

Comparative Survey of Object Serialization Techniques and the Programming Supports

Kazuaki Maeda

Abstract—This paper compares six approaches of object serialization from qualitative and quantitative aspects. Those are object serialization in Java, IDL, XStream, Protocol Buffers, Apache Avro, and MessagePack. Using each approach, a common example is serialized to a file and the size of the file is measured. The qualitative comparison works are investigated in the way of checking whether schema definition is required or not, whether schema compiler is required or not, whether serialization is based on ascii or binary, and which programming languages are supported. It is clear that there is no best solution. Each solution makes good in the context it was developed.

Keywords—structured data, serialization, programming

I. INTRODUCTION

Technology has been changing and continues changing day by day. In software research area, a compiler is one of the oldest and maturing technologies. If new technology can be embed in the compiler, we can have a chance to change it to a more convenient tool.

The compiler reads source code and generates object code. In the front-end of the compiler, it reads the plain text, analyzes it, and builds a tree to represent the structure in source code. To represent the structured data, the compiler front-end builds the AST which stands for an abstract syntax tree. The AST contains syntactic and semantic information of source code. The compiler back-end receives the AST from the front-end and generates object code.

Most of the compiler does not export the AST out of the compiler. The author had an experience to develop a compiler front-end for Java version 5.0 from scratch to analyze source code to use in commercial products. The task was complicated because approximately two hundreds of classes, containing more than one thousands of methods and fields, were required to build an AST. If we develop some software tools, for example, a reverse engineering tool or a source code search engine, we need to build a compiler front-end from scratch. It is not easy task to develop it.

XML has gained the popularity in a short time, as can be seen in the following [1]:

XML is a buzzword you will see everywhere on the Internet, but it's also a rapidly maturing technology with powerful real-world applications, particularly for the management, display, and organization of data.

XML is currently used as a standard language for data representation in wide application area.

Kazuaki Maeda is with the Department of Business Administration and Information Science, Chubu University, Kasugai, Aichi, 487-8501 Japan (e-mail: kaz@acm.org).

One of the XML applications to the compiler is JavaML [2]. It is a typical representation providing syntactic and semantic information after analyzing Java source code. The JavaML compiler reads the source code, and create AST, and writes the AST as a XML document. Once software tools are implemented using the JavaML representation, we do not need to develop a compiler front-end. The software tools can easily obtain information using XML utilities about Java source code which contains class declarations, method declarations, method invocations. We do not need to reinvent the wheel.

In contrast of XML advantages, there are some disadvantages of XML. One of the disadvantages is that XML documents are composed of many redundant tags, those are start tags and end tags. They are written in human readable text format, but it is difficult to read and understand them. To overcome the disadvantages, JSON is currently becoming a popular representation for structured data [3], [4]. Structured data in 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 because of XML's end tags.

Object serialization [5] is included in standard Java packages, and it supports the translation of objects (and objects reachable from them) into a byte stream. It is called serialization. Moreover Java also supports the reconstruction of objects from the byte stream. It is called deserialization.

The object serialization in Java is useful for saving objects to persistent storage media such as hard disks. The value of each field in the objects is saved 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. This is the fact that even if the object is serialized in binary-based format, the size is not always small. Therefore, the author thinks we need some experiments to investigate the size of serialized data. Moreover, there is another disadvantage. It is that 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.

This paper describes comparison with six approaches of object serialization from quantitative and qualitative aspects. A common example is chosen, it is serialized to a file using six approaches, and the file size is measured. From qualitative aspect, the comparison works are investigated as the following points;

- whether the schema definition is required or not,

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

Section II describes six approaches of object serialization and APIs using small sample programs. Section III explains the comparison and a new structured data representation designed by the author. Section IV summarizes this paper.

II. APPROACHES OF OBJECT SERIALIZATION

To compare six approaches for object serialization, an Order class and an Option class in Java to express a coffee order with some options were chosen as shown in Fig. 1. It is used as a common example for the comparison works. The values of the example are assigned to all fields as shown in Fig. 2. It is a XML document containing an order with some options.

```
public class Order {
    String product;
    CupSize size;
    String date;
    int price;
    List<Option> options;
}
public class Option {
    String name;
    int price;
}
```

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

A. Object Serialization in Java

A mechanism to represent structured data is included in standard Java packages. It provides encoding of objects and saving the current state to a byte stream (ObjectOutputStream). The encoding is called serialization. It also provides reconstruction of the equivalent objects from a byte stream (ObjectInputStream). The reconstruction is called deserialization. The writeObject method in the class ObjectOutputStream is responsible for serialization, and the readObject method in the class ObjectInputStream is used to restore it.

For example, we can write the state of the objects to an ObjectOutputStream "os" as described in Fig. 3, and read objects from an ObjectInputStream "is" in the program. 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.

The serialized data using object serialization in Java is written in a binary format. The detailed information of the binary format is documented in Java Object Serialization Specification [6]. But the specification is dependent on Java so that it cannot be used in other programming languages. Java is one of the popular programming languages for the application development, but other programming languages may have to be used in distributed computing. Moreover, there are some cases with critical constraints including the lack of the computing resource, the severe timing constraint, and

```
<starbucks.Order>
  <product>Coffee Frappuccino</product>
  <size>Grande</size>
  <date>December 25,2011</date>
  <price>460</price>
  <options>
    <starbucks.Option>
      <name>Shot</name>
      <price>50</price>
    </starbucks.Option>
    <starbucks.Option>
      <name>Extra whip</name>
      <price>50</price>
    </starbucks.Option>
    <starbucks.Option>
      <name>Hazelnut</name>
      <price>0</price>
    </starbucks.Option>
    <starbucks.Option>
      <name>Caramel sauce</name>
      <price>0</price>
    </starbucks.Option>
    <starbucks.Option>
      <name>Chocolate sauce</name>
      <price>50</price>
    </starbucks.Option>
    <starbucks.Option>
      <name>Chocolate chip</name>
      <price>50</price>
    </starbucks.Option>
  </options>
</starbucks.Order>
```

Fig. 2. XML document for an order with options

```
Order anOrder = new Order();
anOrder.setProduct("Coffee Frappuccino");
anOrder.setSize(CupSize.Grande);
anOrder.setDate("December 25,2011");
anOrder.setPrice(460);
List list = new ArrayList();
anOrder.setOptions(list);
Option thisOption = new Option();
thisOption.setName("Shot");
thisOption.setPrice(50);
list.add(thisOption);
// more five options are added to the list ....

FileOutputStream fos
    = new FileOutputStream("order.dat");
ObjectOutputStream os
    = new ObjectOutputStream(fos);
os.writeObject(theOrder);

FileInputStream fis
    = new FileInputStream("order.dat");
ObjectInputStream ois
    = new ObjectInputStream(fis);
Order theOrder = (Order)ois.readObject();
```

Fig. 3. Snippet of a program using writeObject and readObject

the connectivity to the legacy software. Therefore, multiple programming languages should be supported for approaches of object serialization.

B. IDL

As a practical application, we can use an interface description language (IDL)¹ compiler in Scorpion Toolkit [8]. The IDL is a notation to define structured data. The definition for the common example is shown in Fig. 4. It is called “schema definition” in this paper.

```
Structure StarbucksOrder Root Order Is
  Order => product : String,
           size   : CupSize,
           date   : String,
           price  : Integer,
           options : Seq Of Option;

  For CupSize Use Representation Enumerated;
  CupSize ::= Small | Tall | Grande | Venti;
  Small =>; Tall =>; Grande =>; Venti =>;

  Option => name   : String,
           price  : Integer;

End
```

Fig. 4. Schema definition in IDL

IDL was developed in 1980’s, but the software tools are still available. One of the tools is an IDL compiler which reads schema definitions in the IDL and generates useful functions to manipulate an AST including code fragments to map the definition to a specified programming language. The compiler also generates functions for serialization to translate the AST to a representation, and for deserialization to reconstruct the AST from the representation. In this paper, the compiler which reads schema definitions and generates code fragments is called “schema compiler.”

The IDL user writes a program in terms of the target programming language data declarations and utilities generated by the schema compiler. The program writes the structured data to a file using an ascii representation or a binary representation.

If we choose C programming language for the target, the schema compiler generates C data type declarations, macro declarations and function declarations. In Fig. 5, the generated macros and functions are shown: NOrder, productOfOrder, sizeOfOrder, dateOfOrder, priceOfOrder, order_out(), and order_in().

```
Order anOrder = NOrder;
productOfOrder(anOrder) = "Coffee Frappuccino";
sizeOfOrder(anOrder) = GrandeToCupSize(NGrande);
dateOfOrder(anOrder) = "December 25,2011";
priceOfOrder(anOrder) = 460;

FILE *ofp = fopen("order-idl.txt","w");
order_out(ofp,theOrder);

FILE *ifp = fopen("order-idl.txt","r");
Order theOrder = order_in(ifp);
```

Fig. 5. Snippet of a program using IDL

¹The IDL in this context is different from OMG IDL [7].

The Scorpion Toolkit is very useful in developing language-oriented software tools. However, the representation of structured data for the toolkit is not designed for general purposes.

C. XStream

XStream [9] is a Java library to serialize objects in XML and to deserialize the objects. We do not need to implement the Serializable interface like object serialization in Java. 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 some options.

To serialize objects using XStream, all we have to do is to instantiate the XStream object and to call a toXML method shown in Fig. 6. For deserialization, all we have to do is to call a fromXML method and to cast the object. In current version of XStream, JsonHierarchicalStreamDriver class provides serialization in JSON.

```
Order anOrder = new Order();
anOrder.setProduct("Coffee Frappuccino");
anOrder.setSize(CupSize.Grande);
anOrder.setDate("December 25,2011");
anOrder.setPrice(460);

XStream xstream = new XStream();
FileOutputStream fos
    = new FileOutputStream("order.xml");
xstream.toXML(anOrder, fos);

FileInputStream fis
    = new FileInputStream("order.xml");
xstream = new XStream();
Order theOrder = (Order)xstream.fromXML(fis);

JsonHierarchicalStreamDriver json
    = new JsonHierarchicalStreamDriver();
xstream = new XStream(json);
fos = new FileOutputStream("order.json");
xstream.toXML(theOrder, fos);
```

Fig. 6. Snippet of a program using XStream

D. Protocol Buffers

Protocol Buffers [10] 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. 7.

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.

In Protocol Buffers, the schema compiler reads the schema definition and generates data access classes including accessors for each field (like getProduct() and setProduct()),

```

option java_package = "org.example";

message Order {
  enum CupSize {
    Small = 1; Tall = 2;
    Grande = 3; Venti = 4;
  }
  required string product = 1;
  required CupSize size = 2;
  required string date = 3;
  required int32 price = 4;

  message Option {
    required string name = 1;
    required int32 price = 2;
  }

  repeated Option options = 5;
}

```

Fig. 7. Schema definition for Protocol Buffers

methods to serialize/deserialize the structured data and special builder classes to encapsulate internal data structure. Fig. 8 shows the accessors, serialization and deserialization in Java. In current version of Protocol Buffers, three programming languages, Java, C++ and Python, are supported.

```

Order.Builder anOrder = Order.newBuilder();

anOrder.setProduct("Coffee Frappuccino");
anOrder.setSize(Order.CupSize.Grande);
anOrder.setDate("December 25, 2011");
anOrder.setPrice(460);

FileOutputStream fos
  = new FileOutputStream("order.dat");
anOrder.build().writeTo(fos);

FileInputStream fis
  = new FileInputStream("order.dat");
Order theOrder = Order.parseFrom(fis);

```

Fig. 8. Snippet of a program using Protocol Buffers

E. Apache Avro

Apache Avro [11] 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 shown in Fig. 9. 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. Fig. 10 shows snippets of a program to serialize and deserialize an order object using Apache Avro. In the program, first encoder serializes the object in a binary format to a file, and a decoder

```

{
  "namespace": "org.example",
  "name": "Order",
  "type": "record",
  "fields": [
    {"name": "product", "type": "string"},
    {"name": "size", "type": "enum",
     "symbols" : ["Small","Tall","Grande","Venti"]
    },
    {"name": "date", "type": "string"},
    {"name": "price", "type": "int"},
    {"name": "options", "type": {
      "type" : "array",
      "items": org.example.Option
    } } ] }

```

Fig. 9. Schema definition for Apache Avro

```

File file = new File("schema-order.avro");
Schema schema = AvroUtils.parseSchema(file);

GenericRecord r = new GenericData.Record(schema);
r.put("product", "Coffee Frappuccino");
r.put("size", "Grande");
r.put("date", "December 25, 2011");
r.put("price", 460);

FileOutputStream fos
  = new FileOutputStream("order.dat");
EncoderFactory ef = EncoderFactory.get();
Encoder enc = ef.binaryEncoder(fos, null);
GenericDatumWriter<GenericRecord> writer
  = new GenericDatumWriter<GenericRecord>(schema);
writer.write(r, enc);

FileInputStream fis
  = new FileInputStream("order.dat");
DecoderFactory df = DecoderFactory.get();
Decoder dec = df.binaryDecoder(fis, null);
GenericDatumReader<GenericRecord> reader
  = new GenericDatumReader<GenericRecord>(schema);
GenericRecord theOrder
  = reader.read(null, dec);

fos = new FileOutputStream("order.json");
enc = ef.jsonEncoder(schema, fos);
w = new GenericDatumWriter<GenericRecord>(schema);
writer.write(r, enc);

```

Fig. 10. Snippet of a program using Apache Avro

deserializes the object from the file. EncoderFactory class provides two kinds of encoders, binary-based and JSON-based. If we need to serialize it in JSON, we can simply change the Encoder implementation from

```
Encoder enc = ef.binaryEncoder(fos, null);
```

to

```
Encoder enc = ef.jsonEncoder(s, fos);
```

After changing the Encoder to JsonEncoder, JSON data shown in Fig. 11 is serialized.

```
{"product": "Coffee Frappuccino", "size": "Grande", "date": "December 25, 2011", "price": 460, "options": []}
```

Fig. 11. Representation of an order in JSON data using Avro

F. MessagePack

MessagePack [12] is an efficient object serialization library using a binary format. The schema definition is not always required. If Java is chosen as a programming language, an annotation @Message enables us to serialize public fields in objects as shown in Fig. 12.

```
@MessagePackMessage
public class Order {
    public String product;
    public String size;
    public String date;
    public int price;
    public List<Option> options;
}
```

Fig. 12. Order class using MessagePack without IDL

A schema definition is provided shown in Fig. 13 and the syntax is similar as one of Protocol Buffers. Current implementation supports only Java, but in specification multiple programming languages will be supported.

```
namespace com.example
enum Size {
    0: Small
    1: Tall
    2: Grande
    3: Venti
}
message Option {
    1: string name
    2: int price
}
message Order {
    1: string name
    2: Size size
    3: string date
    4: int price
    5: list<Option> options
}
```

Fig. 13. Schema definition for MessagePack

If a program need to serialize objects, all we have to do is to call a pack method in MessagePack class shown in Fig .14. To deserialize objects from bytes, all we have to do is to call a unpack method in MessagePack class.

In current implementation, seventeen programming languages are supported. Those are C, C++, C#, Java, D, Go, Node.JS, JavaScript, Perl, Python, Ruby, Scala, Lua, PHP, Erlang, Haskell and OCaml.

```
Order anOrder = new Order();

anOrder.setProduct("Coffee Frappuccino");
anOrder.setSize("Grande");
anOrder.setDate("October 6, 2011");
anOrder.setPrice(460);
anOrder.setOptions(new ArrayList<Option>());

byte[] raw = MessagePack.pack(anOrder);
FileOutputStream output
    = new FileOutputStream("order.dat");
output.write(raw);

FileInputStream fis
    = new FileInputStream("order.dat");
byte[] allbytes = // read all bytes from fis
Order theOrder
    = MessagePack.unpack(allbytes, Order.class);
```

Fig. 14. Snippet of a program using MessagePack

III. DISCUSSION

A. Comparison in Six Approaches of Object Serialization

In the previous section, six approaches of serialization are explained. From quantitative aspects, the size of serialized data is measured and the results are shown in TABLE I. Binary-based serialization using IDL, Protocol Buffers, Apache Avro, and Message Pack are small in size. Object serialization in Java is binary-based representation, but the size of serialized data is not small. Because the serialized data contains much type information for Java. The case of XStream shows the size of JSON-based serialization is about 36% decreased in contrast with the size of XML-based serialization. Binary-based serialization using Apache Avro is the smallest in this case. XML-based serialization using XStream is the largest in size.

TABLE I
SIZE OF THE SERIALIZED DATA FOR THE COFFEE ORDER

Approach	Binary or Ascii	Size (bytes)
Serialization in Java	Binary	502
IDL	Ascii	339
	Binary	135
XStream	Ascii in XML	782
	Ascii in JSON	497
Protocol Buffers	Binary	141
Apache Avro	Binary	121
	Ascii in JSON	296
Message Pack	Binary	128

TABLE II shows some qualitative aspects as the following;

- whether schema definition is required or not,
- whether schema compiler is required or not,

TABLE II
COMPARISON OF OBJECT SERIALIZATION

	Is schema required ?	Is compiler provided ?	Binary or ascii	Supported programming languages
Java serialization	No	No	Binary	Java
IDL	Yes	Yes	Binary, Ascii	C, Pascal
XStream	No	No	XML, JSON	Java
Protocol Buffers	Yes	Yes	Binary	Java, C++, Python
Apache Avro	Yes	No	Binary, JSON	C, C++, C#, Java, PHP, Python
Message Pack	Optional	Yes	Binary	17 languages including C, Java, Ruby

- 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 IDL and Protocol Buffers. The schema definitions are used to map them to a target programming language. Message Pack 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. Comparison with RugsOn

The author designed RugsOn, a new representation written in a text-based data format for multiple programming languages [13]. RugsOn was designed to reach good readability and simplicity of structured data representation shown in Fig. 15. The representation of structured data should be simple and easy to read. Moreover, structured data should be represented using a simple language and it should be available for multiple programming languages. A common subset of Ruby, Groovy and Scala was carefully chosen for RugsOn design so that one representation in RugsOn is available for multiple programming languages.

```
Order{
  "Grande".size
  "October 6,2011".date
  660.price
  "Coffee Frappuccino".product
}
```

Fig. 15. Serialized data using RugsOn

The size of serialized data for the common coffee order program is 385 bytes. RugsOn is not binary-based format, but the size is smaller than object serialization in Java.

A schema definition language was designed using a subset of Ruby syntax. A program generator was developed to create Ruby, Groovy, Scala and Java programs from the schema definitions. It means that four programming languages are supported in current version.

IV. CONCLUSION

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

If we need multiple programming language support, schema definitions are required using a schema definition language

and a schema compiler generates some code fragments from the schema definitions. 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 is the best in the six serialization approaches.

The author believes good readability and simplicity of structured data representation so that ascii-based representation is desirable. Research about survey of object serialization and development of RugsOn will continue. The results will be published in a future paper.

REFERENCES

- [1] D. Hunter, J. Rafter and others, *Beginning XML*, 4th edition, Wiley, 2007.
- [2] Greg Badros, *JavaML: A Markup Language for Java Source Code*, 9th International World Wide Web Conference, <http://www9.org/w9cdrom/index.html>, 2000.
- [3] JSON, <http://www.json.org/> (accessed at Oct. 20, 2011).
- [4] The application/json Media Type for JavaScript Object Notation (JSON), RFC 4627, <http://www.ietf.org/rfc/rfc4627.txt> (accessed at Oct. 20, 2011).
- [5] Java SE 7 Serialization-related APIs and Developer Guides, <http://download.oracle.com/javase/7/docs/technotes/guides/serialization/> (accessed at Oct. 20, 2011).
- [6] Java Object Serialization Specification: Contents, <http://download.oracle.com/javase/7/docs/platform/serialization/spec/serialTOC.html> (accessed at Oct. 20, 2011).
- [7] OMG, Common Object Request Broker Architecture (CORBA), OMG Released Versions Of CORBA, <http://www.omg.org/spec/CORBA/3.1/>.
- [8] Richard Snodgrass, *The Interface Description Language: Definition and Use*. Computer Science Press, 1989.
- [9] XStream – About XStream, <http://xstream.codehaus.org/> (accessed at Oct. 20, 2011).
- [10] protobuf – Protocol Buffers, <http://code.google.com/p/protobuf/> (accessed at Oct. 20, 2011).
- [11] Welcome to Apache Avro, <http://avro.apache.org/> (accessed at Oct. 20, 2011).
- [12] The MessagePack Project, <http://msgpack.org/> (accessed at Oct. 20, 2011).
- [13] Kazuaki Maeda, *Executable Representation for Structured Data Using Ruby and Scala*, 10th International Symposium on Communications and Information Technologies, pp.127–132, 2010.



Kazuaki Maeda He is a professor of Department of Business Administration and Information Science at Chubu University in Japan. He is a member of ACM, IEEE, IPSJ and IEICE. His research interests are Compiler Construction, Domain Specific Languages, Object-Oriented Programming, Software Engineering and Open Source Software.