Performance Evaluation of Object Serialization Libraries in XML, JSON and Binary Formats

Kazuaki Maeda
Department of Business Administration and
Information Science, Chubu University
Kasugai, Aichi 487–8501, Japan
Email: kaz@acm.org

***Abstract*—This paper compares twelve libraries of object serialization from qualitative and quantitative aspects. Those are object serialization in XML, JSON and binary formats. Using each library, a common example is serialized to a file. The size of the serialized file and the processing time are measured during the execution to compare all object serialization libraries. Some libraries show the performance penalty. But it is clear that there is no best solution. Each library makes good in the context it was developed.**

***Keywords*—object serialization, performance evaluation, XML, JSON**

I. Introduction

During the last two decades, many applications based on modern computing technologies has been moving to distributed systems. In the book[1], they defined a distributed system as the following:

> *A distributed system consists of a collection of autonomous computers linked by a computer network and equipped with distributed system software.*

We are already familiar with distributed systems such as automatic teller machines, airline reservation systems and online shopping in everyday life. Software developers heavily use E-mail, remote login, network file systems and remote file transfer. All systems are based on distributed systems.

Many distributed systems have been used the message exchange between computers to implement the distributed systems. From programmers' point of view, a remote procedure call (RPC) provides powerful capability[2]. It provides an execution of a procedure in another computers without programmer explicit coding the details. The detailed interactions between computers are hidden under the hood to improve programmers productivity.

Java remote method invocation (RMI) supports object-oriented programming interface in Java equivalent of RPC. It provides facility to invoke a method of an object existing in a Java VM from another object existing in another Java VM. Java RMI has mainly two advantages. Firstly, a method invocation of a remote object is nearly the same as one of a local object. Secondly, RMI allows Java developers to declare the methods that are available for the remote invocation using a normal Java interface. These advantages simplifies the task to develop distributed systems in Java.

When the method invocation is written to a network, Java RMI writes the specified parameter objects and the objects reachable from them into bytes. It is called serialization. Java RMI sends the resulting bytes onto the TCP/IP socket. After the bytes are sent to the receiver object on a remote machine, the receiver object reads them and reconstruct the parameter objects from the bytes. It is called deserialization. During RMI, serialization and deserialization are automatically executed. so that the implementation details are hidden. Developers can develop programs without considering about object serialization.

The value of each field in the Java objects is saved using object serialization in a binary format so that we usually believe that the size of serialized bytes is small. However, the size is not small in contrast with other serialization formats under the author's experiences. Therefore, the author decides an experiment to investigate the size of serialized data. Moreover, there is another disadvantage. The serialized data cannot be used in other programming languages due to the lack of libraries for such languages that can read this type of data. The author believes that multiple programming language support is desired in heterogeneous Internet environments.

XML is currently used as a standard language for data representation in wide application area. It can be used different platforms. XML is an advantage in heterogeneous Internet environments. However, when data is encoded in XML, the result is typically larger in size than an equivalent encoding in another format because of XML's many redundant tags.

Another disadvantage of XML is that it is difficult to read and understand them. To overcome the disadvantage, JSON is currently becoming a popular data representation[3] JSON is represented by object notation of JavaScript to simplify the representation. When data is encoded in JSON, the result is typically smaller in size than an equivalent encoding in XML.

This paper describes comparison of twelve object serialization libraries in XML, JSON and binary formats. A common example is chosen and it is serialized to a file using each library in a supported format. The execution time are measured in serializing and deserializing. After serializing, the size of serialized files are measured.

The paper is organized as follows: Section II describes briefly twelve libraries of object serialization using a sample program. Section III explains an experiment and compares the performance. Section IV summarizes this paper.

978-1-4673-0734-5/12/$31.00 ©2012 IEEE

```
public class Order {
  CupSize size;
  String  product;
  String  date;
  double price;
  int  count;
  List<Option> options;
}
public class Option {
  String name;
  int price;
}
public enum CupSize {Small, Tall, Grande, Venti;}
```

Fig. 1. Snippet of Order class and Option class in Java

```
<order>
  <size>Grande</size>
  <product>Cafeineless Coffee</product>
  <date>March 3,2012</date>
  <price>3.14</price>
  <count>2</count>
  <options>
    <option>
      <name>option0000</name>
      <price>0</price>
    </option>
    <option>
      <name>option0001</name>
      <price>1</price>
    </option>
  </options>
</order>
```

Fig. 2. XML document for an order with two options

```
Order anOrder = new Order();
anOrder.setSize(CupSize.Grande);
anOrder.setProduct("Cafeineless Coffee");
anOrder.setDate("March 3,2012");
anOrder.setPrice(3.14);
anOrder.setCount(2);

XStream xstream = new XStream();
xstream.alias("order", Order.class);
xstream.alias("option", Option.class);
FileOutputStream os=new FileOutputStream("a.xml");
xstream.toXML(anOrder,os);

FileInputStream is=new FileInputStream("a.xml");
Order theOrder =(Order)xstream.fromXML(is);
```

Fig. 3. Snippet of a program using XStream

```
{"order": {
  "size": "Grande",
  "product": "Cafeineless Coffee",
  "date": "March 3,2012",
  "price": 3.14,
  "count": 2,
  "options": [
    {
      "name": "option0000",
      "price": 0
    },
    {
      "name": "option0001",
      "price": 1
    }
  ]
}}
```

Fig. 4. JSON data for an order with two options serialized using XStream

II. Object Serialization

For comparison of object serialization libraries, an Order class and an Option class in Java as shown in Fig. 1 were chosen to express a coffee order with some options. It is also used as a common example to perform an experiment for the comparison works in next section. The values of the example are assigned to all fields as shown in Fig. 2. It is a XML document containing an order with two options.

A. Serialization in XML

XStream[4] is a Java library to serialize objects in XML and to deserialize the objects. We do not need to modify source code. It uses a reflection mechanism at run time to investigate in the object graph to be serialized. In the case of XStream, we do not need any schema definitions. Class names and field names become element names in the XML document to be serialized. Fig. 2 shows a XStream output to express a coffee order with two options. The size of the XML document is 332 bytes.

To serialize objects using XStream, all we have to do is to instantiate a XStream object and to call a toXML() method shown in Fig. 3. For deserialization, all we have to do is to call a fromXML() method and to cast the object. In Fig. 3, alias methods are written to map the Order class to an order element and the Option class to an option element.

In current version of XStream, JsonHierarchicalStream-Driver class provides serialization in JSON. Fig. 4 shows an output in JSON using XStream. The size of the file in JSON is 260 bytes. It is smaller than the serialized file in XML using XStream.

B. Serialization in JSON

There are a lot of Java libraries for object serialization in JSON. In this section, seven libraries to serialize in JSON are briefly explained.

Flexjson[5] is a lightweight library for serializing and deserializing Java objects into and from JSON. It provides an easy way to specify to serialize fields and not to serialize fields.

Gson[6] was originally created to use inside Google. It can work with arbitrary Java objects to convert into representation in JSON. Even if we do not have source code, we can serialize it using Gson. Gson is easy to learn and implement. All we have to do is invoke two methods: toJson() and fromJson().

```
option java_package = "org.rugson.proto;
message Order {
  enum CupSize {
    Small = 0; Tall = 1; Grande = 2; Venti = 3;
  }
  required CupSize size = 1;
  required string product = 2;
  required string date = 3;
  required double price = 4;
  required int32 count = 5;
  message Option {
    required string name = 1;
    required int32 price = 2;
  }
  repeated Option options = 6;
}
```

Fig. 5. Schema definition for Protostuff

Jackson[7] aims to be the best possible combination of fast, correct, lightweight, and ergonomic for developers. It offers three variants to process JSON. They are streaming API, tree model and data binding. The streaming API is similar as StAX[8]. The tree model provides an in-memory tree representation of a JSON document and it is similar as DOM[9], The data binding converts JSON to and from Java beans to provide an easy way to convert Java objects to JSON and back.

Json-lib[10] is based on the work by Douglas Crockford who has popularized JSON. It requires the following libraries: Apache commons (Commons Lang, Commons BeanUtils, Commons Collections, Commons Logging)[11] and ezmorph[12]. Developers have to install these five jar files to execute programs using Json-lib.

The json-smart[13] provides fast and highly customizable using Java generics. There are no external dependencies.

The goal of JsonMarshaller[14] is simplicity and efficiency. It uses two annotations @Entity and @Value to define Java classes. We have to embed these annotations in source code so that we need all source code to serialize.

Protostuff[15] is an extended version of google's Protocol Buffers[16]. Protostuff compiler is provided to read schema definitions and generate classes with accessors. Before programming, we have to prepare schema definitions for the structured data shown in Fig. 5. Users define protocol buffer message types. Each protocol buffer message contains a series of name-value pairs. It has uniquely numbered fields, and each field has a name and a value type. The value type can be integer, floating-point, boolean, string, enumeration, or other protocol buffer message type. Protostuff supports many formats which are protostuff native format, JSON, SMILE (binary JSON), XML, YAML and KVP.

Protostuff provides flexibility about usage of schema definitions. We can choose to read the schema definitions at run time. Fig.6 shows a snippet of a program to read the schema definitions and serialize an order.

```
Order anOrder = new Order();
.....
Schema<Order> schema
        = RuntimeSchema.getSchema(Order.class);
PrintWriter pw = new PrintWriter("order.json");
JsonIOUtil.writeTo(pw, anOrder, schema, false);
```

Fig. 6. Schema definition for Protstuff at run time

C. Serialization in Binary

A mechanism for object serialization is included in standard Java packages. The writeObject() method in ObjectOutputStream class is responsible for serialization, and readObject() method in ObjectInputStream class is used to restore it. We do not need schema definitions for serialization and deserialization. All we have to do is to embed "implements Serializable" in serialized classes and to use the writeObject() method for serialization and the readObject() method for deserialization. In the same case as a coffee order with two options shown in Fig. 2, the standard object serialization in Java writes a file with 380 bytes. The size is larger than the XML file shown in Fig. 2 and the JSON file shown in Fig. 4.

Protocol Buffers[16] is used in Google to encode structured data in an efficient format. It is encoded to binary data for implementing smaller and faster serialization. Before programming, we have to prepare schema definitions for the structured data shown in Fig. 5. The syntax to define schema is same as Protostuff.

In Protocol Buffers, the schema compiler reads the schema definition and generates data access classes including accessors for each field (like getProduct() and setProduct()), methods to serialize and deserialize the structured data and special builder classes to encapsulate internal data structure. It supports three programming languages, Java, C++ and Python.

Thrift[17] is a software library and set of code-generation tools originally developed by Facebook. The currently supported programming languages are C++, Java, Python, PHP, Ruby, Erlang, Perl, Haskell, C#, Cocoa, JavaScript, Node.js, Smalltalk, and OCaml. We have to prepare schema definitions shown in Fig. 7 and the code generation tool generates source code for a specified programming language.

Apache Avro[18] is serialization framework developed as a Hadoop subproject. It provides serialization formats for persistent data and communication between Hadoop nodes. To integrate with dynamic programming languages, code generation is not required to serialize and deserialize structured data, but schema definitions for Apache Avro are required. Avro uses JSON for schema definitions. When Avro data is read, the schema is read at run time before reading the data. One of features for Avro is that it enables direct serialization from schema definitions without code generation.

TABLE I
COMPARISON OF OBJECT SERIALIZATION

	Is schema required ?	Is compiler provided ?	Binary or ascii	Supported programming languages
XStream	No	No	XML, JSON	Java
JSON libraries	No	No	Ascii	Java
Protostuff	Optional	Yes	Many	Java
Java serialization	No	No	Binary	Java
Protocol Buffers	Yes	Yes	Binary	Java, C++, Python
Thrift	Yes	Yes	Binary	14 languages including C++, Java, Ruby
Apache Avro	Yes	No	Binary, JSON	C, C++, C#, Java, PHP, Python

```
namespace java org.rugson.thrift
enum CupSize {
  Small = 0, Tall = 1, Grande = 2, Venti = 3,
}
struct Option {
  1: string name,
  2: i32 price,
}
struct Order {
  1: CupSize size,
  2: string product,
  3: string date,
  4: double price,
  5: i32 count,
  6: list<Option> options,
}
```

Fig. 7. Schema definition for Thrift

III. COMPARISON AND EXPERIMENT

A. Comparison of Object Serialization Libraries

In the previous section, many libraries of serialization were explained. TABLE I shows some qualitative aspects as the following;

- whether schema definition is required or not,
- whether schema compiler is required or not,
- whether serialization is based on binary or ascii, and
- which programming languages are supported.

If we need multiple programming language support, schema definitions are required like Thrift and Protocol Buffers. The schema definitions are used to map them to a target programming language. Thrift supports many programming languages because there are many contributed developers. It shows that world-wide open source development is important to increase activity of software projects.

B. Experiment

To examine the performance in serializing and deserializing structured data, an experiment was designed using the following hardware and software:

- Hardware: iMac (by Apple Inc.) with Intel Core2 Duo 2.66 GHz and 2 GB memory
- Operating System: Mac OS X version 10.6.8
- Java: version 1.6.0_29

Current version of object serialization libraries were selected shown in the following:

XStream	1.4.2
Flexjson	2.1
Gson	2.1
Jackson	1.9.5
Json-lib	2.4
JsonMarshaller	0.21
json-smart	2.0 beta2
Protostuff	1.0.4
Protocol Buffers	2.4.1
Thrift	0.8.0
Apache Avro	1.6.3

The experiment was designed as follows:

- Ten kinds of orders were prepared. They were orders with ten sizes of options: 0, 100, 200, 300, 400, 500, 600, 700, 800 and 900 options.
- 500 iterations were executed for warming-up, and then 500 iterations were executed for measuring.
- The serialized file size was measured and the execution time was measured using System.currentTimeMillis() shown in Fig.8.

```
long start = System.currentTimeMillis();
for (int i = 0; i < ITERATION; i++) {
  json = serialize(order);
}
long end = System.currentTimeMillis();
double time = (double)(end - start)/ITERATION;
```

Fig. 8. Snippet of serialization and deserialization programs

The average size of ten kinds of serialized files and the size in the case of zero options are given bellow.

	Average (bytes)	no options (bytes)
XStream in XML	36973.6	160
JSON libraries	15307.9	106
Numberd JSON	12124.2	66
Thrift in binary	11328	78
Object Serialization in Java	10746.6	348
Protobuf	7585.4	47
Avro in binary	5844.1	49

From the point of the average size, the largest is a serialized file using XStream in XML. But in the case of an order with no options, the standard object serialization in Java writes the largest file. Because the serialized data by the standard object serialization in Java contains type information. The standard object serialization writes objects in binary format, but the

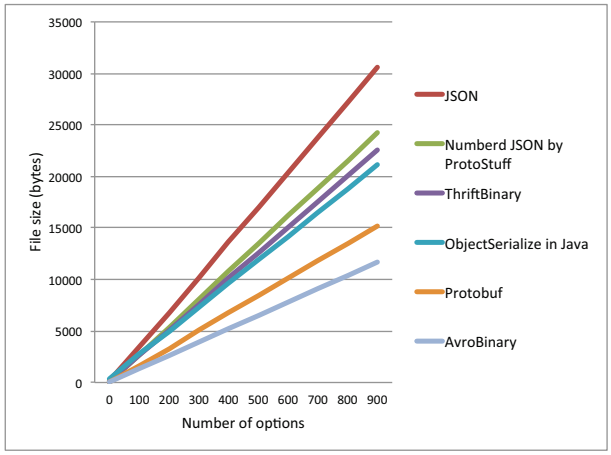

Fig. 9. Size of object serialization

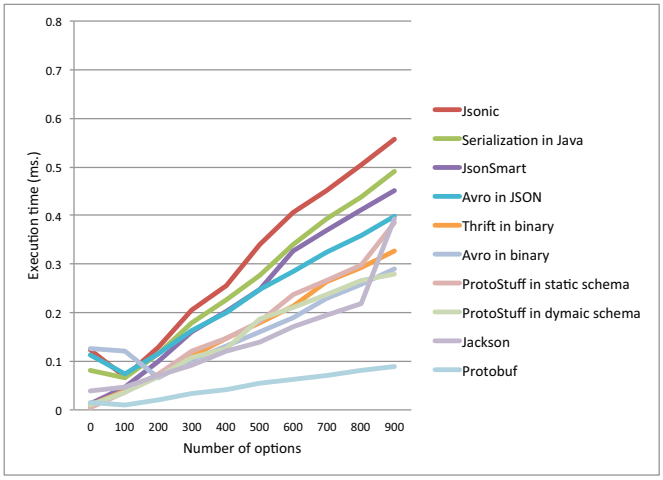

Fig. 10. Execution time of object serialization

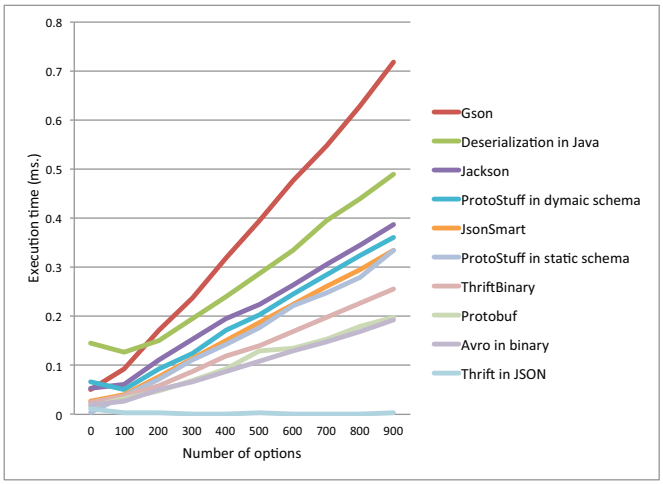

Fig. 11. Execution time of object deserialization

size is larger than the serialized files in ascii format, such as XML and JSON. The sizes of serialized files in JSON using any libraries (XStream, JsonLib, FlexJson, ThriftJson, Gson, JsonMarshaller, Jsonic, JsonSmart, AvroJson and ProtoStuff) are same, and it is less than a half of XML. Other four (Thrift in binary, Object Serialization in Java, Protobuf and Avro in binary) are binary based files. The smallest is a serialized file using Avro.

To clarify the tendency, twenty libraries without XStream in XML are plotted in Fig. 9. In each technique, there is a linear increase in the number of options. The size of the binary serialized files in Protobuf and Avro are smaller than JSON files.

The average execution time to serialize ten kinds of serialized files are given bellow.

	Average (ms.)
XStream in JSON	9.7858
JsonLib	2.9422
XStream in XML	2.8698
FlexJson	0.9740
ThriftJson	0.7422
Gson	0.4756
JsonMarshaller	0.4094
Jsonic	0.3044
Object serialization in Java	0.2606
JsonSmart	0.2328
AvroJson	0.2278
Thrift in binary	0.1654
Avro in binary	0.1672
ProtoStuff with static schema	0.1750
ProtoStuff with dymaic schema	0.1532
Jackson	0.1488
Protobuf	0.0476

The worst one is serialization in JSON using XStream. It takes about 10 milliseconds to serialize an order objects with options. It is very slower than other serialization techniques. The worst top three are XStream in JSON, JsonLib and XStream in XML. To clarify the details, Fig. 10 shows fourteen techniques without the three. If serialization in JSON is required, Jackson is the best choice from the viewpoint of execution time. If serialization in any format is permitted, Protobuf is the best choice from the viewpoint of execution time.

The average execution time to deserialize ten kinds of serialized files are given bellow.

	Average (ms.)
JsonLib	22.112
XStream in XML	5.1280
XStream in JSON	4.2166
FlexJson	1.4978
JsonMarshaller	0.9874
Jsonic	0.9038
Avro in Json	0.4036
Gson	0.3638
Object Serialization in Java	0.2800
Jackson	0.2098
ProtoStuff with dymaic schema	0.1922
JsonSmart	0.1710
ProtoStuff in static schema	0.1626
Thrift in binary	0.1316
Protobuf	0.1058
Avro in binary	0.0996
Thrift in JSON	0.0020

The programs for deserialization need to read the serialized file, parse it and reconstruct an order with options. The worst

one is deserialization in JSON using JsonLib. It takes more than twenty milliseconds to deserialize a serialized file in JSON. It is extremely slower that other techniques. To clarify the details, Fig. III-B shows ten techniques without the worst seven. In the figure, Thrift in JSON is the best. But the result is doubtful. Another investigation in the case should be carried out. In other cases, if deserialization in JSON is required, JsonSmart is the best choice. If deserialization in any format is permitted, Avro in binary and Protobuf are the best choice.

IV. Conclusion

This paper compares twelve libraries of object serialization from qualitative and quantitative aspects. It is clear that there is no best solution. Each library makes good in the context it was developed.

If we need multiple programming language support, schema definitions are required. A schema compiler reads the schema definitions and it generates some code fragments in a specified programming language. From quantitative aspects, the size of binary-based serialized data is better than XML-based and JSON-based serialization. If we need easy interoperability with dynamic languages, Apache Avro and Protostuff are the best in the twelve serialization libraries.

The author believes good readability and simplicity of structured data representation so that ascii-based representation is desirable. Moreover a schema compiler is important for good performance and multiple programming language support. An original object serialization library developed by the author is now under development. Research about survey of object serialization and development of the original library will continue. The results will be published in a future paper.

Acknowledgment

This work is supported by JST A-STEP (Adaptable and Seamless Technology Transfer Program through target-driven R&D) Exploratory Research.

References

[1] George Coulouris, Jean Dollimore and Tim Kindberg, Distributed Systems: Concepts and Design, 2nd edition, Addison-Wesley, 1994.
[2] Andrew D. Birrell and Bruce Jay Nelson Implementing Remote Procedure Calls ACM Transaction on Computer Systems, Vol.2, No.1, pp.39-59, 1984.
[3] JSON, http://www.json.org/ (accessed at March 22, 2012).
[4] XStream – About XStream, http://xstream.codehaus.org/ (accessed at March 22, 2012).
[5] JSON Serialization Usage, http://flexjson.sourceforge.net/ (accessed at March 22, 2012).
[6] google-gson - A Java library to convert JSON to Java objects and vice-versa, http://code.google.com/p/google-gson/ (accessed at March 22, 2012).
[7] Jackson JSON Processor - Home, http://jackson.codehaus.org/ (accessed at March 22, 2012).
[8] JSRs: Java Specification Requests, JSR 173: Streaming API for XML, http://www.jcp.org/en/jsr/summary?id=173 (accessed at March 22, 2012).
[9] Document Object Model (DOM) Specifications, http://www.w3.org/DOM/DOMTR (accessed at March 22, 2012).
[10] Maven - Json-lib::Welcome, http://json-lib.sourceforge.net/ (accessed at March 22, 2012).
[11] Apache Commons, http://commons.apache.org/ (accessed at March 22, 2012).
[12] Maven - Welcome to EZMorph, http://ezmorph.sourceforge.net/ (accessed at March 22, 2012).
[13] json-smart - A small and very fast json parser/generator for java, http://code.google.com/p/json-smart/ (accessed at March 22, 2012).
[14] jsonmarshaller - Fast, Lightweight, Easy to Use and Type Safe JSON marshaling library for Java, http://code.google.com/p/jsonmarshaller/ (accessed at March 22, 2012).
[15] protostuff - java serialization library, proto compiler, code generator, protobuf utilities, gwt overlays, j2me, android and kindle interoperability, http://code.google.com/p/protostuff/ (accessed at March 22, 2012).
[16] protobuf – Protocol Buffers, http://code.google.com/p/protobuf/ (accessed at March 22, 2012).
[17] Apache Thrift, http://thrift.apache.org/ (accessed at March 22, 2012).
[18] Welcome to Apache Avro, http://avro.apache.org/ (accessed at March 22, 2012).