of the cache used for streams. Increasing this value increases the amount of memory potentially used by VRMP, but will allow Voyager to have a greater number of streams cached to handle simultaneous connections.

3. `com.recursionsw.ve.messageprotocol.vrmp.enableVrmpConnectionCache = {true|false} default=false`

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

Voyager's TCP transport maintains a connection pool for outbound traffic. Turning on the `enableVrmpConnectionCache` flag enables a connection pool specifically for VRMP connectivity. This pool has improved performance. However, connections in this pool are not subject to the connection management policy currently enabled; specifically, the maximum number of client connections; maximum number of idle client connections; and maximum idle time.

4. `com.recursionsw.ve.messageprotocol.vrmp.enableVrmpLookahead = {true|false} default=false`

By default, the request handler for VRMP returns connections to the request manager after each invocation has been handled. Enabling this flag allows the VRMP request handler to "look ahead" at the input stream to determine if another VRMP invocation is available to be processed, and if so, processes that invocation in a more efficient fashion.

5. `com.recursionsw.ve.messageprotocol.vrmp.enableObjectDescriptorCache = {true|false} default=false`

When writing objects using Java's standard serialization process, ObjectOutputStream writes metadata for each type of object serialized. For some cases, the overhead of writing this descriptor can be significant. Enabling the VRMP object descriptor cache allows the VRMP protocol layer to cache object descriptors, minimizing the amount of information written to the output stream. This flag must be set to the same value for all Voyager VM's communicating together. When enabled, an incompatibility between versions of a serializable class will result in Voyager throwing an IOException with the message "Failure reading VRMP protocol".

RequestManager Properties

Voyager's RequestManager is responsible for processing server requests. By default, each server connection is assigned a Processor, which waits for an incoming request and then passes off the request to a request handler. The Processor then reschedules itself with the Voyager task management system.

The default behavior maximizes the robustness and reliability of the server. Two properties exist which offer improved server performance at the cost of less overall robustness.

1. `requestManager.rescheduleProcessor` = {true|false} (default: true)

By default, the Processor processing an connection is rescheduled after processing an incoming request. The thread allocated to it is returned to the Voyager thread pool and becomes available for other Voyager (or user) tasks in the Voyager task management system's queue.

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

By disabling the rescheduling of the Processor, a thread is associated with a connection until the connection is closed. This improves performance, but if the size of the thread pool is constrained, it is possible for connections to go unserviced for an unacceptable length of time. In general, if you set this property to false you should not constrain the size of Voyager's thread pool. If you do constrain it, use a connection management policy that limits the number of server side connections to a lower limit than the maximum thread pool size.

2. `requestManager.useFastInputStream` = {true|false} (default: false)

By default, the Processor processing a connection reads from a monitored input stream, which wraps the actual input stream. This monitored input stream will signal the Processor at any time if a read times out (i.e., if the connection management policy server idle time is exceeded).

Setting useFastInputStream causes the Processor to pass the wrapped stream to the request handler responsible for processing an incoming request once an incoming request is detected. This will result in faster performance, but if there is a network/socket failure while reading the incoming request, there may be an unacceptably long delay before the operating system notifies the Voyager process of the dead connection. In general, this property should not be set in an environment where network connections are unreliable or where client connections are likely to be improperly closed.

Configuring Voyager for Internet Protocol Version 6

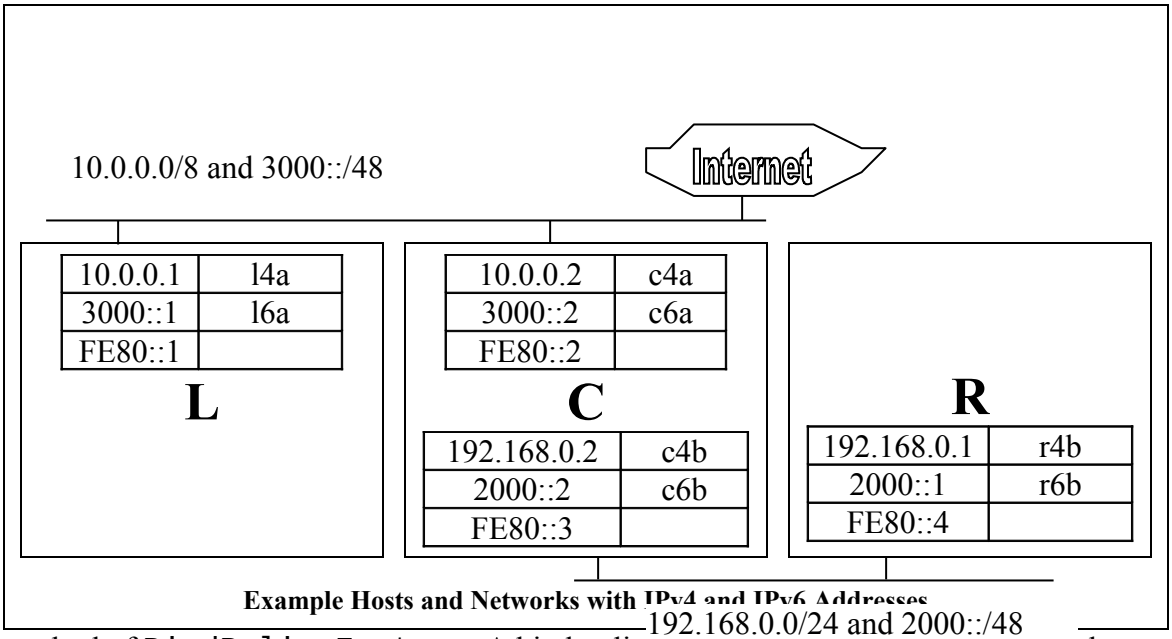
Introduction

For many years networked computers have used addresses referred to as IP (Internet Protocol) addresses. More specifically, the address format of 32 bit addresses, usually expressed as four 8-bit numbers, is Internet Protocol Version 4. Not too many years ago the organization managing Internet address assignments foresaw the possibility of running out of addresses. In addition, stresses caused by the growth of the Internet suggested a number of new features to handle both the growth and new applications, such as voice over IP (VOIP) and real time video and TV. A new addressing scheme, referred to as Internet Protocol Version 6 (IPv6), was created and is now being deployed.

Voyager implements a “bind policy” for server connections and a separate “bind policy” for client connections. A bind policy describes how to configure addresses for hosts with multiple addresses. The bind policy for server connections extends and generalizes the Voyager multihome capability.

A bind policy is described by the `IBindPolicy` interface. A policy is built using static methods found in the `BindPolicyFactory` class. A collection of policies is managed by a realization of `IBindPolicyManager`, which is also constructed by a static

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved



Example Hosts and Networks with IPv4 and IPv6 Addresses

method of `BindPolicyFactory`. A bind polic d policy manager to either `BindPolicyFactory.implementClientBindPolicies` or `BindPolicyFactory.implementMultiHomeBindPolicies`. An alternate mechanism is to specify the bind policies in the Voyager property file.

Example Network

The paragraphs below will use the hosts and networks found in the above diagram to illusrate how to define client and multihome policies. The single interface on each of hosts L and R is configurated with a single IPv4 address, a global IPv6 address, and a link local IPv6 address. The link local IPv6 address is autoconfigured while the global IPv6 address may be autoconfigured or it may come from a DHCP server or, in some operating systems, it can be manually configured. The IPv4 addresses are either manually configured or retrieved from a DHCP server. For purposes of this discussion, assume that addresses allocated out of the 10.0.0.0/8 network are routed to the Internet, although in the real world 10.0.0.0/8 network addresses are private.

The link local IPv6 addresses are not routed, and appear only on the network segement to which the interface is connected. That is, a socket connection from L's FE80::1 address to C's FE80::2 address will succeed, but attempts to connect from L's FE80::1 address to either C's FE80::3 or R's FE80::4 will fail.

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

In the example network host C is multihomed, i.e., C is configured with two network interfaces, each connecting to networks with differing address ranges.

There is no routing of addresses between the two networks connected to C.

Multihome Bind Policies

Specifying a multihome address (see the Multihome Support section) causes requests for server sockets, e.g., the sockets created when Voyager is started with an explicit URL, to be opened on the same port number for all the IP addresses to which the URLs resolve. The multihome bind policy extends the multihome capability with the ability to open a different set of IP addresses for different request URLs.
For example, Voyager running on host C probably wants to open at startup server sockets for addresses 127.0.0.1, ::1 (the IPv6 equilavent of 127.0.0.1), 10.0.0.2, 192.168.0.2, 3000::2, and 2000::2, all on the same port number. Some agents, however, may wish to limit themselves to connections from one or the other interfaces, e.g., to serve only clients from the Internet. Such a limitation is realized by specifying a multihome bind policy that results in opening sockets only for 10.0.0.2 and 3000::2.

Client Bind Policies

When a client needs a connection to a server, frequently to create proxy to a remote object, the clent needs to create a socket, which will need both an IP address and a port number, through which the server can respond. The normal connection using IPv4 simply asks that the client's socket be built using the local address and a port number assigned by the operating system. When connecting over IPv6, however, the client's socket needs to be built using not the default local IPv4 address but an IPv6 address. The client bind policy implements a mechanism for selecting the appropriate address for the client's end of the connection.

Bind Policy Specification

To illustrate two approaches to specifying bind policies, the following two sections will show how to define a bind policy for host L that returns both IPv4 and IPv6 loopback addresses when an IPv4 address is requested, and a second policy that returns both IPv4 and IPv6 global addresses when the IPv4 address is requested.

Network Address Scope

An address range with a network mask size of zero, such as seen below for both `policy3` and `policy4`, matches any address of the same type. The `policy3` network specification matches any IPv4 address. However, the `policy4` network specification matches only IPv6 addresses with global scope. Similarly an IPv6 network of FE80:: with netmask size of 0 matches only IPv6 addresses with local scope, and an IPv6 network of ::1 with a netmask size of 0 matches only IPv6 loopback addresses.

Programmatic Specification

The following Java code will build the two policies.

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

```
ArrayList result = new ArrayList();
ArrayList policies = new ArrayList();

InetAddress addr1 = InetAddress.getByName("127.0.0.1");
result.add(addr1);
IBindPolicy policy1 = BindPolicyFactory.constructBindPolicy(addr1, 8,
  (List) result.clone());
InetAddress addr2 = InetAddress.getByName("::1");
result.clear();
result.add(addr2);
IBindPolicy policy2 = BindPolicyFactory.constructBindPolicy(addr2,
  128, (List) result.clone());
policies.add(policy1);
policies.add(policy2);
IBindPolicyManager mgr1 =
  BindPolicyFactory.constructBindPolicyManager((List)policies.clone()
  );

result.clear();
policies.clear();
// either of the following two lines works
//InetAddress addr3 = InetAddress.getByName(c4a);
InetAddress addr3 = InetAddress.getByName("10.0.0.1");
result.add(addr3);
IBindPolicy policy3 = BindPolicyFactory.constructBindPolicy(addr3, 0,
  (List) result.clone());
result.clear();
// either of the following two lines works
//InetAddress addr3 = InetAddress.getByName(c6a);
InetAddress addr4 = InetAddress.getByName("3000::1");
result.add(addr4);
IBindPolicy policy4 = BindPolicyFactory.constructBindPolicy(addr4, 0,
  (List) result.clone());
policies.add(policy3);
policies.add(policy4);
IBindPolicyManager mgr2 =
  BindPolicyFactory.constructBindPolicyManager((List)policies.clone()
  );
```

Text Format Specification

The `BindPolicyFactory` static methods `implementClientBindPolicies` and `implementMultiHomeBindPolicies` accept a text representation of a collection of bind policies, construct an instance of `IBindPolicyManager`, and install the new manager as a client or multihome policy. These two methods can be referenced in a Voyager property file, thus offering a mechanism for configuring a host's policies without writing code.

Each collection of policies is expressed as a sequence of individual policies separated by semicolons. Elements within a policy are separated by spaces.

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

The first of two textual policy formats consists of a single policy element that describes both the range of addresses to which the policy applies and the single address returned when a candidate address falls within the policy's address range. The policy expressed as `127.0.0.1/8` produces an instance of `IBindPolicy` that matches any address in the range `127.0.0.0` to `127.255.255.255` and returns `127.0.0.1`. The policy expressed as `127.0.0.1/0` produces an instance of `IBindPolicy` that matches any IPv4 address and returns `127.0.0.1`.

The second of two textual policy formats consists of a network or address range specification followed by one or more result addresses. The result addresses can be any mix of IPv4 and IPv6 addresses, or domain names that resolve to an IP address of any type except the address `::`, or all zeroes, which is the IPv6 "any" address. The policy expressed as `127.0.0.1/8 127.0.0.1` produces an instance of `IBindPolicy` that matches any address in the range `127.0.0.0` to `127.255.255.255` and returns `127.0.0.1`, exactly the same policy as `127.0.0.1/8`. However, the policy expressed as `127.0.0.1/8 127.0.0.1 ::1` produces an instance of `IBindPolicy` that matches any address in the range `127.0.0.0` to `127.255.255.255` and returns both `127.0.0.1` and `::1`, i.e., an IPv4 loopback address returns both the IPv4 and IPv6 loopback addresses.

The textual representation of the policies built with method calls in the previous section is `::1/128; 127.0.0.1/8;` and `10.0.0.1/0; 3000::1/0`, respectively. Installing these as a client bind policy looks like this in Java code.

```
BindPolicyFactory.implementClientBindPolicies( "::1/128; 127.0.0.1/8;
  10.0.0.1/0; 3000::1/0");
```

When used in a Voyager property file, the above line of Java looks like either of the following lines.

```
Voyager.tcp.BindPolicyFactory.implementClientBindPolicies=:
:1/128; 127.0.0.1/8; 10.0.0.1/0; 3000::1/0

com.recursionsw.ve.transport.impl.tcp.BindPolicyFactory.imp
lementClientBindPolicies=::1/128; 127.0.0.1/8; 10.0.0.1/0;
3000::1/0
```

Example Bind Policies

The following sections describe configuration scenarios for each of the hosts found in the above diagram of hosts and networks.

Host L

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

If Voyager running on host L requires only IPv4 addresses, no explicit configuration is required. If, however, Voyager running on host L utilizes IPv6 addresses, bind policies must be written for both a pure IPv6 case and a mix of IPv4 and IPv6.

Configuration for IPv6 only

Requirements and the policy satisfying each requirement for this configuration are as follows.

Index	Requirement	Bind Policy
1	Requests to open a server port on l6a open the l6a address and fe80::1	`l6a/48 l6a fe80::1`
2	Requests to open a server port on fe80::1 open the port on only that address.	`fe80::1/0`
3	Requests to open a server port on the IPv6 loopback address open only the IPv6 loopback address.	`::1/128`
4	The local address for client connections to any remote IPv6 address with global scope is l6a.	`l6a/0`
5	The local address for client connections to any remote IPv6 address with local scope is fe80::1.	`fe80::1/0`

The policy satisfying the first requirement allows connections to the service from any globally scoped IPv6 address, as well as from any locally scoped address. Connections to fe80::1 should happen only from other hosts on the same LAN segment, since locally scoped IPv6 addresses should never be routed.
As with the first policy, connections to the second policy's fe80::1 result address should happen only from other hosts on the same LAN segment. This policy might be used for connecting the audit subsystem on host L to the audit subsystem on host C, since using this address prevents exposing the audit service to addresses not on the local network.

Configuration for mixed IPv4 and IPv6

Requirements and the policy satisfying each requirement for this configuration are as follows.

Index	Requirement	Bind Policy
1	Requests to open a server port on l6a open the l4a, l6a, and fe80::1 addresses.	`l6a/48 l4a l6a fe80::1`
2	Requests to open a server port on l4a open the l4a, l6a, and fe80::1 addresses.	`l4a/8 l4a l6a fe80::1`
3	Requests to open a server port on 127.0.0.1 open the port on both loopback addresses.	`127.0.0.0/8 127.0.0.1 ::1`
4	Requests to open a server port on ::1 open the port on both loopback addresses.	`::1/128 127.0.0.1 ::1`
5	Requests to open a server port on fe80::1 open the port on only that address.	`fe80::1/128`
6	The local address for client connections to any remote IPv4 address is l4a.	`l4a/0`

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

Index	Requirement	Bind Policy
7	The local address for client connections to any remote IPv6 address with global scope is l6a.	`l6a/0`
8	The local address for client connections to any remote IPv6 address with local scope is fe80::1.	`fe80::1/0`

The policy satisfying the first requirement allows connections to the service from any globally scoped IPv6 address, as well as from any IPv4 address and any locally scoped address. Connections to fe80::1 should happen only from other hosts on the same LAN segment, since locally scoped IPv6 addresses should never be routed.
The policy satisfying the second requirement allows connections to the service from any globally scoped IPv6 address, as well as from any IPv4 address and any locally scoped address.
The policy satisfying the fifth requirement might be used for connecting the audit subsystem on host L to the audit subsystem on host C, since using this address prevents exposing the audit service to addresses not on the local network.

Voyager should be started with a startup URL containing either l4a or l6a so that the server port is opened for all the host's addresses.

The client and multihome lines for the Voyager property file look like the following.

```
Voyager.tcp.BindPolicyFactory.implementClientBindPolicies=
l4a/0; l6a/0; fe80::1/0

com.recursionsw.ve.transport.impl.tcp.BindPolicyFactory.imp
lementMultiHomeBindPolicies= l6a/48 l4a l6a fe80::1; l4a/8
l4a l6a fe80::1; 127.0.0.0/8 127.0.0.1 ::1; ::1/128
127.0.0.1 ::1; fe80::1/0
```

Host C

Host C is multihomed, i.e., it is configured with more than one IPv4 address. While such a configuration most frequently occurs when a host contains more than one network interface, e.g., two wired interfaces or a wired and wireless interface, Voyager defines multihomed as any host with more than one IPv4 address or any host with IPv6 on any interface. Because host c is multihomed, explicit configuration is always required.

Configuration for IPv4 only

Requirements are as follows, based on an assumption that server ports opened on the c4a interface should be globally visible while server ports opened on the c4b interface should not be visible from the 10.0.0.0/8 network.

Index	Requirement	Bind Policy
1	Requests to open a server port on the IPv4 loopback address open only the IPv4 loopback address.	`127.0.0.1/32`

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

Index	Requirement	Bind Policy
2	Requests to open a server port on c4a open the c4a and c4b addresses.	`c4a/32 c4a c4b`
3	The local address for client connections to any remote address in the 192.168.0.0/24 address range is c4b.	`c4b/24`
4	The local address for client connections to any remote IPv4 address not in the 192.168.0.0/24 address range is c4a.	`c4a/0`

Voyager should be started with the address c4a in the startup URL.

Configuration for mixed IPv4 and IPv6

This configuration requirements and cooresponding policies are as follows.

Index	Requirement	Bind Policy
1	Services on host C that open a server port on c4a (10.0.0.2) will open the port on c6a, c4b, c6b, and both loopback addresses.	`c4a/32 c4a c6a c4b c6b 127.0.0.1 ::1`
2	Services on host C that open a server port on c6a (3000::2) will open the port on c6a, c4b, c6b, and both loopback addresses.	`C6a/128 c4a c6a c4b c6b 127.0.0.1 ::1`
3	Services on host C that open a server port on c4b will open that port on only c4b and c6b.	`c4b/32 c4b c6b`
4	Services on host C that open the loopback address open that port only the loopback addresses.	`127.0.0.1/8 127.0.0.1`
5	When connecting to a remote host on the 192.168.0.0/24 network from C, the local client address shall be c4b.	`c4b/24`
6	When connecting to a remote host on the 2000::/48 network from C, the local client address shall be c6b.	`c6b/48`
7	When a client requests a connection to C's IPv4 loopback address, the local client address shall the IPv4 loopback address.	`127.0.0.1/8`
8	When a client requests a connection to C's IPv6 loopback address, the local client address shall the IPv6 loopback address.	`::1/128`
9	When connecting to a remote host on any IPv4 address not in the 192.168.0.0/24 network, the local client address shall be c4a (10.0.0.2).	`c4a/0`
10	When connecting to a remote host's globally scoped address on the IPv6 network c6b/48 (2000::/48), the local client address shall be c6b.	`c6b/48`
11	When connecting to a remote host's globally scoped address on any IPv6 network other than 2000::/48, the local client address shall be c6a.	`c6a/0`

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

Index	Requirement	Bind Policy
12	When connecting to a remote host's locally scoped address on any IPv6 network the local client addresses shall be fe80::2 and fe80::3.	`fe80::0/0 fe80::2 fe80::3`

The textual policy that satisfying requirement 2 is `c4a/32 c4a c4b 127.0.0.1`, which says that any request to open a server port on address c4a (the /32 means all 32 bits of the address must match) actually opens the port on c4a, c4b, and the loopback address.

The property file lines for host C satisfying the above requirements are as follows.

```
Voyager.tcp.BindPolicyFactory.implementMultiHomeBindPolic
ies= c4a/32 c4a c6a c4b c6b 127.0.0.1 ::1; C6a/128 c4a
c6a c4b c6b 127.0.0.1 ::1; c4b/32 c4b c6b; 127.0.0.1/8
127.0.0.1
Voyager.tcp.BindPolicyFactory.implementClientBindPolicies
= c4b/24; c6b/48; 127.0.0.1/8; ::1/128; c4a/0; c6b/48;
c6a/0; fe80::0/0 fe80::2 fe80::3
```

The more general policies' network specifications are placed after less specific ones in the sequence since an address lookup proceeds from left to right. In this configuration the sequence of networks in the multihome policy line doesn't matter, but in the client bind policy line if c4a/0 appeared to the left of c4b/24, all addresses in the c4b/24 network would match c4a/0 and never match c4b/24.

Host R

Host R configurations look like host L's, differing only because host R is on different networks.

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

Appendices

Appendix A – Deployment

Voyager supports JSE, JME, and .NET runtime environments. Different libraries need to be deployed depending on your target virtual machine environment and the desired Voyager features. The following sections discuss the details of deploying to each environment. For the location of installed files refer to Voyager Installation Directories.

In addition to the discussion below, classes and interfaces need to be kept consistent within a Voyager network. If a class or interface it changed, its .class file or assembly must be updated on all Voyager deployments. For further information refer to The Interface must be Common and Serving Classes.

.NET Deployment

The following summarizes the general functionality of the .NET assemblies.

- `ve-core.dll` – Core functionality of Voyager
- `ve.cli.runtime.dll` – A modified version of the IKVM.NET runtime library by Jeroen Frijters. See www.ikvm.net for information about IKVM. This runtime assembly provides basic class equivalence mapping for Object, String, and Exception classes as well as reflection and class loading assistance.
- `ve.portability.dll` – The Java runtime libraries used by `ve-core.dll`, `ve.cli.runtime.dll`, and the `cligen` utility.

The assemblies provided in the installation are signed which allow several deployment options.

1. Insert the assemblies in the global assembly cache (GAC). Not recommended.
2. Reference the assemblies with an `<application>.config` file.
3. Place the assemblies in the directory of the application.

The following table lists the assemblies needed for deployment to a .NET environment:

Core Features	Rules Features	Database Features
`ve-core.dll`	`n/a`	`n/a`
`ve.cli.runtime.dll`		

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

ve.portability.dll		

JEE Deployment

See JSE Deployment.

JSE Deployment

The following table lists the jar files need for deployment to a JSE environment:

Core Features	Rules Features	Database Features
ve-core.jar	ve-ai.jar	commons-dbutils-1.0.jar
	antlr-2.7.6.jar	ve-db.jar
	antlr3-3.0ea8.jar	
	commons-jci-core-1.0-406301.jar	
	commons-jci-janino-2.4.3.jar	
	commons-lang-2.1.jar	
	commons-logging-api-1.0.4.jar	
	drools-compiler-3.0.1.jar	
	drools-core-3.0.1.jar	
	janino-2.4.3.jar	
	jung-1.7.2.jar	
	stringtemplate-2.3b6.jar	
	xpp3-1.1.3.4.O.jar	
	xstream-1.1.3.jar	

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

IBM J9 Deployment

Voyager has been tested with IBM's J9 6.1 WEME configured for CDC Personal Profile 1.1 support.

Prerequisites

1. Verify that the IBM J9 CDC 1.1 PersonalProfile 1.1 VM is installed and working on the device.
2. Add the JDBC Optional Package (JDBCOP) to the device. This is an optional package for JME CDC that is available using the IBM WebSphere Studio Device Developer. Start the Device Developer environment. Select "Help -> Software Updates -> Update Manager". Expand "Sites to Visit" and expand "IBM Micro Environment Toolkit for WebSphere Studio" and expand "Extension Services For WebSphere Everyplace" and select "JSR 169 (BETA) for Extension Services 5.7.1" and install it. Note that you may need to install additional dependencies. Once successful the JDBCOP package will be located in `WSDD_HOME\wsdd5.0\technologies\eswe\bundlefiles\jdbc.jar`
   Copy the `jdbc.jar` file into the device's directory `J9_HOME\lib\jclFoundation11\ext`

The following table lists the jar files that need to be deployed for each Voyager feature:

Core Features	Rules Features	Database Feature
ve-core-cdc.jar	gcp.jar	gcp.jar
jta.jar	ve-ai-cdc.jar	ve-db.jar
jnet.jar	drools-core-3.0.1-cdc.jar	bcprov-jdk13-133.jar
	drools-compiler-3.0.1.jar	commons-dbutils-1.0.jar
	jakarta-regexp-1.4.jar	jce-jdk13-133.jar
	jaxp-api.jar	
	sax.jar	

The library gcp.jar contains classes with package names beginning with *java* and must be deployed to `J9_HOME\lib\jclFoundation11\ext`

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

IBM J9 JCE Provider Issues

The files bcprov-jdk13-133.jar and jce-jdk13-133.jar implement a JCE provider developed by Bouncy Castle. IBM's WEME J9 6.1 has a bug in the JCE provider that prevents SealedJDBCProperties from working properly. If deploying in this environment, you will need to configure the J9 JRE to use a different JCE provider. The JCE from Bouncy Castle is known to work. The following describes how to configure Bouncy Castle as the JCE provider:

3. Copy jce-jdk13-133.jar and bcprov-jdk13-133.jar to J9_HOME/lib/jclFoundation11/ext
4. Add security.provider.2=org.bouncycastle.jce.provider.BouncyCastleProvider to J9_HOME/lib/security/java.security
5. Comment out the line security.provider.2=com.ibm.j9.jce.provider.J9JCEProvider in the java.security file by placing a '#' character at the beginning of the line.

The J9 JCE problem has been reported to IBM. A future release of J9 will likely correct this problem. When the problem is resolved, it will not be necessary to configure an alternative JCE provider such as Bouncy Castle.

Appendix B – Utilities

Overview

In this chapter, you will learn to:

6. Use the `voyager` or `voyager_net` utility to start a Voyager server from the command line.
7. Use the `igen` utility to generate a default interface from a class.
8. Use the `pgen` utility to generate the source or bytecode form of a proxy class for a given class.
9. Use the `cligen` to generate a .NET native assembly from a set of Java .class files.

Setting your CLASSPATH for the Voyager Utilities

Note: This section is specific to Java deployment. For .NET deployment make sure CLASSPATH is **not** set. See Loading Classes in .NET for a description of important points about loading assemblies.

When executed, the Voyager utilities search for both your source code and your object code via `CLASSPATH`. The search is successful when your source files and object files reside in the same directory. If your source files and object files reside in different

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

directories, ensure that the directory structure leading to source files mirrors the directory structure leading to object files. Add the root of each path to `CLASSPATH` .

For example, if the source code and object code for the `com.foobar` package is organized as follows, the `CLASSPATH` must include both `\root` and `\root\src`:

```
\root
  \com
    \foobar
     .class files (object code)
  \src
    \com
      \foobar
       .java files (source code)
```

voyager

The `voyager` utility starts a Java Voyager server from the command line. To see usage information, type `voyager` with no arguments. A description of each argument follows.

Argument	Example	Description
`xurl`	`//dallas:8000`	The URL Voyager will accept incoming messages on. See the Voyager Basics chapter for the format of this URL. Voyager uses the URL as the name of the ServerContext.
`-a filename`	`-a arguments.txt`	Processes lines in the given file as if they were commandline parameters.
`-b classname`	`-b com.foo.FooBar`	Load the specified class and then execute its `static main(String[] args )` method. To supply args, specify the class and arguments within quotes.
`-c URL`	`-c http://dallas:9000`	Enable network class loading from the specified URL. You can specify this option multiple times.
`-i program`	`-i jview`	Specify an interpreter other than *java* to execute Voyager.
`-l loglevel`	`-l verbose`	Enable logging at the specified level (silent, exceptions, verbose). Messages are printed on the Java console.
`-p filename`	`-p Voyager.properties`	Load Voyager properties from the given file. These properties are processed after

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

		commandline arguments are processed, and consequently override any commandline arguments for those properties.
`-q`	`-q`	Suppress display of Voyager copyright notice on startup.
`-r`	`-r`	Enable Voyager's built-in HTTP resource server to provide classes to remote Voyager instances.
`-s`	`-s`	Enable Voyager's security manager. Enabling this is equivalent to invoking `System.setSecurityManaget( new VoyagerSecurityManager() ).`
`-max_user_threads size`	`-max_user_threads 50`	Set the maximum number of threads allowable in Voyager's thread pool.
`-v`	`-v`	Print version information and exit.
`-x parameters`	`-x -Djava.compiler=none`	Pass remaining arguments to Java interpreter. All commandline arguments after `-x` will be passed to the Java interpreter, so this must be the last option specified on the Voyager commandline.

voyager_net

Note: This section and the `voyager_net.exe` utility are deprecated as are the DLLs referenced by `voyager_net.exe`. Instead, please use a Visual Studio project to launch your application. Refer to the examples solution that is provided with Voyager for examples.

The `voyager_net` utility starts a .NET Voyager server from the command line. To see usage information, type `voyager_net` with no arguments. A description of each argument follows.

Argument	Example	Description
`xurl`	`//dallas:8000`	The URL Voyager will accept incoming messages on. See the Voyager Basics chapter for the format of this URL.
`-a assemblyName`	`-a ..\..\myAssembly.dll`	Dynamically load an assembly. Classes in the assembly become available for Voyager to use

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

		if needed.
`-ad directory`	`-ad ..\..`	Dynamically load all files is the directory that end with ".dll" as assemblies.
`-l loglevel`	`-l verbose`	Enable logging at the specified level (silent, exceptions, verbose). Messages are printed on the Java console.
`-verbose`	`-verbose`	Same as `'-l verbose'`
`-tl`	`-tl`	Trace all dynamic loading of assemblies
anything	*anything*	Any other arguments are handled as in voyager

igen

Note: This utility requires a Java VM to run. Once this utility is used the `cligen` utility can be used to compile the igen generated .class file into a native .NET assembly.

The `igen` utility creates a default Java interface from a Java class. The interface has all the public methods of the original class, and is named to the original class name prefixed with `I`. For a list of the `igen` run-time options, run `igen` from the command line with no parameters.

To use the `igen` utility, specify the class name without the `.class` or `.java` extension. `igen` searches the directories, `.zip` files, and `.jar` files in the `CLASSPATH` for the specified file and generates an interface for the class. If the current directory contains the source or object code of the original class, typing the full class name is optional. You can generate interfaces for multiple classes by naming the classes separated by a space on the command line.

By default, `igen` places interfaces created from `java.*` classes into the related `com.recursionsw.java.*` package. The `igen` utility reminds you of this behavior with a note each time you run `igen` on a `java.*` class.

To see which classes Voyager is dynamically generating at runtime, use the `-verbose` parameter. Along with the normal verbose output, the name of each dynamically generated class will be printed.

For example, to create the interface `.\IVector` from `java.util.Vector` , execute the `igen` utility as shown.

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

```
C:igen4java.bat java.util.Vector
igen 7.2.1.0 Copyright 2009 Recursion Software, Inc.
note: java.* interfaces are placed into com.recursionsw.java.*
```

The `igen` utility generates an interface for all classes in the class's hierarchy. For example, when `igen` is run on `Hashtable`, it generates `IHashtable` and `IDictionary` because `Hashtable` extends `Dictionary`. A description of each `igen` argument follows.

Argument	Example	Description
`-d directory`	`-d \java\classes`	Specify a root directory for igen. Igen will place generated classes in subdirectories of this directory. This option is analogous to the -d option of `javac`.
`-i program`	`-i jview`	Specify an interpreter other than `java` to execute igen.
`-q`	`-q`	Suppress display of copyright notice on startup.
`-v`	`-v`	Print verbose status information to the Java console.
`-x parameters`	`-x -Djava.compiler=none`	Pass remaining arguments to Java interpreter. All commandline arguments after `-x` will be passed to the Java interpreter, so this must be the last option specified on the commandline.

pgen

Note: This utility requires a Java VM to run. Once this utility is used the `cligen` utility can be used to compile the `pgen` generated Java .class files into a native .NET assembly.

The `pgen` utility generates the proxy class source code or bytecode for a given class.

Manual Proxy Class Generation

Dynamic proxy generation is most useful as an aid to development. Dynamic proxy generation lets the developer focus on creating application logic instead of being distracted by the steps required for manual proxy generation and the associated class synchronization problems. However, Voyager offers manual proxy generation for situations when performance is critical, proxy classes need to be post-processed, or when classloading issues prevent the use of the Voyager classloader (such as when classes are being provided by a non-Voyager HTTP server).

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

To see which classes Voyager is dynamically generating at runtime, use the `-verbose` parameter when running Voyager. Along with other verbose output, the name of each dynamically generated class will be printed.

Generating Proxy Classes

Voyager uses two different types of proxy classes. Normal proxy classes are used when the client is a normal Voyager application that has access to all Voyager classes. These proxy classes are by default generated from interfaces (interface-based proxies), but may also be generated from the actual class (class-based proxies). The default interface-based proxies are recommended. Class-based proxies require all VMs to have access to the implementation class.

To create the proxy class for `java.util.Vector`, execute the `pgen` utility as shown.

```
C:>pgen4java.bat java.util.Vector
Note: java.* proxy classes are placed into com.recursionsw.java.*
pgen 7.2.1.0, Copyright 2009 Recursion Software, Inc.
```

You can generate the proxy classes for multiple classes by naming the classes separated by a space on the command line.

Performance

Generation of proxy classes typically takes less than 250 milliseconds. After the proxy class for a given class has been generated, it never needs to be generated again for the lifetime of the VM's process. Though this one-time hit is rarely noticeable over the lifetime of the application, it can have a noticeable impact on the perceived system performance at proxy generation time, particularly when many classes are processed at once, such as in system startup. By generating proxy classes manually, proxy class loading becomes equivalent to loading of any class, and is unnoticeable. Note that loading proxy classes across a network may in some cases be slower than generating them dynamically, and other variables may affect performance of dynamic versus static proxy class generation.

Post-Processing

Some designs require modifications of each proxy class to allow custom functionality. For example, you can modify each proxy method to print to a log when invoked. These modifications are impossible to make to proxies generated on the fly. By generating proxy source code, you can modify the default proxy logic, providing custom functionality to proxy classes.

pgen Command Line Options

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

For a list of the `pgen` run-time options, run `pgen` from the command line with no parameters. A description of each `pgen` option follows.

Argument	Example	Description
`-d directory`	`-d \java\classes`	Specify a root directory for `pgen`. `pgen` will place generated classes in subdirectories of this directory. This option is analogous to the `-d` option of `javac`.
`-i program`	`-i jview`	Specify an interpreter other than `java` to execute pgen.
`-q`	`-q`	Suppress display of copyright notice on startup.
`-v`	`-v`	Print verbose status information to the Java console.
`-s`	`-s`	Generate Java source code instead of bytecode for generated classes.
`-c`	`-c`	Generate class-based proxy classes instead of interface-based proxy classes.

cligen

The cligen utility is a modified version of the IKVM.NET Compiler by Jeroen Frijters.

The `cligen` utility will convert existing Java `.class` files into a native .NET assembly. The compilation is based on converting the Java bytecode to CIL bytecode. Missing types will produce warnings, which may later result in runtime errors if the missing type was needed at runtime.

```
usage: cligen [-options] <classOrJar1> ... <classOrJarN>
```

An example of converting some Java bytecode to a .NET assembly is shown here.

```
cligen –assembly:Mytest –target:library Utils.jar mytest\*.class
```

Selected options are listed here. For detailed documentation and additional options see the www.ikvm.net

Argument	Example	Description
`-out:<name>`	`-out:somefile.dll`	Specify output filename
`-assembly:<name>`	`-assembly:Mytest`	Specify assembly name
`-target:exe`	`-target:exe`	Build a console executable
`-target:winexe`	`-target:winexe`	Build a windows executable
`-target:library`	`-target:library`	Build a library

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

`-target:module`	`-target:module`	Build a module for use by the linker
`-keyfile:<name>`	`-keyfile:mykeyfile`	Use a key file to sign the assembly
`-main:<class>`	`-main:MyMain`	Specify the class containing the main method
`-debug`	`-debug`	Generate debug information. To successfully debug the source code the Java .class files must have been built with debug information.

Appendix C – Examples

This appendix provides an overview of the examples which cover the core Voyager features.

Running the Examples

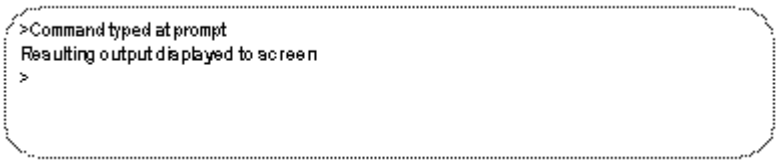
After you install Voyager, the source code for these examples is located in the `examples` directory. Each example description specifies the directory in which the example resides. For Java the CLASSPATH must include `ve-core.jar` **and** the examples class files to successfully run the examples. For .NET, do **not** set CLASSPATH but use the provided project file to build the examples, "Rebuild Solution", and then use a command shell to execute the example. Each example is presented as follows.

1. The command(s) used to prepare the example program for execution are presented. Commands that generate interfaces, generate holders, and compile source code belong in this category.

2. The command(s) used to run the example program are presented, followed by the program output.

3. The source code location for the example programs is listed.

Commands the user types and the resulting output displayed are presented as shown below.



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

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

Running under Eclipse

The eclipse project for the examples can be found at:

`%VOYAGER_HOME%\examples\java\se-cdc\eclipse_project\VoyagerExamples`

Please refer to "Getting started with Eclipse" guide (%VOYAGER_HOME%\doc\ EclipseGettingStartedGuide.pdf) for details on setting up the Eclipse project for the examples. Details for running the examples can be found in the readme.txt file located in the respective directory of the example.

Running under Visual Studio

Please refer to "Getting started with Microsoft Visual Studio" guide (%VOYAGER_HOME %\doc\ VisualStudioGettingStartedGuide.pdf) for details on running the examples from VisualStudio. Details for running the examples can be found in the readme.txt file located in the respective directory of the example.

Basics

The "Basics" examples in this section demonstrate basic Voyager functions like messaging, remote construction, naming and lookup, and remote class loading.

Basics1 Example

The `Basics1` example uses Voyager's remote construction mechanism to construct a `Stockmarket` object in a remote Voyager server. It then sends the object messages. The last message demonstrates how exceptions thrown on the server are transparently propagated to the client.

The remote server does not terminate automatically. A Voyager server must be terminated explicitly, preferably by using `Voyager.shutdown()`.

Basics2 Example

The `Basics2` example demonstrates sending a remote message to a proxy obtained by a `Namespace` lookup. The `Basics2A` program exports the object to a specified port and

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

binds the associated proxy into the `Namespace`. `Basics2B` looks up the object and sends it a message.

Source code location

The java examples can be found under the %VOYAGER_HOME%\examples\java\se-cdc directory. The corresponding csharp examples can be found under %VOYAGER_HOME %\examples directory.

Java

```
java\examples\basics\Basics1.java
java\examples\basics\Basics2A.java
java\examples\basics\Basics2B.java
java\examples\stockmarket\IStockmarket.java
java\examples\stockmarket\Stockmarket.java
```

C#

```
csharp\Common\ExamplesCommon\examples\Basics1Example.cs
csharp\Common\ExamplesCommon\examples\Basics2Example.cs
csharp\Common\ExamplesCommon\IStockmarket.cs
csharp\Common\ExamplesCommon\Stockmarket.cs
```

Running under Eclipse
Running under Visual Studio

Dynamic Aggregation

The "Aggregation" examples in this section explain Voyager's dynamic aggregation framework.

Aggregation1 Example

This example demonstrates facet creation and remote access of facets. An Account facet is added to an `Employee` object in `Aggregation1A`. It is then accessed remotely in `Aggregation1B`.

Aggregation2 Example

This example differs from the Aggregation1 Example in that it demonstrates the use of the `of()` and `get()` convenience methods. Not only do those methods present a cleaner API for frequently used aggregation operations, they also ensure compile-time checking of facet class types, as opposed to run-time parsing of String names.

Aggregation3 Example

This example demonstrates custom facet to class mapping. In the example, instances of class `Employee` get a custom facet of type `EmployeeBonusPlan`, whereas instances of `Programmer` get a custom facet of type `ProgrammerBonusPlan`. An attempt is made to get a `BonusPlan` facet for an instance of String. Because no custom facet class is found

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

using the class-matching rules, and there is no default `BonusPlan` facet class, an exception is thrown.

Assuming the files have been compiled for the first aggregation example, run the **`Aggregation3`** example

Aggregation4 Example

This example demonstrates the facet-aware facet class `ManagerBonusPlan`. When an instance of `BonusPlan` is aggregated with an instance of `Manager`, the `ManagerBonusPlan` class is chosen. Because it implements `IFacet` and provides the correct constructor, it is constructed with the `Facets` object of the `Manager` primary object.

Assuming the files have been compiled for the first aggregation example, run the `Aggregation4` example.

Source code location

The examples can be found under the %VOYAGER_HOME%\examples\java\se-cdc directory.

Java

```
java\examples\aggregation\Aggregation1A.java
java\examples\aggregation\Aggregation1B.java
java\examples\aggregation\IEmployee.java
java\examples\aggregation\Employee.java
java\examples\aggregation\IAccount.java
java\examples\aggregation\Account.java

java\examples\aggregation\Aggregation2A.java
java\examples\aggregation\Aggregation2B.java
java\examples\aggregation\ISecurity.java
java\examples\aggregation\Security.java

java\examples\aggregation\Aggregation3.java
java\examples\aggregation\IProgrammer.java
java\examples\aggregation\Programmer.java
java\examples\aggregation\IBonusPlan.java
java\examples\aggregation\BonusPlan.java
java\examples\aggregation\EmployeeBonusPlan.java
java\examples\aggregation\ProgrammerBonusPlan.java

java\examples\aggregation\Aggregation4.java
java\examples\aggregation\IManager.java
java\examples\aggregation\Manager.java
java\examples\aggregation\ManagerBonusPlan.java
```

Running under Eclipse

Advanced Messaging

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

The “Messaging” examples in this section demonstrate more advanced forms of remote messaging such as one-way, future, remote static, and dynamic synchronous invocation.

Message1 Example

The `Message1` example demonstrates dynamic invocation for remote method invocation. It creates an `Alarm` object in a remote `Voyager` server. It then invokes an instance method synchronously. Next, it invokes a static method on the `Alarm` class in the remote `Voyager` server.

Message2 Example

The `Message2` example demonstrates invocation of a one-way message. It creates an `Alarm` object in a remote `Voyager` server. It then invokes a one-way instance method on the alarm. The example pauses one second to delay shutdown to allow time for the one-way invocation to fully flush from the program.

Message3 Example

The `Message3` example demonstrates invocation of a future message. It creates an `Alarm` object in a remote `Voyager` server. It then invokes a future instance method on the alarm. Because the future invocation is asynchronous, the program can to execute other code while the message is being delivered. It then executes a blocking read on the result. Next, the program demonstrates how reading a result from a future can re-throw any exception that occurs during delivery or execution of the future message.

Message4 Example

The `Message4` example demonstrates invocation of a future message with listeners. It creates an `Alarm` object in a remote `Voyager` server. It then invokes a future instance method on the alarm, passing in an array of listeners. These listeners receive a callback when the result of the future invocation is received.

Message5 Example

The `Message5` example demonstrates invocation of a future message with two threads blocking on the result. When the result of the future invocation is received, both blocking threads read the return value.

Message6 Example

The `Message6` example demonstrates invocation of a future message with a timeout. The message is designed to take longer than the timeout value. Consequently, a `TimeoutException` is thrown.

Message7 Example

The `Message7` example demonstrates a special form of dynamic invocation that returns results by reference instead of by value. The program invokes a static method on the

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

`Alarm` class in a remote `Voyager` server. This method returns a new `Alarm` object; however, the invocation uses return by reference, so the client receives a proxy to the remote alarm object instead of the alarm object itself.

Source code location

The java examples can be found under the %VOYAGER_HOME%\examples\java\se-cdc directory. The csharp examples can be found under the %VOYAGER_HOME %\examples directory.

Java

```
java\examples\message\Message1.java
java\examples\message\IAlarm.java
java\examples\message\Alarm.java

java\examples\message\Message2.java
java\examples\message\Message3.java
java\examples\message\Message4.java
java\examples\message\Message5.java
java\examples\message\Message6.java
java\examples\message\Message7.java
```

C#

```
csharp\Common\ExamplesCommon\examples\Message1Example.cs
csharp\Common\ExamplesCommon\examples\Message2Example.cs
csharp\Common\ExamplesCommon\examples\Message3Example.cs
csharp\Common\ExamplesCommon\examples\Message4Example.cs
csharp\Common\ExamplesCommon\examples\Message5Example.cs
csharp\Common\ExamplesCommon\examples\Message6Example.cs
csharp\Common\ExamplesCommon\examples\Message7Example.cs
```

Running under Eclipse
Running under Visual Studio

Multicast and Publish/Subscribe

The “Multicast and Publish / Subscribe” examples in this section demonstrate `Voyager`'s multicast and publish/subscribe features.

Space1 Example

The `Space1` example demonstrates constructing and populating a space. It first constructs a subspace in a remote `Voyager` server on port `8000` and populates it with `Consumer` objects constructed in the same server. It then constructs another subspace in a remote `Voyager` server on port `9000` and populates it with `Consumer` objects. Finally, the two subspaces are connected to form a single distributed space.

Space1Jms Example

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

Like the `Space1` example, the `Space1Jms` example demonstrates constructing and populating a space but this example uses JMS as the transport rather than TCP. The two examples differ only in how the subspaces connect to form a single distributed space.

Space2 Example

The `Space2` example demonstrates two forms of distributed multicast. The first form demonstrates multicast of standard Java method invocations. The second form demonstrates distributed JavaBeans-style event multicasting. Both multicasts are sent into the space constructed in `Space1` or `Space1Jms` by using one of the subspaces as a gateway.

Space3 Example

The `Space3` example demonstrates `Voyager`'s publish/subscribe mechanism. It uses the space that is built by the `Space1` or `Space1Jms` example. The program creates a subscriber for three of the consumers created in `Space1` and subscribes each to a given topic. The subscriber for a given consumer is added to the subspace local to the consumer. It can therefore receive messages that are published to that subspace. The program uses a `ConsumerAdapter` to allow filtering of published events without coupling the `Consumer` class to the details of the publish/subscribe mechanism. Next, the program publishes messages to each of the three topics.
Note: Subscriber objects must be manually removed from a subspace when the client disconnects, otherwise they will be orphaned on the server and never garbage collected.

Source code location

The examples can be found under the %VOYAGER_HOME%\examples\java\se-cdc directory.

Java

```
java\examples\space\Space1.java
java\examples\space\IConsumer.java
java\examples\space\Consumer.java

java\examples\space\Space1Jms.java
java\examples\space\IJmsExampleConstants.java

java\examples\space\Space2.java
java\examples\space\NewsEvent.java
java\examples\space\NewsListener.java
java\examples\space\Reporter.java

java\examples\space\Space3.java
java\examples\space\ConsumerAdapter.java
```

Running under Eclipse

Mobility

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

The “Mobility” examples in this section demonstrate how any serializable object can be moved around the network.

Mobility1 Example

The `Mobility1` example demonstrates mobility and messaging. An object is constructed, moved and sent messages. The movement of the object is transparent to the client. The client's reference to the moving object is valid whether the object is local, remote or in the process of moving.

Mobility2 Example

The `Mobility2` example demonstrates receiving mobility callbacks using the `IMobile` interface. An object is constructed and moved around. The various move callbacks are executed on the object throughout the operation.

Source code location

The examples can be found under the %VOYAGER_HOME%\examples\java\se-cdc directory.

Java

```
java\examples\mobility\Mobility1.java
java\examples\mobility\IDrone.java
java\examples\mobility\Drone1.java

java\examples\mobility\Mobility2.java
java\examples\mobility\Drone2.java
```

Running under Eclipse

Agents

The “Agents” example in this section demonstrates how any serializable object can use `Voyager`'s dynamic aggregation feature to become a mobile, autonomous agent.

Agents1 Example

The `Agents1` example demonstrates mobile autonomous agents. An object uses dynamic aggregation to access its `Agent` facet. This allows the object to move itself around the network. When the agent has completed its tasks, it disables its autonomy, allowing the agent to be garbage collected.

Note that messaging speeds are greatly improved after the agent co-locates itself with its target.

Source code location

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

The examples can be found under the %VOYAGER_HOME%\examples\java\se-cdc directory.

Java
```
java\examples\agents\ITrader.java
java\examples\agents\Trader.java
java\examples\agents\Agents1.java
```

Running under Eclipse

Naming Service

The "Naming Service" examples in this section demonstrate `Voyager`'s federated directory system and pluggable naming service.

Naming1 Example

The `Naming1` example demonstrates the federated directory system. It first creates a directory in a remote `Voyager` server into which it places a few items. It then creates another directory on a different remote `Voyager` server and binds the first into the second. Next, it demonstrates traversal of names across the federated directory service.

Naming2 Example

The `Naming2` example demonstrates `Voyager`'s naming service. It first creates an object in a remote `Voyager` server. It then binds that object to a name in that server's name space. After the object is bound to a name, the program is able to look up that object by name. The program then unbinds the object and demonstrates that lookup with the old name will no longer succeed.

Source code location

The java examples can be found under the %VOYAGER_HOME%\examples\java\se-cdc directory. The corresponding c# examples can be found under the %VOYAGER_HOME %\examples directory.

Java
```
java\examples\naming\Naming1.java
java\examples\naming\Naming2.java
```

C#
```
csharp\Common\ExamplesCommon\examples\Naming1Example.cs
csharp\Common\ExamplesCommon\examples\Naming2Example.cs
```

Running under Eclipse
Running under Visual Studio

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

Yellow Pages

The “Yellow Pages” examples in this section explain Voyager's Yellow Pages Directory.

Yellow Pages 1 Example

This example demonstrates how to acquire and use a Yellow Pages instance to register and lookup services.

Source code location

The examples can be found under the %VOYAGER_HOME%\examples\java\se-cdc directory.

Java

```
java\examples\yellowpages\ExchangeBroker.java
java\examples\yellowpages\IExchangeBroker.java
java\examples\yellowpages\NASDAQExchange.java
java\examples\yellowpages\NYSEExchange.java

java\examples\yellowpages\PEStockRecommendationService.java
java\examples\yellowpages\TrendStockRecommendationService.java
java\examples\yellowpages\StockPickerClient.java
```

Running under Eclipse

Audit

Audit1 Example

The Audit1 example illustrates use of the Audit service in a standalone program, i.e., a Voyager without an Audit neighbor. The application changes the Audit File Logger’s directory and file name prefix, and sets up the Console Logger’s filters print all messages. After initializing Voyager, the program creates a session object and two record objects. The record objects are then added to the stream of audit records by a call to each record’s commit() method.
When executed the example will display the two audit records and write the same two records to a file in the /temp directory on the current drive. The file name will begin with the prefix string “Audit1ExampleLog” and end with the suffix string “.txt”, e.g., “Basics1AuditLog20070613163851GMT.txt”. The characters between the prefix and suffix are time stamp that makes the file name unique.
The two records will look similar to the following two examples. Line breaks have been added before each major part of the records to make them easier to read. The records contain no delimiters other than the colons.

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

```
HDR:263:61:0x46703ca6:::tick.usno.navy.mil:CST6CDT:0x1000025:0x0:
ORG:joedeveloper-xp:dal-exch.recursionsw.com::recursionsw:Joe
Developer:joedeveloper:
INT:auth service:name in auth svc:id in auth svc:
TGT:Voyager:localhost%:8000:::::
SRC::
EVT:Test record 1!:
END

HDR:280:61:0x46703ca6:::tock.usno.navy.mil:CST6CDT:0x1000026:0x0:
ORG:joedeveloper-xp:dal-exch.recursionsw.com::recursionsw:Joe
Developer:joedeveloper:
INT:System Operator:John Smith:523:
TGT:Headquarters:1001 Headquarter Drive, Plano, Tx:::John Smith:523:
SRC::
EVT:Test record 2!:
END
```

Audit2 Example

The Audit2 example consists of three programs: Audit2a, a neighbor acting as both a peer and a parent; Audit2b, a neighbor acting as a child; and Audit2aPeer a neighbor acting as a peer. The Audit2a program must be started first. The Audit2aPeer and Audit2b programs can be started in any order.

When started, the Audit2a program configures the console logger filters to print records with success and denial outcomes, but suppress failure outcomes. Audit2a then configures the file logger and waits, with a timeout, for the two neighbors to connect, then commits an audit record with a failure outcome code. Audit2a should display two audit records on the console and write all four records to the log file.

When started, the Audit2aPeer program configures the console logger to print records with all outcomes, connects to Audit2a as a peer then commits an audit record with a failure outcome code. Audit2aPeer does not configure the file logger, and as a result, does not create a log file. Audit2aPeer then sleeps for a few seconds to give the other programs' audit records time to arrive, then quits. Audit2aPeer should display on the console all four records.

When started, the Audit2b program configures the console logger filters to print records with denial and failure outcomes, but suppress records with success outcomes. Audit2b does not configure the file logger, and as a result, does not create a log file. Audit2b then connects to Audit2a as a child and commits two audit records with success outcomes. Audit2b should display no audit records on the console.

The four records, which may be displayed in a different order, will look similar to the following example records.

```
HDR:344:61:0x4670406d:::localhost:CST6CDT:0x1000025:0x0:
ORG:joedeveloper-xp:dal-exch.recursionsw.com::recursionsw:Joe
Developer:joedeveloper:
INT:Initiator Authority:Domain-specific initiator name:Domain-specific
initiator ID:
```

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

```
TGT:Target Location Name:Location
Address:ServiceType:Authority:Principal Name:Principal ID:
SRC::
EVT:Test record 1!:
END

HDR:271:61:0x4670406d:::localhost:CST6CDT:0x1000025:0x0:
ORG:joedeveloper-xp:dal-exch.recursionsw.com::recursionsw:Joe
Developer:joedeveloper:
INT:System Operator:John Smith:523:
TGT:Headquarters:1001 Headquarter Drive, Plano, Tx:::John Smith:523:
SRC::
EVT:Test record 2!:
END

HDR:270:61:0x46704065:::localhost:CST6CDT:0x1000025:0x1:
ORG:joedeveloper-xp.recursionsw.com:dal-
exch.recursionsw.com::recursionsw:Joe Developer:joedeveloper:
INT:auth service:name in auth svc:id in auth svc:
TGT:Voyager:localhost%:8000:::::
SRC::
EVT:Test record 3!:
END

HDR:270:61:0x4670406a:::localhost:CST6CDT:0x1000025:0x1:
ORG:joedeveloper-xp.recursionsw.com:dal-
exch.recursionsw.com::recursionsw:Joe Developer:joedeveloper:
INT:auth service:name in auth svc:id in auth svc:
TGT:Voyager:localhost%:8000:::::
SRC::
EVT:Test record 4!:
END
```

Source code location

The examples can be found under the %VOYAGER_HOME%\examples\java\se-cdc directory.

Java

```
java\examples\audit\audit1.java

java\examples\audit\audit2.java
java\examples\audit\audit2a.java
java\examples\audit\audit2aPeer.java
java\examples\audit\audit2b.java
```

Running under Eclipse

Security

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

The examples in this section demonstrate Voyager's support for Java security features, and also involve Voyager's ability to serve as a resource loader.

Security Example 1

The `Security1` program sends a `Visitor` object to two remote locations, one "native" server, started in the same directory as the `Security1` program, and one "foreign" server, started in a different directory. The native server's classloader finds the `Visitor` object in the startup directory because '.' is in the CLASSPATH. However, because the "foreign" program started in a different directory the Voyager class loader must use a resource loader to load the `Visitor` class, and therefore will not trust instances of the `Visitor` class.

Source code location

The examples can be found under the %`VOYAGER_HOME%\examples\java\se-cdc` directory.

Java

```
java\examples\security\nativesec\Security1.java
java\examples\security\nativesec\IVisitor.java
java\examples\security\nativesec\Visitor.java
java\examples\security\nativesec\Native.java
java\examples\security\nativesec\java.policy

java\examples\security\foreignsec\Foreign.java
java\examples\security\foreignsec\java.policy
```

Running under Eclipse

Timers

The examples in this section demonstrate Voyager's Timing services.

Stopwatch1 Example

The `Stopwatch1` example demonstrates use of the `Stopwatch` class to clock time intervals.

Timer1 Example

The `Timer1` example demonstrates `TimerListeners` listening for an alarm using shared thread notification. The first listener's event notification completes before the second listener's event notification begins.

Timer2 Example

The `Timer2` example demonstrates `TimerListeners` listening from within different timer groups. The event notifications are interlaced due to the multi-threaded listening.

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

Timer3 Example

The `Timer3` example demonstrates `TimerListeners` listening for an alarm using separate thread notification. The event notifications are interlaced due to the multi-threaded listening.

Source code location

The java examples can be found under the %VOYAGER_HOME%\examples\java\se-cdc directory. The corresponding c# examples can be found under the the %VOYAGER_HOME%\examples

Java

```
java\examples\timer\Stopwatch1.java

java\examples\timer\Timer1.java
java\examples\timer\SleepyHead.java

java\examples\timer\Timer2.java
java\examples\timer\Timer3.java
```

C#

```
csharp\Common\ExamplesCommon\examples\StopwatchExample.cs

csharp\Common\ExamplesCommon\examples\Timer1Example.cs
csharp\Common\ExamplesCommon\SleepyHead.cs

csharp\Common\ExamplesCommon\examples\Timer2Example.cs
csharp\Common\ExamplesCommon\examples\Timer3Example.cs
```

Running under Eclipse
Running under Visual Studio

Configuration

The examples in this section demonstrate user-customized configuration of Voyager properties.

Configuration1 Example

The Configuration1 example demonstrates configuration of a property in a Voyager server. The server is started on port `8000` with a properties file that changes the Console log level from `SILENT` to `VERBOSE`. The output to this server's console shows the modification.

Configuration2 Example

The `Configuration2` example demonstrates configuration of multiple properties in a custom Voyager application. The thread pool's max idle thread count is set, as is the routing address. The router is configured to send all remote communication through the server on port `8000`. The output in this server's window shows confirmation of the router.

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

Configuration3 Example

The `Configuration3` example demonstrates configuration of multi-property, that is, a property that can take multiple values. The property demonstrated is the URL resource property. Setting these two properties allows Voyager to load classes from the otherwise hidden folders `./hidden1` and `./hidden2`.

Source code location

The examples can be found under the %VOYAGER_HOME%\examples\java\se-cdc directory.

Java

```
java\examples\configuration\configuration1.properties
java\examples\configuration\configuration2.properties
java\examples\configuration\configuration3.properties

java\examples\configuration\Configuration1.java
java\examples\configuration\IDisplay.java
java\examples\configuration\Display.java

java\examples\configuration\Configuration2.java
java\examples\configuration\Configuration3.java

java\examples\configuration\hidden1\Hidden1.java
java\examples\configuration\hidden1\Hidden2.java
```

Running under Eclipse

Managing Connections

The examples in this section demonstrate how to obtain a finer degree of control over Voyager connections using Connection Management Policies. The code for the examples can be found in `examples\connectionmgmt`. `BasicConnectionServer` demonstrates how to use a `BasicConnectionManagementPolicy`to control server connection behavior. `RangeConnectionServer` demonstrates how to use a `RangeConnectionManagementPolicy` to provide connection control.

BasicConnectionServer

In this example, `BasicConnectionServer` creates a `DelayedStockmarket` object and deploys it to a local Voyager server. A `BasicConnectionManagementPolicy` is registered with Voyager, which will permit a maximum of three connections at a time, and will close idle connections after five seconds. When the `ConnectionClient` is started, it will make six simultaneous calls to the `DelayedStockmarket`. The first three calls will get connections, but

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

none of the other calls will execute until one of the first calls completes and its connection is closed. The opening and closing of connections will be seen because a `ConnectionListener` (which is a `TransportListenerAdapter`), is registered with Voyager as well, and will print messages when connections are opened and closed.

RangeConnectionServer

This example is similar to the `BasicConnectionServer` example, except that here the connection restriction is implemented using a `RangeConnectionManagementPolicy`. A `CasePolicy` to restrict the number of server connections to three is associated with a `HostAddressRange` in the `RangeConnectionManagementPolicy` class. When the `ConnectionClient` is started, it will make six simultaneous calls to the `DelayedStockmarket`. The first three calls will get connections, but none of the other calls will execute until one of the first ones completes and its connection is closed. The opening and closing of connections will be seen because a `ConnectionListener` (which is a `TransportListenerAdapter`), is registered with Voyager as well, and will print messages when connections are opened and closed.

Source code location

The examples can be found under the %VOYAGER_HOME%\examples\java\se-cdc directory.

Java

```
java\examples\stockmarket\IStockmarket.java
java\examples\stockmarket\Stockmarket.java
java\examples\connectionmgmt\BasicConnectionServer.java
java\examples\connectionmgmt\ConnectionClient.java
java\examples\connectionmgmt\RangeConnectionServer.java
java\examples\connectionmgmt\ConnectionClient.java
java\examples\connectionmgmt\ConnectionListener.java
```

Running under Eclipse

UDP

The examples in this section demonstrate unicast, broadcast and multicast UDP communication.

Unicast

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

`BasicUdpServer` is an example of a UDP server, which can receive UDP unicast messages.
`BasicUdpClient` illustrates proxies for both a concrete type and an interface. Run `BasicUdpServer` before `BasicUdpClient`.

Broadcast

`BroadcastUdpServer` is an example of a UDP server, which can receive UDP broadcast messages. The `BroadcastUdpServer` can receive messages from either `BroadcastClient` or `BasicUdpClient`.

`BroadcastClient` is an example of a directed UDP broadcast invocation. Illustrates proxies for both a concrete type and an interface. Run `BroadcastUdpServer` before `BroadcastClient`.

Multicast

MessageStreamer

This example illustrates the use of a message streamer (custom data marshalling code). A custom message streamer can help increase remote invocation performance and minimize message size. UDP invocations (which are limited in the number of bytes for an invocation) are a good fit for this facility. Multicast UDP is used for this example in order to simplify execution of the example.

MulticastChatter

This example illustrates a UDP multicast server and client. Multiple `MulticastChatter` processes can run concurrently.

Source code location

The examples can be found under the %`VOYAGER`_HOME%\examples\java\se-cdc directory.

Java

```
java\examples\udp\BasicUdpServer.java
java\examples\udp\BasicUdpClient.java

java\examples\udp\BroadcastUdpServer.java
java\examples\udp\BroadcastClient.java

java\examples\udp\IChat.java
```

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved

```
java\examples\udp\ChatterClient.java
java\examples\udp\MulticastChatter.java

java\examples\udp\INewsListener.java
java\examples\udp\NewsListener.java

java\examples\udp\MessageStreamerExample.java
```

Running under Eclipse

Multihome and IPv6

The examples in this section demonstrate multihome support and bind policies.

IPv6SingleInterface

This example illustrates automatic generation of bind policies when the computer configuration contains a single network interface configured to run both IPv4 and IPv6.

MultihomeAgents1

This example illustrates how to configure TcpTransport for a multihome environment where "multihome" means the host connects to more than one network, typically using multiple network interfaces.

This example requires a command line argument that is an alternate host name or IP address. The command line argument should resolve to the same IP address as the address returned by `InetAddress.getLocalHost()`.

Source code location

The examples can be found under the %`VOYAGER`_HOME%\examples\java\se-cdc directory.

Java

```
java\examples\transport\IPv6SingleInterface.java
java\examples\transport\MultihomeAgents1.java
```

Running under Eclipse

Copyright © 2006-2011 Recursion Software, Inc.
All Rights Reserved