Comp 734 - Assignment 3: Extendible and Multi-Platform Object (De)Serialization in GIPC

Date Assigned: October 8, 2015

Part 1 Completion Date: Oct 8, 2015

Part 2 Target Date: Tue Oct 27, 2015

Part 2 and 3 Completion Date: Tue Nov 2, 2015

The assignment is divided into three parts to make it easier. In the first part, you will simply convert your previous assignment to GIPC. This part has already been done by some of you.

In the second part, you will build an extensible (de) serialization mechanism, which requires you to set up a complex recursive scheme involving objects of different classes.

The third part will allow non-tree data structures to be serialized and de-serialized. In addition, it will support a reflection-based scheme that is more flexible than Java's and can support heterogeneous-platforms. Only the reflection-based scheme requires you to learn material – the remaining components can be done on your own.

Part 1: RMI → GIPC

Port your RMI system of the previous assignment to GIPC duplex RPC. In converting to GIPC, none of your remote objects or proxies should change, ideally. All you should be doing is writing client and server launchers. Look in GIPC at the package inputport.rpc.duplex.counter.example for how a distributed counter is implemented in GIPC. Copy this package into your project and ensure it works. Now use it as a model for the port of your system by changing the code that registers objects and generates proxies.

Compare the timings you get for the 4-person synchronous session with RMI and GIPC and submit a document containing these timings. You do not have to submit a new video as I assume nothing will change in the UI.

You may find that the number of outstanding writes allowed by GIPC by default is small. This value used to be a constant. In the version you have, you can call the method:

public static void setMAX_OUTSTANDING_WRITES(**int** mAX_OUTSTANDING_WRITES) in the class AScatterGatherSelectionManager to set this number. Make it as small as you need as Java gave me a memory overflow error when it was in thousands.

As we saw in class, proxies for remote objects do not involve communication with a server. This means they can be generated before the client port is connected. Connection is required,

however, before a method is invoked on the proxy. You know the port is connected when the createUI() or doPostConnectAsyncOperations() method is called in the launcher.

You can pass remote callback objects in GIPC-RMI as you would do in RMI. Pass the remote objects directly, do not try and create proxies for them, as GIPC automatically creates the proxies.

Part 2: Serializer Interface, Factory and Interface

In this part you will define a class (that can use additional classes you create) that implements the following GIPC interface:

```
package serialization;
import java.nio.ByteBuffer;
public interface Serializer {
      public ByteBuffer  outputBufferFromObject(Object object);
      public Object objectFromInputBuffer(ByteBuffer inputBuffer) ;
}
```

The first method serializes an object into a byte buffer, and the second does the reverse. As you have seen, a bytebuffer is a wrapper around an array, with markers indicating the position, limit and capacity. Both methods should access these markers in an appropriate fashion. In particular, the second method should not assume that the first byte to be read is the first byte in the array. This means you might have to make a flip() call after writing to the buffer.

To minimize copying and data structure allocation, ideally a serializer should use a single byte buffer for all serializations. This means that in each recursive step, the same byte buffer should be used. Also even two independent calls to `outputBufferFromObject` should return the same byte buffer. This makes your implementation more difficult – do this optimization as the last step and in your document indicate if you make it.

You should also define a factory that instantiates your class and implements the following interface:

```
package serialization;
public interface SerializerFactory {
      Serializer createSerializer ();
}
```

Your serializer will be used by GIPC when you register an instance of this factory by calling the setter method of `serialization. SerializerSelector`

You will not be able to serialize (and deserialize) arbitrary objects. The objects you can handle will be called *serializable* objects.

For part 2, serializable objects include the following base objects:

Integer, Double, String, null, Boolean

In other words, they include the **null** value and the Integer, Double, Boolean and String objects.

In addition to these base types, serializable objects, include the following composite objects:

> java.util.Collection whose elements are (base or composite) serializable objects
> java.util.Map whose key and values are (base or composite) serializable objects

As Java automatically converts between primitive and wrapper values, you will also be able to handle **int** and **double** primitive values.

To serialize a non-null serializable object of class C, you will need to send the name of the class in a byte sequence followed by a byte sequence representing the object value. Thus the full serialization of an object consists of its class name serialization and its value serialization. To de-serialize an object you will convert the class name to a class and then instantiate the class based on the value serialization. For a null value you will need some variation of this scheme.

The method getClass() can be invoked on any object to get the class of an object, and the method getName() can be invoked on a class to get its name. The static method Class.forName(String) can be used to convert a class name to a class, and the method newInstance() can be invoked on a class to create a new instance of the class. The instantiated class must have a null constructor, which will be assumed for classes of all composite objects. ByteBuffer provides putInt(), getInt(), putDouble(), getDouble() for encoding and decoding value serializations of base integer and double values, and we have seen methods for doing the same for String values.

Part 3 will break up the class you wrote above into multiple classes, so if you feel confident, you can use the scheme given in Part 3 to implement the functionality above.

Part 3: Extendible Full Serializer and External Value Serializers

To send and receive the class name and null value, you can hardwire some scheme for serializing and de-serializing these values. For other values, you must allow external serializers to be registered, which encode and decode value serializations of these objects. You must write a full serializer that calls (and is called by) the registered value serializers.

The exact form of the registration scheme is up to you. It should allow the programmer to associate a class with a serializer object that creates a value serialization of the class. Multiple classes can be associated with an instance of the same serializer class or even the same instance.

In addition, you must register value serializers for the three base classes (String, Integer, Double, Boolean), three implementations of java.util.Colllection (java.util.HashSet, java.util.ArrayList, and java.util.Vector) two implementations of java.util.Map (java.util.Hashtable, java.util.Hashmap). To illustrate, you must support equivalents of the following registrations supported by my implementation:

```
ACustomSerializer.registerSerializer(Integer.class,
                                     new AnIntegerSerializer());
ACustomSerializer.registerSerializer(Boolean.class,
                                     new ABooleanSerializer());
```

```
ACustomSerializer.registerSerializer(Double.class,
                                     new ADoubleSerializer());
ACustomSerializer.registerSerializer(String.class,
                                     new AStringSerializer());
ACustomSerializer.registerSerializer(HashSet.class,
                                     new ACollectionSerializer());
ACustomSerializer.registerSerializer(ArrayList.class,
                                     new ACollectionSerializer());
ACustomSerializer.registerSerializer(Vector.class,
                                     new ACollectionSerializer());
ACustomSerializer.registerSerializer(HashMap.class,
                                     new AMapSerializer());
ACustomSerializer.registerSerializer(Hashtable.class,
                                     new AMapSerializer());
ACustomSerializer.registerSerializer(Hashtable.class,
                                     new AMapSerializer());
```

Alternate Serialization

Allow an object implementing an interface to be de-serialized as an instance of another class that implements the same interface. For instance, allow an ArrayList to be deserialized as a Vector. The alternate class is registered by giving the classes of the serialized and deserialized objects, as in:

ACustomSerializer.registerDeserializingClass(ArrayList.class, Vector.class)

Non Tree Data Structures

Using the composite types, it is possible to create structures that are general graphs rather than trees. In a tree, a node has a single parent, while a graph allows multiple parents. This means that in a graph, the same node may be visited multiple times in a descent through the serialized/de-serialized structure. The de-serialized object should be isomorphic to the serialized object. This means that the serialization should contain a reference rather than a value when a component is visited again, and this reference should be converted to a memory address at the destination. If you ensure that the serialization and deserialization traversals serialize and de-serialize corresponding objects in the same order, this scheme can be easily and efficiently implemented.

Arrays, List Patterns, and Beans

The scheme, as described above, does not extend to classes for which serializers have not been registered, and for a serializer to handle types about which it does not know. Extend this scheme to support serialization of enums, arrays, and objects that follow the Bean and list patterns described in class. We will refer to these serializers as type-independent serializers.

You must define three new registration methods to handle these three kinds of objects, as illustrated below:

```
ACustomSerializer.registerenumSerializer(new AnenumSerializer());
ACustomSerializer.registerArraySerializer(new AnArraySerializer());
ACustomSerializer.registerBeanSerializer(new ABeanSerializer());
ACustomSerializer.registerListPatternSerializer(
      new AListPatternSerializer());
```

In addition, you should implement these four serializers.

It is possible for an object to be serialized both by a type-dependent serializer and a type-dependent one.

Your serializers should allow only instances of java.io.Serializer to be serialized. Encountering some other kind of object should throw the exception `java.io.NotSerializableException`. You should make an exception for null, which is not a Serializable. Thus, you should allow null to be sent.

To handle enums, you can use: class.isEnum() to determine if a value of type class is an enum class. Also given an enum class object, you can call, e.getEnumConstants() to get the enum constants. The toString() method of these constants gives the String representation of each constant. You can also cast the enum class object, e, to Enum and execute Enum.valueOf(e,enumStringRepresentation) to get the enum object associated with enumStringRepresentation.

You can call the isArray() method on the class of an object to determine if it is an array. You can call Array.newInstance(Class componentClass, int size) to instantiate a new array. Array.length(`Object array`), Array.`get(Object array, int index)`and Array.`set(Object array, int index, Object value)` can be used to read and write and elements of instantiated arrays.

To handle beans, you should use the java.beans.BeanInfo class and invoke the invoke the getPropertyDescriptors() method in it to get each property and its read and write method. It is possible to get the BeanInfo of a class, invoke the method Introspector.getBeanInfo(class) method.

To support list pattern, use the oe class util.misc.RemoteReflectionUtility. The methods isList(), listGet(), listAdd(), and listSize() methods.

Alternate serialization along with type-independent serializers makes your scheme multi - platform.

Do not die when bean get read/write methods misbehave, but to just skip them in serialization. Do print the error, if any. A missing write method (e.g. for the class property) s not an error such a property is inherently transient.

Transients and InitSerialized Objects

You should not serialize objects whose read methods are associated with the util.misc.Transient annotation (not be confused the Java beans annotation with the same name), which is checked by invoking RemoteReflectionUtility.isTransient(Method) call. After an object is serialized, invoke the method initSerializedObject() in it, if such a method exists, which can be done by invoking RemoteReflectionUtility. invokeInitSerializedObject(Object)

Testing and Writeup

The GIPC class serialization.examples.SerializationTester includes a main method that has test cases with a comment delineating the two parts. Execute this in debug mode because in run mode it throws exceptions. (The new version of GIPC does not have this problem)
After performing this test, modify your part 1 to use your serializer and compare the performance of 4-person synchronous GIPC-implemented session using the GIPC serializer and yours.

For part 2, *describe the interface of registered serializers.* For part 3, extend the explanation to describe how you handle graph data structures. Be sure to describe any modifications to the interface of the registered serializers. Use the tracer to show the important various steps of your algorithm. To print a tracing string, s, rather than saying System.out.println(s), say Tracer.info (this, s);

You will have to modify your launchers to (a) enable tracing and (b) set the serializer factory. It will be best to override the inherited launcher method:

```
protected void initPortLaucherSupports()
```

to perform these steps. This method sets the default tracing and factories. This method should call the inherited method and then execute your (tracing and factory) overriding code. Your tracing code should consist of the statements:
Tracer.showInfo(true);
Tracer.setKeywordPrintStatus((C1, true);
…
Tracer.setKeywordPrintStatus((CN, true);

Where C1..CN are the classes whose Trace.info() calls you want traced.
In the client (server), the GIPC class you want traced is: AGenericDuplexBufferClientInputPort.class (AGenericDuplexBufferServerInputPort.class). In both cases, you want the statements in your serializer class traced.

Tracer.setKeywordPrintStatus(AGenericDuplexBufferServerInputPort.class, true);

Submission

(a) Create a YouTube video demonstrating your code running serialization.examples.SerializationTester . Show traces and debugger variable values to convince me that the method runs correctly with your code.

(b) Create a Youtube video demonstrating 4-user GIPC synchronous case of Part 1 in which your serializer is used instead of the GIPC serializer. In this video, convince me through traces and/or the debugger that it is your serializer that is running.

(c) Submit a handout in which you (i) submit the traces produced by (a) and (b); (ii) compare the timings of the two serializer as indicated above; and (iii) explain the workings of parts 2 and 3 as indicated above.

(d) Some of you will modify your code in response to what you learnt in class and some you will not as I suggested. Even though I recommended to not fix something that is not broken, I do want to reward those of you who do change. So in your write up, indicate how you handled *all* of the issues we discussed in class (including logical vs physical) and also indicate how you would have liked to have handled them. If you are handling physical structures, it should be in addition to logical structures and not in place of them.

(e) Submit your code on Sakai.

.