JSON on Mobile: is there an Efficient Parser?

Ricardo Queirós

CRACS & INESC-Porto LA & DI-ESEIG/IPP, Porto, Portugal
ricardo.queiros@eu.ipp.pt

Abstract

The two largest causes for battery consumption on mobile devices are related with the display and network operations. Since most application need to share data and communicate with remote servers, communications should be as lightweight and efficient as possible. In network communication, serialization plays a central role as the process of converting an object into a stream of bytes. One of the most popular data-interchange format is JSON (JavaScript Object Notation). This paper presents a survey on JSON parsers in mobile scenarios. The aim of the survey is to find the most efficient JSON parser in mobile communications characterised by high transfer rate of small amounts of data. In the performance benchmark we compare the time required to read and write data with several popular JSON parser implementations such as Gson, Jackson, org.json and others. The results of this survey are important for others that need to select an efficient parser for mobile communication.

1998 ACM Subject Classification D.2.2 Design Tools and Techniques, D.4.4 Communications Management

Keywords and phrases serialization formats, mobile communication

Digital Object Identifier 10.4230/OASIcs.SLATE.2014.93

1 Introduction

Mobile devices have become a necessity for many people around the world. The ability to keep in touch with family and business partners or to share data in real time are only a few of the reasons for the increasing importance of mobile devices. The flip side of this global trend is related with battery consumption. Smartphones are evolving from the past ten years with faster CPUs, cheaper and bigger storage, and higher-quality displays. However, battery technology did not improve at the same pace. The two biggest causes for battery consumption on mobile devices are related with the display and network traffic. The display is a major mobile phone energy hog, that can be softened by reducing its brightness and timeout.

Network operations are unavoidable in today's clouds world where everything is a service. Mobile devices need to communicate to achieve usefulness whether to transmit data over the Internet or to share data with another device. Therefore, developers followed best practices to reduce the amount of network operations in order to increase the battery's life. Basically they all resume to the following four best practices [3]:

- Consider first the need to perform a network call right now. Alternatives are pulling the service at regular intervals or allowing the server to push the data down to the client.
- Consider how much data you need to retrieve. It is possible to use different types of caches (e.g., response cache introduced for *HttpUrlConnection* in Android 4/ICS) and retrieving smaller pages of data from the service will greatly reduce your application's network traffic.
- Use transparent compressions (supported by *HttpUrlConnection* class) verifying that the data retrieved from the server is gzip-compressed.

- Choose a better data format, which usually involves a balance between size optimization and how dynamic the format is. If you can, choose a format that allows you to extend your data definition without losing backward-compatibility. There are several solutions such as XML, YAML, JSON, Protobuf, and others.

The last recommendation touches in a very important factor data transmission over the Internet. In a communication process it is necessary to transform data into a format that is suitable for transmission over the network and that allows the recipient to consume it without any problems. This technique is called serialization. In the context of data storage and transmission, serialization is the process of writing an object to a stream of bytes. That stream can then be sent through a socket, stored to a file and/or database or simply manipulated so that this exact same memory representation can be read later. This last process is called deserialization.

In the serialization realm, XML was used as the standard language for data representation. The most notable advantage regarding the use of XML is its heterogeneous facet. However, when encoding data in XML, the result is typically larger in size than other formats due to XML's well-known verbosity which also negatively affects the reading process. To overcome this disadvantage, JSON [1] is currently becoming a popular data representation. When data is encoded in JSON, the result is typically much smaller in size than an equivalent encoding in XML.

This paper presents a survey on JSON parsers in mobile scenarios. The aim of the survey is to find the most efficient JSON parser implementation in mobile communications. The types of communication are characterized by a high transfer rate of small amounts of data. Based on a performance benchmark we compare the time required to read and write data with several popular JSON parser implementations such as Gson, Jackson, org.json and others. The criteria used for the selection of the parser implementations were based on their popularity.

With this paper we do not intend to present an in depth description of the serialization mechanism. The results of this survey are important for others that need to select an efficient parser for mobile communication.

The remainder of this paper is organized as follows: Section 2 introduces several serialization formats. Then, we focus on the comparison of several JSON parser implementations to evaluate efficiency. Finally, we conclude with a summary of the main contributions of this work.

2 Serialization Formats

Serialization consists in the conversion of an object into a representation that can be transmitted. An application that is aware of the serialization format used can then recreate a serialized object by deserialization. The object is then restored to its original state.

In this process the serialization format plays a central role. There are two types of serialization: textual and binary. The following subsections enumerate various serialization format for both types.

2.1 Textual Serialization

One of the first standard data serialization formats was the External Data Representation (XDR) developed and published in 1987 at Sun Microsystems. XDR became an IETF standard in 1995.

Table 1 Textual serialization formats.

Name	Date	Creator	Based on	Schema/IDL	Human-Readable
CSV	1967	Yakov Shafranovich	n/a	partial	yes
XML	1998	W3C	SGML	yes	yes
XML-RPC	1998	Dave Winer	XML/SOAP	no	yes
JSON	2001	Douglas Crockford	JavaScript	partial	yes
YAML	2001	Clark Evans	C/Perl/XML	partial	yes
Candle	2005	Henry Luo	XML/JSON	yes	yes
OpenDDL	2013	Eric Lengyel	C/PHP	no	yes

In 1998, XML was introduced for asynchronous transfer of structured data between client and server in Ajax Web applications. In this context XML was defined as a human readable text-based encoding that can be used to persist objects and transmit them to other systems regardless of the platform or programming language used. Despite the format verbosity, the human readability and language independent features were very appreciated. In order to overcome the compactness issue, Binary XML has been proposed as an alternative to the regular XML.

To overcome XML's disadvantages, JavaScript Object Notation (JSON) is currently becoming a popular data representation. When data is encoded in JSON, the result is typically smaller in size than an equivalent encoding in XML. JSON is defined as a low-overhead alternative to XML and is commonly used for client-server communication in Web applications. JSON is based on JavaScript syntax, but is supported in other programming languages as well. There also exists binary encoding for JSON (e.g. BSON, Smile, UBJSON).

Another human-readable serialization format is YAML (a super-set of JSON). The main features of this format includes tagging data types, support for non-hierarchical data structures, data structures with indentation, and multiple forms of scalar data quoting.

Table 1 presents a comparison of textual data serialization formats [6].

2.2 Binary Serialization

In addition to textual formats, several binary data interchange formats have been proposed over the last decade in order to address the verbosity and the efficiency limitations of widely-accepted text-based formats such as XML and JSON [5, 4]. Among these formats we highlight the Apache Thrift, Apache Avro and the Google Protocol Buffers. Each of these protocols uses a custom Interface Description Language (IDL) to specify the structure of the exchanged data.

Google Protocol Buffer, or protobuf, is an extensible way (regardless of platform/language) for serializing structured data for use in communication protocols, data storage, among others. Protobuf is used at Google to encode structured data in binary format for implementing smaller and faster serialization. The implementation of a strategy using the protobuf format follows the sequence:

1. Definition of the schema file for the structured data;
2. Compilation of the file for generation of access classes;
3. Use of the programming language API for reading and writing messages.

After the schema definition is stored in a file (.proto), we use the protobuf compiler to generate the data access classes. These classes provide accessors for each field, methods to serialize and deserialize data and special builder classes to encapsulate internal data structure. Listing 1 presents an example of a protobuf schema that defines the shopping item entity.

Listing 1 The protobuf schema.

```
package com.example.protobuf.model;
option optimize_for = LITE_RUNTIME;
option java_package = "com.example.protobuf.model";
option java_outer_classname = "Shopping";
message Item {
  required string name = 1;
  required string category = 2;
  optional int32 quantity = 3 [default = 1];
  enum status {
    BOUGHT = 1;
    CANCEL = 2;
  }
  message Provider {
    required string name = 1;
    required float price = 2;
  }
  repeated Provider providers = 4;
}
```

Listing 2 shows how to build a new protobuf object to an item. You start by creating a new *Builder* for the specific object you want to build and then sets up the desired values, and finally, we use the *Builder.build()* method to create an immutable protobuf object (object item). The *Item object* is then serialized to an *OutputStream*.

Listing 2 The protobuf serialization.

```
public void writeToStream(
String name, String cat, int qt, Shopping.Item.Status status,
List<Shopping.Item.Provider> providers,
OutputStream os) throws IOException {
  Shopping.Item.Builder builder = Shopping.Item.newBuilder();
  builder.setName(name);
  builder.setCategory(cat);
  builder.setQuantity(qt);
  builder.setStatus(status);
  if (providers != null)
    builder.addAllProviders(providers);
  Shopping.Item item = builder.build();
  item.writeTo(os);
}
```

Listing 3 shows how to deserialize a protobuf object from an *InputStream*.

Protobuf has a lite version for Java suitable for Android. Protobuf has more limited language reach compared to JSON or XML. Officially, Google only provides compilers for C++, Java and Python.

Thrift is a binary communication protocol. Although developed at Facebook, it is now an open source project of the Apache Software Foundation. The currently supported programming languages are C++, Java, Python, PHP, Ruby, Erlang, Perl, Go, Haskell, C#,

Listing 3 The protobuf deserialization.

```
public Shopping.Item readFromStream(InputStream is) {
 Shopping.Item item;
 item = Shopping.Item.newBuilder().mergeFrom(is).build();
 Log.d("ProtobufDemo", "Read item name: " + item.getName());
 return item;
}
```

Table 2 Binary serialization formats.

Name	Date	Creator	Based on	Schema/IDL	Human-Readable
Avro	2009	ASF	n/a	yes	no
BSON	2003	MongoDB	JSON	no	no
Cap'n Proto	2013	Kenton Varda	protobuf	yes	no
Protocol Buffers	2008	Google	n/a	yes	no
Thrift	2007	Facebook/Apache	yes	yes	no

Cocoa, JavaScript, Node.js, Smalltalk, and OCaml. Similarly to the protobuf format, we need to prepare a schema definition as input for the code generation tool generates source code for a specified programming language. A typical thrift schema representing a phone object is presented in Listing 4.

Listing 4 The thrift schema.

```
enum PhoneType {
 HOME,
 OTHER
}
struct Phone {
 1: i32 id,
 2: string number,
 3: PhoneType type
}
```

Apache Avro is a serialization framework. It uses JSON for defining data types and protocols, and serializes data in a compact binary format. Its primary use is in Apache Hadoop, where it can provide both a serialization format for persistent data, and a wire format for communication between Hadoop nodes, and from client programs to the Hadoop services. It is similar to Thrift, but does not require running a code-generation program when a schema changes. The currently supported programming languages are Java, Scala, C, C++, C#, Python and Ruby.

Table 2 presents a comparison of binary serialization formats [6].

Choosing textual or binary data formats often depends on the context in which thay are used. Text-based formats (XML, JSON) are parsed character by character, thus imposing a limit on deserialization speed. On the other hand, binary formats make use of positional binding which allows storing the"name part of the name-value pairs in a separate file (e.g., '.proto' for ProtoBuf). These files do not need to be sent over the Web, which decrease the size of the data to be communicated. However, since these files have to be compiled before being included in a program, there are restrictions based on what languages each protocol supports.

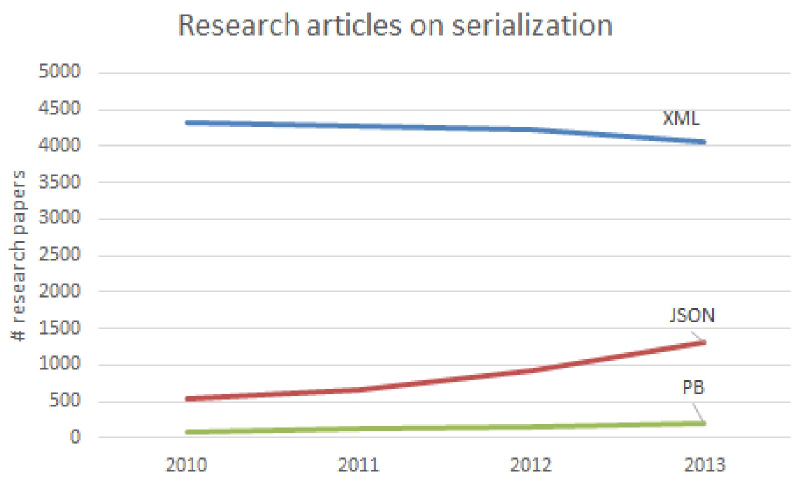

Figure 1 Research articles about serialization on Google Scholar.

2.3 Selection of a Serialization Format

In this subsection, we present the criteria used for the selection of the serialization format that will be used in the benchmark. Several criteria could be used to select a serialization format: the most popular, the most used among existent Web services, the one used in most popular applications, and others. In this case we decide that the selection will be based on the research papers found in the freely accessible Web search engine Google Scholar. This search engine indexes the full text of scholarly literature.

Figure 1 shows a comparison of the three most cited serialization formats on the Google Scholar website.

Based on the values presented in Figure 1 and, despite the XML format being the most cited in research articles, is the JSON format that has the highest growth in recent years. For this reason we decided to use JSON for the benchmark tests in the next section.

3 Comparison and Benchmark of JSON Libraries

In this section we compare the performance of several JSON parser implementations. The purpose of this benchmark is only to ensure a reasonable reading and writing performance compared to other parsers. It is obvious that the performance depends on several factors such as the used operating system, the programming language and network signal. All this just to say that the benchmark results may be misleading – if you want to infer results for a concrete case it is better to produce your own tests, with your custom data on your own hardware.

3.1 Setup and Methodology

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

- Hardware: ASUS Padfone with 1.5 GHz dual-core Qualcomm Krait and 1 GB memory
- Operating System: Android version 4.1.1
- Java: version 1.6.0

The test object used for this experiment is a JSON object obtained from a weather service called OpenWeatherMap. This service is often used in order to present a description of the weather of a given city. A request to the REST service returns OpenWeatherMap meteorological data of a certain city (set in the request) in JSON format. For instance, this is a typical URL request: `http://api.openweathermap.org/data/2.5/weather?q=porto`. The service returns the output in JSON format presented in listing 5.

Listing 5 OpenWeatherMap meteorological data.

```
{"id":88319,"dt":1345284000,"name":"Porto",
    "coord":{"lat":41.15,"lon":-8.61},
    "main":{"temp":306.15,"pressure":1013,"humidity":44,
    "temp_min":306,"temp_max":306},
    "wind":{"speed":1,"deg":-7},
    "weather":[
                 {"id":520,"main":"Rain",
                    "description":"light intensity shower rain",
                    "icon":"09d"},
                 {"id":500,"main":"Rain",
                    "description":"light rain","icon":"10d"},
                 {"id":701,"main":"Mist",
                    "description":"mist","icon":"50d"}
              ],
    "clouds":{"all":90},
    "rain":{"3h":3}}
```

The JSON libraries are selected based on their popularity. Tested libraries and their versions are the following: Gson (2.2.4), Jackson (2.2.1), Minimal-json (0.9.1) and org.json (n/a).

The experiment was designed as follows:

1. 100 iterations were executed for warming-up, and then 100 iterations were executed for measuring.
2. The execution time was measured using *System.currentTimeMillis()*.
3. Finally, the average execution time is taken for each operation and library.

3.2 Performance Benchmark

While mobile devices are becoming more powerful, they still lack the processing speed of desktop PCs. Despite this fact, it is essential that the chosen data serialization format allows fast serialization and deserialization of an object. For the performance comparison of the JSON libraries previously enumerated, we compared the time required to read and write a typical weather message with the parser implementations. The results are presented in Figure 2.

Our conclusion is that when you need to serialize/deserialize Java POJOs without sacrificing performance you should choose Jackson [2]. Although minimal-json cannot outperform Jackson's writing performance, it offers a very good reading and writing performance.

4 Conclusions

This paper presented a comparison on the use of a set of JSON libraries within a mobile application. When comparing serialization libraries on a mobile platform, it is necessary to

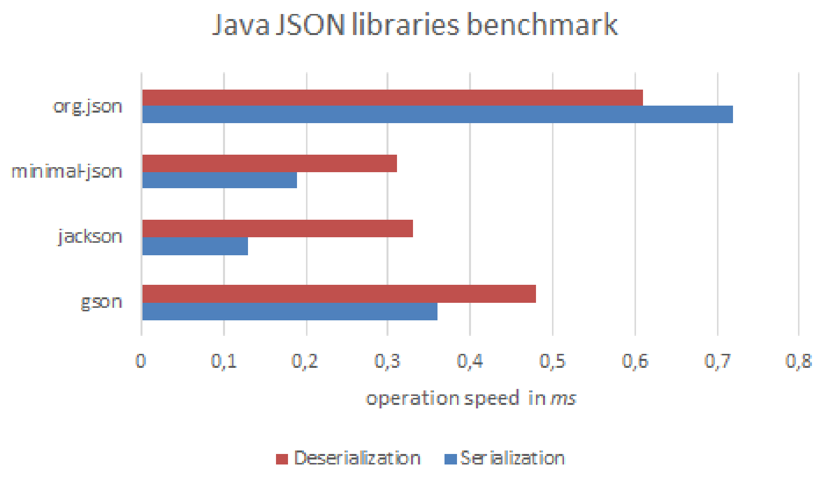

Figure 2 Java JSON libraries benchmark.

consider the most important aspects for this environment, such as data size and serialization speed. In this paper we focus on the performance facet.

The main contribution of this paper is two-fold: a survey on serialization formats organized by types: textual and binary; a performance benchmark that could be important for others that need to select an efficient parser for mobile communication.

Based on the benchmark results one can conclude that Jackson showed the best combined results. However, if your mobile app will only deserialize data, minimal-json offers the best performance in the experiment.

References

1 T. Bray. RFC 7159 – the javascript object notation (json) data interchange format. `http://tools.ietf.org/html/rfc7159`, 2014. [Online; accessed 06-May-2014].

2 Codehaus. High-performance json processor. `http://jackson.codehaus.org/`, 2013. [Online; accessed 06-May-2014].

3 Erik Hellman. *Android Programming – Pushing the limits*. Wiley, 2013.

4 K. Maeda. Performance evaluation of object serialization libraries in XML, JSON and binary formats. In *Digital Information and Communication Technology and its Applications (DICTAP), 2012 Second International Conference on*, pages 177–182, May 2012.

5 Audie Sumaray and S. Kami Makki. A comparison of data serialization formats for optimal efficiency on a mobile platform. In *Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication*, ICUIMC'12, pages 48:1–48:6, New York, NY, USA, 2012. ACM.

6 Wikipedia. Comparison of data serialization formats. `http://en.wikipedia.org/wiki/Comparison_of_data_serialization_formats`, 2014. [Online; accessed 15-April-2014].