Voyager Core Developer's Guide

.NET Compact Framework

Version 1.0 for Voyager 8.0

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

Table of Contents

Overview....6
Preface....6
  Common Definitions 6
  Voyager Development Requirements 7
  Voyager Installation Directories....7
  Deploying Voyager Applications....8
  Contacting Technical Support 8
Feature Summary....8
    Architectural Flexibility....8
  Voyager Features....8
    Remote-Enabling of Classes....8
    Remote Object Construction....9
    Dynamic Class Loading....9
    Remote Messaging....9
    Remote Exception Handling....9
    Distributed Garbage Collection....9
    Dynamic Aggregation™....9
    SOAP and WSDL Support....10
    Object Mobility....10
    Autonomous Intelligent Mobile Agents....10
    Task Management....10
    Advanced Messaging....10
    Security....10
    Naming Service....11
    Yellow Pages Directory....11
    Multicasting....11
    Publish-Subscribe....11
    Timers....11
    Multi-home Support....11
    TCP Connection Management 11
Core Features....12
Overview....12
Using Interfaces for Distributed Computing....12
Creating or Retrieving a ClientContext....13
Creating or Retrieving a ServerContext....13
Creating a Remote Object....14
Sending Messages and Handling Exceptions....15
Logging Information to the Console 16
Understanding Distributed Garbage Collection....16
  DGC Notification 17
  DGC Discard Delay Configuration 17
Using Naming Services 17
Working with Proxies....18
  Special Methods....19

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

Exporting Objects....20
Working with Federated Directory Services....21
Task and Thread Management....22
Timers....22
Clocking Time Intervals....22
Using Timers and TimerEvents....24
Constructing a Timer....24
Setting a Timer....24
Adding a Listener to a Timer....25
Voyager .NET Compact Framework Basics....26
Starting and Stopping a Voyager Program....27
Type Resolution in .NET CF....28
Understanding Assembly Loading....28
Creating Proxy Classes....29
Creating and Deploying a Voyager Smart Device Application....29
Using Vgen to Generate Interfaces....30
Advanced Features....30
Advanced Messaging....31
Invoking Messages Dynamically....31
Synchronous Messages....31
One-Way Messages....32
Future Messages....33
Retrieving Remote Results by Reference....35
Dynamic Discovery....35
Generic Application Programming Interface....35
Using the Generic API....36
Implementing Dynamic Discovery....37
Using UDP Dynamic Discovery Implementation....37
Using Multicast and Publish/Subscribe....38
Understanding the Space Architecture....38
Understanding the Space Implementation....39
Using TCP Spaces....39
Space Topologies....39
Creating and Populating a Space....40
Nested Spaces....41
Subspace Event Listeners....41
Multicasting....42
Publishing and Subscribing Events....42
Administering a Space....43
Yellow Pages Directory....46
Creating a Yellow Pages Directory....47
Registering a Service....48
Performing a Yellow Pages Lookup....49
Using a Discovery Listener....50
Using UDP as a messaging transport....50
Using custom object streamers....51

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

Voyager Administration........51
Configuration and Management........51
Understanding Voyager Properties........52
Connection Management........53
Understanding Connection Management Policies........53
Understanding Case Policies........54
Maximum Number of Server Connections........54
Maximum Number of Client Connections........54
Maximum Number of Idle Client Connections........54
Client Connection Idle Time........55
Server Connection Idle Time........55
Establishing Case Policies for RangeConnectionManagementPolicy........55
About HostAddressRange........55
Examples........56
Setting the Global CasePolicy........56
Setting Case Policies........56
ServerSocket Policies........57
Adding Custom Sockets to Voyager........57
Appendices........58
Appendix A – Compact Framework Deployment........58
Appendix B – Utilities........58
Overview........58
pgen4csharp........59
pgen4csharp Command Line Options........59
vgen........60

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

<This page intentionally left blank>

Overview

The vision behind Voyager is to make distributed applications easier to design, develop and deploy across multiple operating systems, languages, and devices. Voyager has an extensive set of services and features for distributed application development and deployment, and its APIs are easy to learn and use. 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 C# programming language.

This preface covers the following topics:

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

Common 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 the Microsoft .NET Framework.

CF — *Microsoft .NET Compact Framework.* In reference to running Voyager this term implies a supported version and configuration for the Microsoft .NET Compact Framework.

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

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 Development Requirements

To develop with Voyager, ensure that you have:

- A Java Development Kit (JDK) 1.4.2 or later for Java development. You can download the JDK from java.sun.com free of charge.
- 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.
- Currently, development for the .NET Compact Framework is supported for the PocketPC.

Voyager Installation Directories

The directory structure of Voyager follows:

Directory	Contents
`.\`	Voyager readme, install, changes, environment, copyright, and license text files.
`bin\`	Utilities and other binary files.
`bin\wizard\`	Voyager Wizard application for Java rules-based development.
`doc\`	Developer guides and user guides.
`examples\`	Example files organized by programming language / environment.
`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.
`licenses\`	Licenses for 3rd-party products Voyager uses.
`platform\windows-dotnet\`	.NET API documentation and assemblies.
`platform\windows-mobile\`	Compact Framework API documentation and assemblies.

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

Deploying Voyager Applications

Once you have written a Voyager application selected 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.

Feature Summary

Following is an outline of Voyager features and capabilities. This summary covers all languages/environments; some features may not be available in certain languages/environments:

Architectural Flexibility

Voyager components can be extended or replaced to integrate into a customer's existing computing infrastructure. For example, you can add a new communication protocol to communicate across a proprietary internal network. In addition, Voyager supports multiple distributed architectures including client-server, peer-to-peer, agent-based, pub/sub or message-oriented, or any combination thereof.

Voyager Features

Voyager provides a complete set of features for distributed application development, including the following:

Remote-Enabling of Classes

Java and .NET interfaces can be remote-enabled without being modified in any way, and no specialized additional files are necessary to remote-enable an interface. Thus, there is no difference between a "regular" Java/.NET interface and a remote-enabled interface. Interfaces may also be explicitly remote-enabled by declaring them to implement `recursionsw.voyager.IRemote or com.recursionsw.ve.IRemote`. In Java JSE, JME and Microsoft .NET environments, proxy classes are constructed dynamically at

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

runtime. Microsoft .NET Compact Framework and Java CLDC environments do not support runtime generation of classes; for these environments Voyager provides a proxy generation tool.

Remote Object Construction

You can create a remote instance of any class on any Voyager VM.

Dynamic Class Loading

Voyager allows classes to be loaded at runtime from one or more remote locations. 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 Voyager proxy are transparently forwarded to its object referent. If the object is in a remote VM, the arguments are serialized and sent using the appropriate messaging protocol to the destination, where they are deserialized. The morphology of the arguments is maintained. If an object's class implements `recursionsw.voyager.IRemote (or com.recursionsw.ve.IRemote)` the object is passed by reference. If an object's class implements com.recursionsw.ve.VSerializable (or `java.io.Serializable`), it will be passed by value. Objects that implement none of these interfaces are passed by reference.

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 written to the Voyager logging console.

Distributed Garbage Collection

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™

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

Dynamic aggregation complements the traditional object-oriented 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 source code for the primary object is decoupled from the code for its facets, simplifying your object model.

SOAP and WSDL Support

Voyager provides support for exposing and accessing SOAP services on Java and .NET. 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 between 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. Object mobility is useful for optimizing message traffic in a distributed system.

Autonomous Intelligent Mobile Agents

Voyager supports the creation of mobile, autonomous agents that can be deployed to a VM and execute on arrival (Java and .NET environments). 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 application from being overloaded by threads. User code can leverage this API.

Advanced Messaging

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

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

For Java environments, Voyager provides an enhanced Java Security Manager that supports remote permissions. Remote permissions can be assigned to privileged code to prevent execution by unauthorized clients.

For Java and .NET environments, Voyager provides socket factories 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 (Java and .NET environments) 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 UDP multicast (Java and .NET environments) 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/IP addresses.

TCP Connection Management

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

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

Core 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 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
- Use the federated directory service
- Understand Voyager's task manager to control tasks
- Use the timer and stopwatch utilities

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:

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

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

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

```
IStockmarket market = new Stockmarket();
```

This creates a new instance of the `Stockmarket` class in the local VM.

What about creating and using objects in a remote VM?

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 `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 a 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` 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.

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

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. This creates and returns a proxy to the newly created object.

There are several variations of `create()`, depending on whether the object is to be created locally and 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`, type:

```
String className = "examples.stockmarket.Stockmarket";
```

VoyagerContext voyagerContext = Voyager.startup();

```
// 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. Note that both `market1` and `market2` are declared as type `IStockmarket`. Voyager infers the proxy type based on the instance of the actual object created, in this case `examples.stockmarket.Stockmarket`. Your application code does not reference the proxy type. (If you are curious, you can call `GetType().Name` on a proxy and get its type name.)

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

```
object[] args = new object[] { "NASDAQ", 42 };
```

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

```
IStockmarket market3 =
            (IStockmarket) aFactory.create(className, args);
```

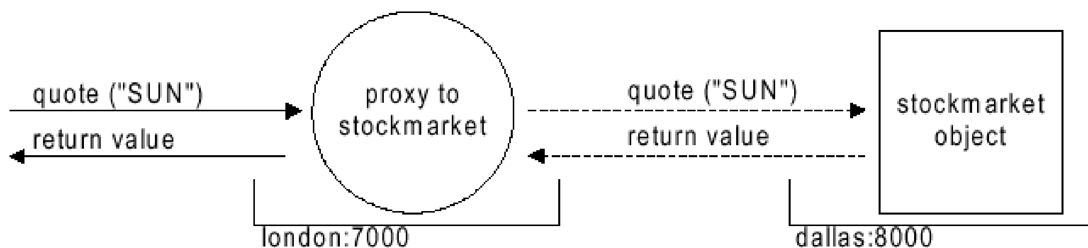
Sending Messages and Handling Exceptions

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 `recursionsw.voyager.IRemote or 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 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.

The Basics1 Example demonstrates basic messaging and remote construction.

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

Logging Information to the Console

The `recursionsw.voyager.lib.util.Console` class allows you to log information, including stack traces of remote exceptions, to the console or a `TextWriter`. Use `Console.enableTopic()` or `Console.addEnabledTopics()` 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

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.

Since most CF environments do not provide a shell or console environment, you may find it useful to redirect Voyager's console output to a file. To do this, create a stream and set it as the output stream for the console. For example:

```
FileStream fs = new FileStream("\\voyagercf_log.txt", FileMode.Create);
recursionsw.voyager.lib.util.Console.LogStream = new StreamWriter(fs);
```

You can then view this log file on the emulator or copy it to the host environment.

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 anchor on the object, permitting the VM's garbage collection mechanism to reclaim the object. The DGC mechanism will also release its anchor on 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 over time.

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

Objects that have been bound in Voyager's naming service are anchored permanently.

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 `recursionsw.voyager.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 `IDGCListener` for more details.

DGC Discard Delay Configuration

DGC reference discard delay configuration support, provided via the `DGC.setDiscardDelay` 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 `recursionsw.voyager.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 `recursionsw.voyager.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 between 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 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:

```
String className = "examples.stockmarket.Stockmarket";
VoyagerContext voyagerContext = Voyager.startup();
ClientContext cc = voyagerContext.acquireClientContext("Dallas");
```

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

```
cc.openEndpoint("//dallas:8000");
aFactory = cc.getFactory();
IStockmarket market = (IStockmarket) aFactory.create(className);
cc.getNamespace().bind("/NASDAQ", market );
```

The construction and binding step may be combined as follows:

```
IStockmarket market = (IStockmarket)
aFactory.create("examples.stockmarket.Stockmarket",
  "//dallas:8000/NASDAQ" );
```

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

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

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 which 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 `recursionsw.voyager.Proxy` and implement the interface(s) of their referent. For Java JSE and .NET environments, Voyager 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). Use any of the following to obtain or create a proxy to an object.

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

Returns a proxy to a newly created remote object, where `classname` is the name of the class that you are creating an instance of.

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

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

Returns a proxy to the object bound to a particular name.

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

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 special methods:

```
GetType()
```

This method is executed directly by the proxy and returns the type of the proxy.

```
GetHashCode()
```

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.

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 `recursionsw.voyager.IRemote, 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, or, for CF, is tagged Serializable`).

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

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 SeverContext 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 )
```

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 port.

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

Working with Federated Directory Services

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 `recursionsw.voyager.directory.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" );
```

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

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

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.

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. In the Java JSE and .NET environments, Voyager uses a custom thread pool. In the .NET CF environment, the task manager uses threads from the standard `System.Threading.ThreadPool` thread pool to run tasks.

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:

- `getDate()`

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

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.

- `getAverageLapTime()`

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

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 (`ThreadPriority.Normal`).

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(
  ThreadPriority.AboveNormal );
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, for example:

```
new TimerListenerThread(listener1, ThreadPriority.Highest)
```

The Timer1 Example demonstrates a ramification of Voyager's default thread behavior, 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.

Voyager .NET Compact Framework Basics

This chapter describes the basic operation and usage of Voyager on the .NET Compact Framework.

In this chapter, you will learn to:

- Start and stop a Voyager program.
- Understand type resolution for the .NET Compact Framework.
- Use the pgen4csharp utility to create proxy classes.
- Create and deploy a Voyager Smart Device application to the Pocket PC emulator.
- Use the vgen utility to generate Java and C# interfaces for interoperability.

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

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.

Both startup methods return VoyagerContext, which is the context used by the application to reference Voyager.

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. In general, you should use the hostname of the system, especially if its IP address may change. 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 partially qualified (`//dallas`) or fully qualified (`//dallas.recursionsw.com`) or `//localhost`. If `//localhost` is specified, Voyager attempts to resolve the system's hostname. 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 system.
`File`, `;argument`, `#reference`	These parts of an URL are rarely used with Voyager, but are presented for completeness.

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 URL on which to listen. If the system is multi-

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

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. Daemon threads 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.

Type Resolution in .NET CF

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 a reference to the `Stockmarket.dll` assembly.

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.

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.

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

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.

Creating Proxy Classes

Voyager uses proxy classes to support invocation of methods on remote objects. A proxy class contains special code to serialize any arguments passed to the method, and sends the serialized arguments and other data to the server using a messaging protocol that specifies an "on-the-wire" format for sending and receiving message invocations and responses. Each proxy class implements one or more application interfaces, allowing the application code to remain ignorant of the proxy class.

The Java and .NET environments support dynamic creation and loading of classes. Voyager takes advantage of this capability by automatically generating proxy classes in these environments. Since the .NET Compact Framework does not support runtime class creation, Voyager provides the `pgen4csharp` utility to create C# source code for proxy classes. The `pgen4csharp` utility is in the `bin\` directory of your Voyager installation.

To generate a proxy class from the `examples.stockmarket.Stockmarket` class, type the following from a command window:

```
% pgen4csharp -la stockmarket.dll examples.stockmarket.Stockmarket
```

(Note: this assumes `pgen4csharp.exe` is in your path.) This generates a file called `Stockmarket_IProxy__Proxy.cs` in the current directory. You will need to include this file in your project. (Right-click the project in Visual Studio, select "Add", then "Existing Item".)

For a complete list of pgenc4csharp options, see pgen4csharp.

Creating and Deploying a Voyager Smart Device Application

To create a Smart Device application in Visual Studio 2008, follow these steps:

1. Select File|New Project.

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

2. Under Project types, Visual C#, select Smart Device. Enter your project name and select OK.

3. For Target platform, select Windows Mobile 5.0 Pocket PC SDK. The .NET Compact Framework version can be either 2.0 or 3.5. Select the appropriate Template (typically either Device Application or Console Application) and select OK.

4. Right-click References and select Add Reference. Navigate to $VOYAGER_HOME/cf/dll and add both Hessianmobileclient.dll and Voyager.CF.dll.

Using Vgen to Generate Interfaces

If you are creating a distributed application using both Java and .NET, you will need common interfaces between the two environments. One way of accomplishing this is to write the interface in one language and then manually port it. A better way is to define the interface in a language-neutral way and then let a tool create both the Java and .NET (C#) interfaces.

The vgen utility generates Java and C# interfaces from an interface definition file. The interface definition file, or IDL, uses the CORBA IDL syntax. You can download the formal IDL definition at http://www.omg.org/cgi-bin/doc?formal/02-06-39.

Here is a sample IDL file for the `examples.stockmarket.IStockmarket` interface:

```
module examples {
  module stockmarket {
    public interface IStockmarket {
      int quote(string symbol);
      int buy(int shares, string symbol);
      int sell(int shares, string symbol);
      void news(string announcement);
    };
  };
};
```

To generate Java and C# interfaces for this IDL, use the vgen command:

```
% vgen istockmarket.idl
```

Your server application will implement the appropriate Java or C# interface. The client application will implement a “do-nothing” version of the interface, and if necessary generate a proxy using `pgen` or `pgen4csharp`.

Advanced Features

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

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 (waits) 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 `IStockmarket`.

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

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 line of code uses `Sync` to dynamically invoke a `buy()` message on an instance of `Stockmarket`.

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

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

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(int,System.String)", new Object[] { 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.

- `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 `recursionsw.voyager.message.TimeoutException` is thrown. See the Future Messages section for information about timeouts. The timeout countdown starts when the `readXXX()` method 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 Sending Messages and Handling Exceptions 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

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

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, from the perspective of the caller. Voyager uses a separate thread to deliver the message. You can send a one-way message dynamically using `recursionsw.voyager.message.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`.

```
ClientContext cc =
Voyager.getDefaultVoyagerContext().acquireClientContext("Serve
r8000");
Result result = cc.getOneWayInvoker().invoke( market, "buy",
newObject[] { 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" } );
```

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

```
// 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 an event listener mechanism. When a return value arrives, `Result` sends `resultReceived()` with a `recursionsw.voyager.message.ResultEvent` object to every `recursionsw.voyager.message.ResultListener` that either was specified in the full version of `Future's 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.

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 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.

- `getTimeout()`

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

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

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 object is large, undesirable network traffic can occur. With Voyager, you can tell `Future` or `Sync` to return a proxy to a result instead, thereby 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 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 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 `ServerContexts`.**
- **`IAnnouncementMarshaller`, the methods implemented by the class that builds `IAnnouncement` instances from the Voyager `ServerContexts`.**
- **`IDiscoveryAnnouncementListener`, the methods implemented by a listener registered with `IDiscoveryService`, and which is notified of each `IAnnouncement` received.**

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

- `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.
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 `ListOfDiscoveredVoyagers()` 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 `ServerContexts` by calling `IDiscoveryService`'s `startAnnouncementSenders()` and stop`AnnouncementSenders()` methods.

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

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 namespace of `recursionsw.voyager.discovery.impl` 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
- `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 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 namespace `recursionsw.voyager.discovery.impl.udp`. The default no-argument constructor builds an instance using the defaults defined in the class as public. 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.

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

```
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 from a Voyager configuration file, or 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

Using Multicast and Publish/Subscribe

Distributed systems often require capabilities 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.
- 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.

Voyager uses a high-performance, highly 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 any `Subspace` in that `Space`. That `Subspace` clones the message to all neighboring `Subspace`s and then delivers it 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.

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

This mechanism allows you to connect `Subspaces` to form arbitrary topologies without the possibility of multiple message delivery.

Understanding the Space Implementation

Three interfaces describe behaviors of `Spaces`. 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`. The class `TcpSubspace` implements `ITcpSubspaceConnections` and communicates using the TCP transport `TcpTransport`.

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. Major factors that influence this include:

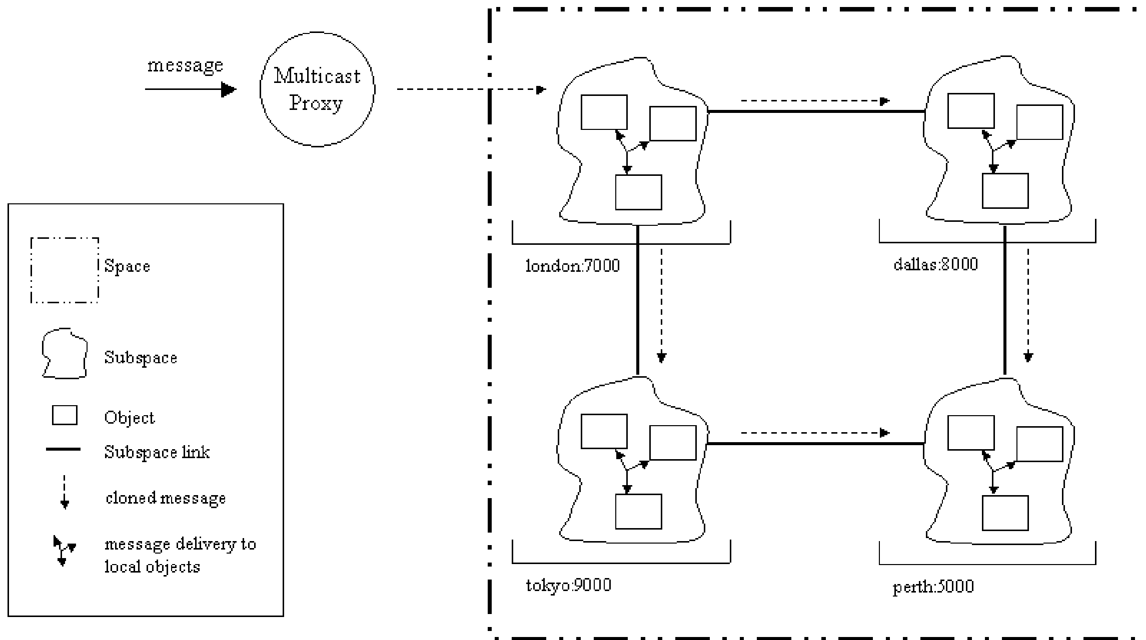
- Where messages or events are generated.
- Network reliability and bandwidth.
- The impact to the application of a `Subspace` becoming unavailable.
- The rate at which events or messages are generated.
- The size of the events or messages published.

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, a server's `TcpSubspace` is connected to each client's `TcpSubspace`, but client `TcpSubspace` are not interconnected. If there are multiple servers, their `TcpSubspaces` are connected. Events or messages are typically created on the server and are efficiently propagated to each client.

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 subspace = new TcpSubspace();
ITcpSubspaceConnections subspace = new TcpSubspace();
```

2. Use the `connect` method to connect the `TcpSubspaces` in a logical `Space`. 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.)

```
subspace1.connect(subspace2);
```

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 and other `TcpSubspaces`, into a `Space`.

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

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

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

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 `recursionsw.voyager.space.ISubspaceMessaging`, `recursionsw.voyager.space.ISubspace`, and `recursionsw.voyager.space.ITcpSubspaceConnections` interfaces for the complete list of features available.

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:

1. `void subspaceEvent( SubspaceEvent event );`

`Subspace` events, defined as constants in the interface `ISubspaceMessaging`, are:

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

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

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

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

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

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

7. 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 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 recursionsw.voyager.space.publish.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.

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

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` 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 VM 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 unless the server `Subspace` is 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.

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

1. ISubspaceMessaging.DIED removes proxies to objects and neighboring Subspaces that have been garbage-collected. A Subspace knows an object is 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 using the specified purge policy.

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 recursionsw.voyager.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)

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

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.

- `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 operative 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 likely 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

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

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 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.

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.

The building block of `Voyager`'s Yellow Pages Directory is the `recursionsw.voyager.yp.YellowPages` class, which implements the interface `recursionsw.voyager.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)`

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

- `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)`
- `void lookup(DiscoveryRequest discoveryRequest, IDiscoveryListener discoveryListener)`

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 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.)

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.GetType().FullName;
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:

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

- `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

Services are registered in a Yellow Pages Directory using a *service description*, implemented in the class `recursionsw.voyager.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 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).

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

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("myAttributeName",
"myAttributeValue"));
```

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 maximum number of matching ServiceDescriptions to be returned:

```
    request.TimeToLive = 50 ;
    request.MinMatches = 4 ;
```

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 YellowPages Example demonstrates the Yellow Pages Directory.

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

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 maximum number of `ServiceDescriptions`, or receiving a response from all Yellow Pages instances in the Yellow Pages Directory.

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
```

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

```
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 `recursionsw.voyager.messageprotocol.vrmp` interface and are registered via `Vrmp.MessageStreamerRegistry.registerStreamer(<Type>, <IStreamer>)`

For a complete example of use, please see `examples.udp.MessageStreamerExample` in the csharp examples under installation directory.

Voyager Administration

Configuration and Management

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

Several of Voyager's internal settings can be modified at runtime using static methods.

In this chapter, you will learn to:

- Understand Voyager runtime properties
- Understand and use Connection Management policies

Understanding Voyager Properties

The following table summarizes Voyager's user-customizable properties. Each property is case sensitive.

Property	Value
recursionsw.voyager.tcp.use_ip_addressing	true | false
console.logLevel	silent | exceptions | verbose
console.enabledTopics	topic1,[topic2,…]
recursionsw.voyager.space.subspaceConnectorMaxQueueSize	int
recursionsw.voyager.space.subspaceConnectorLogging	true | false
recursionsw.voyager.space.rescheduleSubspaceConnector	true | false
recursionsw.voyager.space.subspaceConnectorDeliveryThreadWaitTime	long
recursionsw.voyager.space.enableSubspaceConnectorMonitor	true | false
recursionsw.voyager.space.subspaceConnectorMonitorTimerDelay	long
recursionsw.voyager.space.subspaceConnectorDisableDelay	long
recursionsw.voyager.space.subspaceConnectorDeactivateDelay	long
recursionsw.voyager.space.subspaceConnectorExceptionThreshold	int
recursionsw.voyager.space.enableSubspaceDebugDump	true | false
recursionsw.voyager.space.subspaceDebugDumpPeriodicity	long
recursionsw.voyager.vrmp.useSeparateSerialization	true | false
recursionsw.voyager.vrmp.dgcCycleTime	long
recursionsw.voyager.licenseKeyPath	pathname
recursionsw.voyager.vrmp.enableVrmpLookahead	true | false

- `console.logLevel`

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`.

- `console.enabledTopics`

This property sets the enabled topics for Console logging. The value for this property is a comma-separated list of strings.

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

- `recursionsw.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, and not hostnames. This is required when the remote process might not be able to resolve the local process hostname.

Connection Management

Voyager provides the ability to manage the connections underlying Voyager-to-Voyager communications.

The `recursionsw.voyager.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:

- Creation of client connection to remote URL
- Creation of server connection to remote URL
- Idle of client connection

There is a small 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. 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.

A client connection in Voyager is used when one ORB is initiating an 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

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

Voyager connections are segregated into client connections and server connections. A client connection has a logical association with a Proxy to a remote object, and sends invocation requests. A server connection, logically associated with a local object exported to remote Voyager servers, receives invocation requests.

Voyager provides two default connection management policies: `recursionsw.voyager.transport.impl.tcp.BasicConnectionManagementPolicy` and `recursionsw.voyager.transport.impl.tcp.RangeConnectionManagementPolicy`. `BasicConnectionManagementPolicy` applies a single `CasePolicy` to limit connections to and from all remote VM's. `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 `RangeTcpPolicy` contains a collection of `CasePolicy` objects describing the restrictions on connections between different Voyager VM's 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 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 currently 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.

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

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.

Server Connection Idle Time

A server connection can be given an idle time limit. If a server connection is idle longer than this limit, it will be closed.

Establishing Case Policies for RangeConnectionManagementPolicy

Case policies must be added to a `RangeTcpPolicy` object which is then set as the policy for a given ORB. The easiest way is often to obtain the current policy, modify it, then establish the modified policy as the new ruling policy.

`CasePolicy` objects are managed by three methods on the `RangeTcpPolicy` 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( HostAddressRange range );`

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( HostAddressRange range );`

Removes any established `CasePolicy` for the given range of addresses.

A `GlobalCasePolicy` property also provides access to the global case policy ruling any and all connections for the current Voyager server:

```
RangeTcpPolicy.GlobalCasePolicy = policy;

CasePolicy policy = RangeTcpPolicy.GlobalCasePolicy;
```

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

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

	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.

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 ) );
```

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

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 ) );
```

Sockets are generated by classes that implement the `recursionsw.voyager.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 the `SocketFactory`. Its definition provides only the minimum required for TCP sockets, but can easily be extended for custom implementations.

```
public abstract class SocketPolicy
{
  private long timeout = -1;

  public abstract String ShortName{get};
  public abstract String SocketFactoryClassName{get};

  virtual public int Timeout {
    get { return timeout; }
    set { timeout = value; }
  }
}
```

- `ShortName` returns a String that can be used for convenience naming.
- `SocketFactoryClassName` returns the typename of the `SocketFactory` class used to create sockets governed by this policy type.

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:

```
public ServerSocket createServerSocket( SocketPolicy
  policy, IPAddress bindInterface, int bindPort, int
  backlog );
```

Note: Socket timeouts are not currently used by Voyager.

Adding Custom Sockets to Voyager

Voyager provides the `recursionsw.voyager.transport.impl.tcp.TCPSocketPolicy` and the `recursionsw.voyager.transport.impl.tcp.TCPServerSocketPolicy`

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

classes. These classes implement basic TCP Sockets behavior. You may implement `SocketPolicy` and `ServerSocketPolicy` to provide custom sockets for Voyager connections.

Socket policies are managed by a singleton of the `recursionsw.voyager.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 to register the socket policy. For example, socket policies supporting data compression might be named ZipSocketPolicy and ZipServerSocketPolicy. Assuming these are intended to be used as the default policy for all connections, they can be registered as follows:

```
SocketPolicyManager.Instance.SocketPolicy = new ZipSocketPolicy();
SocketPolicyManager.Instance.ServerSocketPolicy = new
  ZipServerSocketPolicy();
```

Appendices

Appendix A – Compact Framework Deployment

When developing for the .NET Compact Framework, you must include the following assemblies in your project:

Hessianmobileclient.dll
Voyager.CF.dll

You should also include an appropriate Voyager license key file in your deployment. The license key file should be named license.properties and be included in your project. Set the Build Action property to Content and the Copy to Output property to Copy if Newer.

Appendix B – Utilities

Overview

In this chapter, you will learn to:

1. Use the `pgen4csharp` utility to generate the source form of a proxy for a given class.
2. Use the `vgen` utility to create interfaces from IDL that can be used for Voyager programs that require interoperability between Java and .NET CF versions of Voyager.

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

pgen4csharp

The `pgen4csharp` utility generates the C# proxy class source code for a given class.

Example:

```
% pgen4csharp -la stockmarket.dll examples.stockmarket.Stockmarket
```

Generates a proxy class for the type `examples.stockmarket.Stockmarket` in the assembly `stockmarket.dll`.

The `pgen4csharp` utility will use .NET introspection to determine the interfaces each specified type implements. For each specified type, a proxy class will be generated in C# that implements the type's interfaces.

The assembly or assemblies containing the types must be made visible to `pgen4csharp` using the "-la" argument.

pgen4csharp Command Line Options

For a list of the `pgen4csharp` run-time options, run `pgen4csharp` from the command line with no parameters. A description of each option follows.

Argument	Example	Description
`-output <directory>`	`-output gen-proxies`	Specify an output directory for proxy classes (default is current directory)
`-ns <namespace>`	`-ns myapp.proxies`	Specify a namespace for proxy classes (default is namespace of TypeName)
`-la <assembly file>`	`-la myapp.dll`	Load the specified assembly to resolve types
`-loglevel {silent, info, verbose}`	`-loglevel verbose`	Set the logging level
`-c`	`-c`	Generate class-based proxy (internal use)
`-nooverwrite`	`-nooverwrite`	Do not overwrite existing generated proxy classes (default is to overwrite)
`-expandns`	`-expandns`	Generate proxy classes into an expanded directory path based on the namespace
`TypeName`	`example.AClass`	Generate a proxy for the given type

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

vgen

The `vgen` utility will convert an IDL file into Java and C# interfaces that can then be used for interoperability between Java and .NET environments. Note that this utility requires JRE 1.4 or better to operate, as it relies on the Java Voyager implementation for functionality.

Example:

```
% vgen [options] file.idl
```

Options for `vgen` are listed below.

Argument	Example	Description
`-a <file>`	`-a vgen-opts.txt`	Process lines in <file> as arguments
`-d <path>`	`-d vgen-generated`	Store packages relative to <path>
`-I <path>`	`-I idl-include`	Add to list of #include paths
`-p <file>`	`-p prefix.idl`	Process IDL files as if they were prepended with <file>
`-q`	`-q`	Quiet mode
`-v`	`-v`	Verbose mode
`-x`	`-x –Xmx:512M`	Pass remaining arguments to the Java interpreter

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