Voyager Interoperability Guide

Version 1.1 for Voyager 8.0

Table of Contents

Table of Figures

Introduction

Audience

This document addresses distributed system designers, and implementers. The early sections describe Voyager*'s* features supporting applications whose parts execute in diverse runtime environments, frequently on heterogeneous hardware. The later sections discuss interoperability of particular languages, runtime environments, and platforms.

Prerequisites

Voyager Interoperability Guide assumes a basic working knowledge of Java and/or .NET and Voyager, as well as experience developing network applications spanning multiple runtime environments.

Overview

Elements of an application operating in differing runtime environments connected by a network encounter numerous issues. The first few sections discuss the data types common to Java (SE, CDC, CLDC, and Android) and C#. The following sections discuss how Voyager serializes the data types for transfer over the network for each of the supported runtime environments, including tuning Voyager for good performance.

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 via the email or phone at psupport@recursionsw.com or by calling (972) 731-8800.

The Distributed Data Model

Successful exchange of messages among programs executing on differing platform architectures, runtimes, and written in differing languages requires all elements of the distributed application to speak a common language. Voyager solves this problem by defining a subset of Java, C#, and VB data types that Voyager is able to move over the network and represent in all environments. The two parts of the data model are the primitive types and the structured types.

Primitive Data Types

The primitive data types include integral numbers, floating point numbers, boolean, character, character string, and single and jagged arrays of these types.

Structured Types

The structured types include the following.

- classes whose fields are themselves primitive or serializable class types. Classes that are serialized among environments must have field names that match exactly in name (case included) in each codebase.

- maps, i.e., a collection of key-value pairs where each key is associated with a value.

- collections, i.e., a variable length sequence of objects.

Type Interoperability

The supported interoperable types are shown in the following table.

Type Description	Java SE, CDC, & Android	Java CLDC	C#
A value of a reference type that points to nothing.	null	null	null
True or False in 1 bit	java.lang.Boolean,	java.lang.Boolean,	bool

Type Description	Java SE, CDC, & Android	Java CLDC	C#
	boolean	boolean	
8 bit integral value	java.lang.Byte, byte (signed)	java.lang.Byte, byte (signed)	byte (unsigned)
16 bit signed integral value	java.lang.Short, short	java.lang.Short, short	short
32 bit signed integral value	java.lang.Integer, int	java.lang.Integer, int	int
64 bit signed integral value	java.lang.Long, long	java.lang.Long, long	long
IEEE floating point number in 64 bits	java.lang.Float, float	java.lang.Float, float	float
IEEE floating point number in 128 bits	java.lang.Double, double	java.lang.Double, double	double
Unicode character in 16 bits	java.lang.Character, char	java.lang.Character, char	char
Sequence of Unicode characters	java.lang.String	java.lang.String	string or System.String
A map of key object to value object pairs	java.util.HashMap, java.util.Hashtable	java.util.Hashtable	System.Collections.Hashtable
A variable length, single	java.util.ArrayList,	java.util.Vector	System.Colle

Type Description	Java SE, CDC, & Android	Java CLDC	C#
dimension, ordered list of Object	java.util.Vector		ctions.ArrayList
A calendar date	java.util.Date	java.util.Date	DateTime
A “jagged” array of any of the supported types.	[]	[]	[]
Any serializable object	java.io.Serializable, com.recursionsw.ve.VSerializable, com.recursionsw.lib.io.ISerializable, or com.recursionsw.lib.io.ISerializationSurrogate implementor	com.recursionsw.veVSerializable, com.recursionsw.lib.io.ISerializable, or com.recursionsw.lib.io.ISerializationSurrogate implementor,	Object tagged with [Serializable]
Voyager proxies	com.recursionsw.ve.Proxy	com.recursionsw.ve.Proxy	recursionsw.voyager.Proxy

Figure 1 Type Compatibility for Interoperability

Serialization by Language and Platform

Distributed applications typically need to share complex application objects. Passing a complex object to another (remote) process is termed *serialization*, and receiving a complex object from a (remote) process is termed *deserialization*. The classes (types) that are passed back and forth are often termed *data transfer objects*, or DTO's. Data transfer objects are typically defined to contain fields (usually but not always consisting of primitive types), but implement no business logic. They often are also marked with special code that is used for serialization and deserialization.

Voyager implements two versions of object serialization for Java SE 1.4, CDC, and Android, and a single version of object serialization for all other environments. Native Java object serialization is supported for the JSE, CDC, and Android environments. Hessian serialization is also supported on these platforms and is the only serialization mechanism supported in CLDC and C#. When developing a cross-platform application that will run in either a CLDC or C# environment, Hessian serialization is required.

Serialization for Java SE, CDC, and Android

Voyager running in a Java SE, CDC, or Android environment defaults to serialization based on the Hessian 2 specification. If a class requires custom serialization, the custom serialization should be implemented using the Hessian mechanisms. A configuration option allows selection of Java serialization, as show below.

Serialization for Java CLDC

Voyager running in a Java CLDC environment implements serialization based on the Hessian 2 specification. Because the Java CLDC language omits support for introspection, there is no native implementation of serialization. Two features of Voyager for Java CLDC enable remote invocation of methods on objects in this environment: generation of static proxy classes and an associated metadata class (see the discussion of generating source proxies in the Voyager Core Developer's Guide); and custom serialization for objects that move to and from the Java CLDC environment. Voyager supports two implementation patterns for the custom serializers. The first requires the class to be serialized to implement the `com.recursionsw.lib.io.ISerializable` interface. The second requires the class to be serialized implement the `com.recursionsw.ve.VSerializable` interface declaring no methods (a tag interface), and creation of a separate class that implements the `com.recursionsw.lib.io.ISerializationSurrogate` interface.

The following mechanisms for custom serialization also operate in the Java SE and Java CDC platforms, allowing a single Java source file to be used for all three runtime environments.

Custom serialization in the serializable class

A class containing its own custom serialization must implement the `com.recursionsw.lib.io.ISerializable` interface, and it must declare a public constructor with no arguments. The source file for such a class can be compiled and used in all the Java-based environments: Java SE, CDC, CLDC, and Android. That is, a single source code tree can be used for Voyager serializable objects running in Java CLDC, Voyager running in Java SE, Voyager running in Java for Android, and Voyager running in Java CDC.

The `ISerializable` interface declares three methods, detailed in the following paragraphs.

Looking the `examples.space.Consumer` class, found in the Space example, changing it from a proxied class to a serializable class requires implementing the `ISerializable` interface and implementing the `ISerializable` interface's methods.

```
public class Consumer implements IConsumer, ISerializable
{
        private String name;
        private Vector output = new Vector();
        // …
}
```

Figure 2 Consumer class attributes

The `writeClassDefinition()` method, which has a single parameter of type `ClassDefinitionBuilder`, provides to the serialization code the class's attribute names and types. The order in which the attributes are added to `ClassDefinitionBuilder` defines the sequence in which the other two methods read or write the attribute values. Serializing the `Consumer` class requires serializing the `name` and `output` attributes. The String provided for the field name must exactly match the field name in the code. As a matter of good practice the attributes should be added in alphabetical order.

```
Public void writeClassDefinition(ClassDefinitionBuilder builder)
{
      builder.addField("name", String.class);
      builder.addField("output", Vector.class);
}
```

Figure 3 Consumer class `writeClassDefinition()`

The `writeClassDefinition()` method is called by creating a temporary instance of the class using the default constructor, and the `ClassDefinitionBuilder` returned is cached.

The `writeObject()` method is invoked to serialize an instance. The parameter, an `IObjectOutput`, implements methods to serialize the supported types, e.g., `writeString()`, `writeInt()`, `writeObject()`, etc. As stated previously, the attributes must be written to the object output in the same order they are added in `writeClassDefinition()`.

```
public void writeObject(IObjectOutput out) throws IOException
{
      out.writeString(name);
      out.writeObject(Object);
}
```

Figure 4 Consumer class `writeObject()`

As the inverse of `writeObject()`, the `newInstance()` method is invoked to deserialize an instance. The parameter, an `IObjectInput`, implements methods to deserialize the supported types, e.g., `readString()`, `readInt()`, `readObject()`, etc. The attributes must be read in the same order as they are added in `writeClassDefinition()`.

```
public Object newInstance(IObjectInput in) throws IOException
{
      Consumer aConsumer = new Consumer();
      aConsumer.name = in.readString();
      aConsumer.output = in.readObject();
      return aConsumer;
}
```

Figure 5 Consumer class `newInstance()`

If a class contains final attributes that need to be serialized, the `newInstance()` method can be implemented in terms of a constructor. This pattern is also *strongly* recommended for class hierarchies.

```
public Consumer(IObjectInput in) throws IOException
{
      name = in.readString();
      output = in.readObject();
}

public Object newInstance (IObjectInput in) throws IOException
{
      return new Consumer(in);
}
```

Figure 6 Consumer class `newInstance()` with helper constructor

Custom serialization in a separate class

Implementing custom serialization using a class external to the class being serialized requires that the serializable class implement `VSerializable` and have a default constructor. The class doing the serialization, referred to as the surrogate class, must implement `com.recursionsw.ve.ISerializationSurrogate`, and the surrogate class name must be the name of the serializable class with "`Serialization`" appended. For example, if the name of the serializable class is `com.foo.Joe`, then the name of the surrogate class must be `com.foo.JoeSerialization`. The methods declared in `ISerializationSurrogate` perform the same operations as the methods declared in `ISerializable`. The `ISerializationSurrogate` method `writeClassDefinition()` is identical to `Iserializable`'s `writeClassDefinition()` method. The `ISerializationSurrogate` method `readObject()` is equivalent to `ISerializable`'s `newInstance()` method, but adds the serializable object as an additional parameter. The `ISerializationSurrogate` method `writeObject()` is equivalent to `Iserializable`'s `writeObject()` method, but adds the serializable object as an additional parameter. The rule that all three methods must deal with the attributes in the same sequence applies to `ISerializationSurrogate`'s methods.

Serialization for C#

Voyager running in C# .Net environment implements serialization based on the Hessian 2 specification. The developer need only tag the class with `System.SerializableAttribute` (`[Serializable]`), which uses the default serialization semantics. Fields that should not be serialized may be tagged with `[NonSerialized]`. `IDeserializationCallback` is also supported.

Configuring Voyager Serialization

Voyager running in a Java SE, CDC, or Android environment offers configuration options related to serialization. None of the other environments offer or require configuration of the serialization component.

The default serialization implementation can be changed using any of the following configuration mechanisms. If the Voyager being started will exchange messages with Voyager CLDC or Voyager C#, changing the default will improve performance.

- Setting the Voyager property `com.recursionsw.ve.messageprotocol.vrmp.serializationdefault` to either

"`hessian`" or "`java`". The comparison is insensitive to letter case, i.e., "`Hessian`" or "`HESSIAN`" also work. This approach is appropriate when starting Voyager from a command line or IDE launch configuration.

- Adding any of the following property settings in the Voyager property file read at startup time. This approach works well when starting up Voyager as a server using the provided startup script file.

1. `com.recursionsw.ve.messageprotocol.vrmp.VrmpSerialization.selectHessianDefaultSerialization`
2. `com.recursionsw.ve.messageprotocol.vrmp.VrmpSerialization.selectHessianDefaultSerialization=true`
3. `com.recursionsw.ve.messageprotocol.vrmp.VrmpSerialization.selectJavaDefaultSerialization=false`

Invoking any of the following methods.

1. `com.recursionsw.ve.messageprotocol.vrmp.VrmpSerialization.selectHessianDefaultSerialization()`
2. `com.recursionsw.ve.messageprotocol.vrmp.VrmpSerialization.selectHessianDefaultSerialization(true)`
3. `com.recursionsw.ve.messageprotocol.vrmp.VrmpSerialization.selectJavaDefaultSerialization(false)`

Serialization Differences

The Hessian serialization differs in the following ways from the default Java serialization.

Primitive Wrapper Classes

Java serialization of wrapper classes for primitive types, e.g., `Integer`, `Boolean`, `Float`, etc., retains reference equality after deserialization, while Hessian serialization always deserializes the wrapper classes for primitive types into unique instances. As an example, consider the following code sequence.

```
public class Test {
 /** A method **/
 public boolean dupTypes( Integer p1, Integer p2 )
 {
  return p1 == p2;
 }
}
```

```
public class Invoker {
void caller()
 {
  Test t = (Test)Factory.create("Test", "//remotehost:8000" );
  Integer i = new Integer(0);
  t.dupTypes( i, i );
 }
}
```

The code in `Invoker's caller()` method constructs a remote instance of the `Test` class, then invokes the `dupTypes()` method. If the invocation is serialized using Java's default serialization the `dupTypes()` call will return `true,` but if Hessian serialization is used `dupTypes()` will return `false.` This is the case for the following classes.

- `java.math.BigDecimal`
- `java.lang.Boolean`
- `java.lang.Byte`
- `java.lang.Character`
- `java.util.Date`
- `java.lang.Double`
- `java.lang.Float`
- `java.lang.Integer`
- `java.lang.Long`
- `java.lang.Short`
- `java.lang.String`

Summary

Serializable classes moving to or from a Java CLDC environment require implementation of either one of two custom serialization mechanisms. In all other Voyager runtime environments the platform's native serialization suffices for most serializable classes. Figure 1 Type Compatibility for Interoperability lists the standard types supported by Voyager when a serializable type's instances will move among different runtime environments. Finally, the Voyager configuration property relevant to interoperability and serialization was discussed, including how and when the default configuration should be changed.