Voyager Core Developer's Guide

Connected Limited Device Configuration

Version 1.1 for Voyager 8.0

Table of Contents

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 Java programming languages using CLDC 1.1 and MIDP 2.0 interface standards.

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.

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.
CLDC	— *Compact Limited Device Configuration.* This term refers SUNs development platform including a set of programming interfaces and a JVM for resource-constrained devices such as mobile phones, pagers and mainstream personal digital assistants.
MIDP	— *Mobile Information Device Profile.* This term refers to the interface specification for JSR 37 and JSR 118. When combined with CLDC, the MIDP provides a runtime for resource-constrained devices such as mobile phones, pagers and mainstream personal assistants.

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 Sun Java Wireless Toolkit version 2.5.2 or later. This provides a runtime JVM and an emulator for testing. You can download this toolkit from http://java.sun.com/products/sjwtoolkit/download.html free of charge.

Voyager Installation Directories

The directory structure of Voyager follows:

`.\`	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\cf\`	Compact Framework API documentation and assemblies.
`platform\dotNET\`	.NET API documentation and assemblies.
`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.

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. Not all features and capabilities listed in this section necessarily apply to the CLDC platform. This summary covers all languages/environments. Some features may not be available in certain languages/environments.

Voyager Features

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

Remote-Enabling of Classes

Java SE, CDC, and CLDC interfaces can be remote-enabled without being modified in any way. Thus, there is no difference between a "regular" Java interface and a remote-enabled interface. Interfaces may also be explicitly remote-enabled by declaring them to implement `com.recursionsw.ve.IRemote`. In Java JSE, CDC and Microsoft .NET environments, proxy classes are constructed dynamically at 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 remote-enabled class on a remoteVoyager VM. In essence, this is a “distributed new” operator.

Dynamic Class Loading

The Java SE and CDC versions of Voyager allow Java classes to be loaded at runtime from one or more remote locations. The Java CLDC runtime lacks the tools needed to implement runtime class loading, so Voyager for CLDC cannot support dynamic class loading.

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 `com.recursionsw.ve.IRemote` the object is passed by reference. If an object's class implements `com.recursionsw.ve.VSerializable` (or `com.recursionsw.lib.io.ISerializable`), 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.

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.

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

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 SE/CDC 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 SE/CDC 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.

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.

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 );
  }
```

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

Creating or Retrieving a ServerContext

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

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

```
     ServerContext startServer(String url)
```

The URL provided uses the standard format:

```
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. 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.

CLDC has special requirements and restrictions when specifying URLs. The format of a URL used to start Voyager as a server is as follows:

```
protocol://:port
```

Note that the URL used to start Voyager as a server must not specify a host. Doing so to the CLDC platform indicates a request to connect to a client. As an example:

```
protocol://localhost:port
```

indicates to the CLDC platform a request to connect to the client on the localhost at *port* with the given *protocol*.

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(midlet);

// 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 `getClass().getName()` 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:

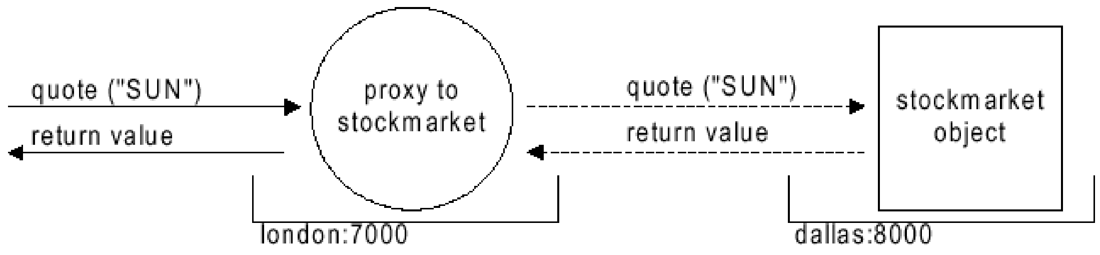
```
Object[] args = new Object[] { "NASDAQ", new Integer( 42 ) };
IStockmarket market3 =
    (IStockmarket) aFactory.create( className, args);
```

Sending Messages and Handling Exceptions

A message sent via a Voyager remote 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`, a proxy to the argument is sent (pass by reference). If the argument implements `com.recursionsw.ve.VSerializable` or `com.recursionsw.lib.io.ISerializable`, 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.

Logging Information to the Console

The `com.recursionsw.ve.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.

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), which 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.

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

NOTE: The Java JSE implementation of Voyager relies on object finalization to automatically perform DGC. Since the CLDC environment does not support the `finalize()` method, you must call `Proxy.dispose()` to release a proxy and notify the local DGC service that the proxy object is no longer in use. Failure to do this may result in excessive memory consumption on the remote (server) virtual machine, as the local Voyager instance’s DGC service will never notify the server of any no-longer-used proxies.

DGC Notification

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

Using Naming Services

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

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

- Voyager federated directory service

The `Namespace` class differentiates among various naming service implementations by using a unique prefix for each implementation. For example, the Voyager federated directory service uses the prefix `vdir:`. Binding and lookup operations use the name's prefix to determine which underlying naming service implementation to access for the operation. Once an object has been bound, it can be looked up by any type of client using any lookup prefix 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(midlet);
ClientContext cc = voyagerContext.acquireClientContext("Dallas");
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)
cc.getFactory().create("examples.stockmarket.Stockmarket",
  "/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, and where `cc` is the `ClientContext`.

```
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 that is installed automatically.

Voyager

Service	Prefix
Voyager federated directory service	`vdir:`

The Naming2 Example illustrates the default naming service.

Working with Proxies

Voyager's proxy classes provide the network communications capabilities to perform remote invocations and work with remote references to objects. All **Voyager** proxy classes extend `com.recursionsw.ve.Proxy` and implement the interface(s) of their referent. 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 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 )
```

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

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

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

Special Methods

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

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

These methods are executed directly by the proxy.

```
hashCode()
```

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

```
equals()
```

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

Additional methods in `Proxy` follow.

```
isLocal()
```

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

```
getLocal()
```

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

```
getClientContext()
```

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

To pass an object by reference, either explicitly pass a proxy obtained using `Proxy.of()`, or implicitly pass a proxy by ensuring that the object class implements `com.recursionsw.ve.IRemote` . (Voyager will also pass a proxy reference if the object does not implement `com.recursionsw.lib.io.ISerializable` or `com.recursionsw.ve.VSerializable`, or, for CF, is tagged with the `[Serializable]` attribute).

CLDC Distributed Garbage Collection and Proxies

The CLDC platform does not support the `finalize()` method, which is utilized by Voyager in the JSE environment to support automatic distributed garbage collection (DGC). Consequently, you must manually manage proxy usage. To notify Voyager you will no longer be using a proxy, you must call the `Proxy.dispose()` method. For convenience there are two versions of this method: a static method that takes an Object to be disposed (which must be a proxy), and an instance method.

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 either (1) the first one created or (2) the one selected by calling the `VoyagerContext.setDefaultServerContext(ServerContext)` method. Note that Voyager never automatically creates a `ServerContext`, and if an implicit export happens before a `ServerContext` is created, the export will fail with an exception.

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

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

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

Exports the object on the `ServerContext`.

```
    unexport( Object object )
```

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

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 `com.recursionsw.ve.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" );
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, CDC, CLDC, and .NET environments, **Voyager** uses a custom thread pool.

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()`

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()`

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.

- `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 `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 Java CLDC Basics

This chapter describes the basic operation and usage of Voyager on the CLDC platform using CLDC 1.1 and MIDP 2.0 interface specifications.

In this chapter, you will learn to:

- Start and stop a Voyager program.
- Understand type resolution for the CLDC platform.
- Use the pgen utility to create proxy classes.
- Create and deploy a Voyager CLDC application to SUN's Wireless Tookit (WTK) emulator.
- Use the vgen utility to generate Java and C# interfaces for interoperability.

Starting and Stopping a Voyager Program

A program must invoke `Voyager.startup()` before it can use any Voyager features:

```
        VoyagerContext vc = Voyager.startup(MIDlet aMidlet)
```

This 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. Voyager will use the specified MIDlet to obtain properties configured in the MIDlet's JAD.

Alternatively, the program can invoke:

```
        VoyagerContext vc = Voyager.startup(Properties props);
```

This method is a useful alternative for CLDC-like platforms such as RIM's BlackBerry that support alternative application classes. (NOTE: The `Properties` class referenced is in the `com.recursionsw.lib.util` package, since the JME platform does not provide the Java `java.util.Properties` class.)

The startup method returns a `VoyagerContext`, which is the context used by the application to reference Voyager.

To shut down Voyager, invoke `voyagerContext.shutdown()`. This method terminates the Voyager internal threads, closes all open connections, and shuts down all ClientContext's and ServerContext's.

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

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 JSE 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 Java CLDC environment does not support runtime class creation, Voyager provides the `pgen` utility to create Java source and/or object code for proxy classes. The `pgen` 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:

```
% pgen4java -m examples.stockmarket.Stockmarket
```

This generates several files in the current directory. You will need to include these files in your CLDC project.

For a complete list of pgen options, see the pgen subsection of Appendix B – Utilities.

Creating and Deploying a Voyager CLDC MIDP Application

When developing for the CLDC environment, you will need to use the `pgen` utility to generate proxy classes. These classes must be compiled into your application. Use the "-m" option to generate the metadata classes required for CLDC.

Interoperability with JSE or .NET/CF Voyager processes requires remote interfaces to declare identical methods with identical signatures in each platform/environment.

Obfuscating CLDC Applications

It is often desirable to use an *obfuscator* to reduce the size of a CLDC obfuscation. Obfuscation tools work by changing the bytecode in the .class files of the application to shorten package, class, method, field and variable names, remove comments and debugging information, and in some cases performing other advanced operations.

When obfuscating a Voyager CLDC application certain rules must be followed in order for remote communications to work correctly. The following classes must *not* be obfuscated:

1. Classes and interfaces involved in remote communication. For example if you have a class `com.foo.remote.Bar` that implements `com.foo.remote.IBar`, neither `Bar` nor `IBar` can be obfuscated. If a method in `IBar` takes or returns a serializable argument (implements `com.recursionsw.lib.io.ISerializable` or similar) the serializable type must not be obfuscated.

2. All proxy classes and metadata classes generated with the pgen tool.

An example configuration file for ProGuard is provided. This contains "-keep" options and other configuration parameters necessary for Voyager. Add your own classes and interfaces to this configuration file as described above.

When using obfuscation on CLDC applications that will communicate with JSE/CDC Voyager processes, it is not necessary to obfuscate the JSE/CDC application.

Advanced Features

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[] { new Integer( 42 ), "SUN" } );
int price = result.readInt();
```

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

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

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

- `isAvailable()`

Returns true if the Result received its return value.

- `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 `com.recursionsw.ve.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 and static method using Voyager's dynamic invocation feature.

One-Way Messages

A one-way message does not return a result. When a caller sends a one-way message, the caller does not block while the message completes, so sending a one-way message is fast, from the perspective of the caller. Voyager uses a separate thread to deliver the message. You can send a one-way message dynamically using `com.recursionsw.ve.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 code uses `OneWay` to dynamically invoke a one-way `buy()` message on an instance of `Stockmarket`.

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

The Message2 Example demonstrates sending a one-way message.

Future Messages

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

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

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

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

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

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

You can be notified when a future return value arrives through an event listener mechanism. When a return value arrives, `Result` sends `resultReceived()` with a `com.recursionsw.ve.message.ResultEvent` object to every `com.recursionsw.ve.message.ResultListener` that either was specified in the full version of `Future.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.

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. It uses UDP multicast packets to advertise and listen for other Voyagers without prior knowledge of their identity or address. While UDP discovery has been implemented on all platforms on which Voyager runs, some platforms, especially Java ME devices, do not support UDP multicast. As a result, dynamic discovery is not supported on Java ME devices.

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

Understanding the Space Implementation

Four interfaces describe behaviors of `Spaces`. 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

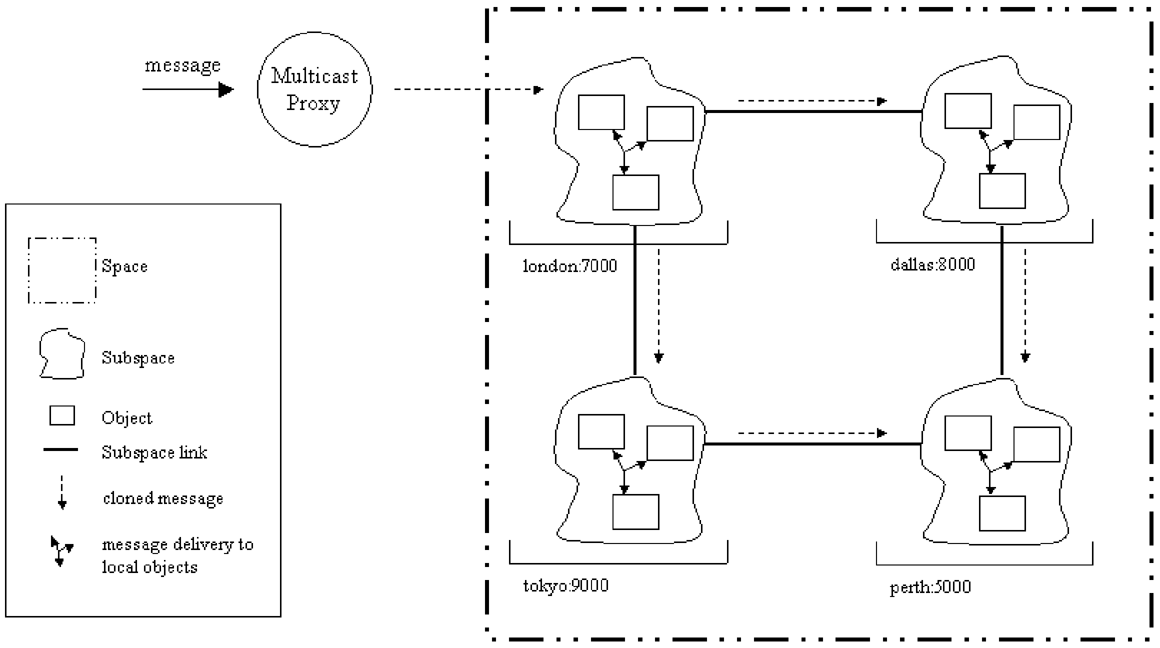
The topology of a `Space` depends on the needs of the application and the environment in which it will run. Major factors that influence this include:

- 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`.



Creating and Populating a Space

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

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

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

2. Use the `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.

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

Nested Spaces

You can nest `Spaces` by adding an (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:

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 `com.recursionsw.ve.space.publishing.Publish.invoke( ISubspace subspace, EventObject event, Topic topic )`. `PublishedEventListener` defines a single method `publishedEvent(EventObject event, Topic topic)` that receives every published event in the `Space`. The listener must handle the event in the appropriate manner.

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

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 `com.recursionsw.ve.space.PropertyHelper` class. Note that changed parameters only apply to newly created `TcpSubspaces`.

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

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

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

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

- `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 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.

Using UDP as a messaging transport

Oneway, asynchronous, unreliable invocations can be made via UDP (unicast, multicast and broadcast). Because of lacking support of UDP multicast and broadcast on Java ME devices, only UDP unicast is discussed. To use UDP as a messaging 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");
```

Voyager Administration

Configuration and Management

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
com.recursionsw.ve.tcp.use_ip_addressing	true | false
console.logLevel	silent | exceptions | verbose
console.enabledTopics	topic1,[topic2,…]
com.recursionsw.ve.space.subspaceConnectorMaxQueueSize	int
com.recursionsw.ve.space.subspaceConnectorLogging	true | false
com.recursionsw.ve.space.rescheduleSubspaceConnector	true | false
com.recursionsw.ve.space.subspaceConnectorDeliveryThreadWaitTime	long
com.recursionsw.ve.space.enableSubspaceConnectorMonitor	true | false
com.recursionsw.ve.space.subspaceConnectorMonitorTimerDelay	long
com.recursionsw.ve.space.subspaceConnectorDisableDelay	long
com.recursionsw.ve.space.subspaceConnectorDeactivateDelay	long
com.recursionsw.ve.space.subspaceConnectorExceptionThreshold	int
com.recursionsw.ve.space.enableSubspaceDebugDump	true | false
com.recursionsw.ve.space.subspaceDebugDumpPeriodicity	long
com.recursionsw.ve.vrmp.useSeparateSerialization	true | false
com.recursionsw.ve.vrmp.dgcCycleTime	long
com.recursionsw.ve.vrmp.enableVrmpLookahead	true | false
com.recursionsw.ve.transport.impl.stream.StreamTransport.client_suffix	string
com.recursionsw.ve.transport.impl.stream.StreamTransport.server_suffix	string
com.recursionsw.ve.taskmanagement.ThreadPool.MaxThreadPoolSize	int

- `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.

- `com.recursionsw.ve.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.

Platform Specific Configuration

Successful operation on some CLDC platforms requires additional configuration unique to that platform. The following paragraphs document the required and optional parameters by platform.

Research In Motion Blackberry

RIM Blackberry devices require suffixes be added to both server and client socket URLs.
For Wi-Fi:
A server URL requires the following string.

```
;deviceside=true;interface=wifi
```

And a client URL requires the following string.

```
;deviceside=true;interface=wifi
```

For 3G:
A server URL requires the following string.

```
;deviceside=true
```

And a client URL requires the following string.

```
;deviceside=true
```

The client and server strings are provided to Voyager by setting the corresponding properties, either com.recursionsw.ve.transport.impl.stream.StreamTransport.client_suffix or com.recursionsw.ve.transport.impl.stream.StreamTransport.server_suffix. If the appropriate string is missing, all attempts to open a connection will result in a timeout failure when javax.microedition.io.ServerSocketConnection.acceptAndOpen()is invoked. Voyager will append the appropriate string to a URL just before the URL is used to create a socket.

The Java virtual machine implemented by Blackberry devices supports a maximum of 16 threads per application, and a maximum of 128 application threads. (See the Blackberry Developer's Knowledge Base article DB-00474, dated June 7, 2006.) Voyager requires a minimum number of threads, usually about 5, before starting threads required to support application requirements. Every connection allocates a thread in which the server socket ServerSocketConnection.acceptAndOpen() executes. A com.recursionsw.ve.space.TcpSubspace requires a thread for each connection to another Space. Because of this limitation, adjusting the maximum size of the Voyager thread pool from the default of 50 to a value of less than 10 is very appropriate. The maximum thread pool size is set using the property name com.recursionsw.ve.taskmanagement.ThreadPool.MaxThreadPoolSize.

Appendices

Appendix A – CLDC Deployment

When developing for the Java CLDC environment, you will need to include `platform/cldc/lib/ve-core-cldc.jar` in your MIDP application jar.

Appendix B – Utilities

Overview

In this chapter, you will learn to use the `pgen` utility to generate the source form of a proxy for a given class.

pgen

The `pgen` utility generates the Java proxy class source code for a given class.

Example:

```
% pgen examples.stockmarket.Stockmarket
```

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

The `pgen` utility will use Java reflection to determine the interfaces each specified type implements. For each specified type, a proxy class will be generated in Java that implements the type's interfaces.

The classes containing the types must be in the CLASSPATH to be made visible to `pgen`.

pgen 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
`-a <file>`	`-a IExample.java`	Specify an interface to be aggregated onto the current file
`-c`	`-c`	Generate class-based instead of interface-based proxies
`-s`	`-s`	Generate source files instead of .class files
`-m`	`-m`	Generate metadata and interface based source files for CLDC
`-d <path>`	`-d src/gen`	Store generated files relative to <path>

`-q`	`-q`	Quiet mode
`-v`	`-v`	Verbose mode
`class...`	`example.MyClass`	Name of class to be processed