Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 56 (2015) 284 – 291

Procedia Computer Science

The 10th International Conference on Future Networks and Communications (FNC 2015)

A lightweight data interchange format for Internet of Things in the PalCom middleware framework.

Mattias Nordahl[a,*], Boris Magnusson[a]

[a]*Lund University, Computer Science, Lund, Sweden*

Abstract

We present the PalCom Object Notation, a textual data representation format for communication between internet of things which support binary and textual data. The format does not require parsing of user data (or the "payload") and is thus efficient to use also for large binary values such as digital images, audio and video as well as for short textual values. These can be mixed in the same messages and thus transported over the same communication link. Its structure is influenced by JSON, making it easy to translate between the two formats. The evaluation show that its size and processing efficiency is comparable to that of JSON for small data, but becomes both smaller and more efficient as data grows, and can yield a tenfold performance increase for binary payloads.

Peer-review under responsibility of the Conference Program Chairs

Keywords: Lightweight protocol; Data interchange format; PON; JSON; Textual format; Efficient binary handling; PalCom; Java

1. Introduction

In this paper we introduce the PalCom Object Notation (PON), a lightweight format for data interchange developed for use in PalCom. PalCom is a pervasive middleware framework implemented in Java that aims to simplify the creation of dynamic networks between devices in distributed systems, and to give an easy and intuitive

* Corresponding author. Tel.: +46-46-2220000
E-mail address: Mattias.Nordahl@cs.lth.se

Peer-review under responsibility of the Conference Program Chairs
doi:10.1016/j.procs.2015.07.210

way for combining services provided by the devices.[1] It allows users to combine services from any device on the network into larger units with more complex functionality. PalCom has automatic device and service discovery, as described by a discovery protocol, which entails data of various types being sent between devices. Communication between services themselves are governed by a service-interaction protocol, but what information are sent with it are up to the service implementer. There is thus a need to be able to send a mixture of types of information (text, structured information, binary information such as images, audio and video) between devices in a general and homogeneous way.

This is exemplified well in the itACiH project, which develops, on top of PalCom, a system to support advanced care in the home with support for visiting nurses, on-line equipment in the patients' home, and process support at hospital wards. Previously all PalCom protocols has been represented in XML[3], with only limited type support for service communications, as everything was represented as strings. One part of the itACiH project is a web-server with PalCom as backend[8], in which JSON[2] has been used for communication with clients, which gives direct support for different data types. But as the project progressed a requirement arose for being able to send binary information as well. However, JSON, just as XML, lack the direct ability to represent such data, and must resort to special encodings which increase their size and adds extra processing.

This motivated the development of a new compact, textual format, similar to JSON, but which could directly handle binary information. In order to outline the design space we take examples from early language design which also illustrate that these problems are nothing new.

1.1. String handling in FORTRAN

Handling of strings in Standard FORTRAN[4], Hollerith constants, was an important influence in designing the PalCom Object Notation. Its format statement was extended to include strings:

```
FORMAT(13H HELLO, WORLD)
```

In this format the programmer was to count the number of characters in the string and give the compiler the length before the 'H' which indicated the string type. Although the manual counting could be a problem, this format had the advantages that 1) the FORTRAN complier did not need to parse the string itself, but simply copy the length of characters, and 2) the string could contain any character in the character set, with no exception for a special ending delimiter. When computers generate the information, the counting of characters ceases to be a problem, while the two advantages remain.

In FORTRAN 77[5] the modern notation for representing strings was introduced:

```
FORMAT('HELLO, WORLD')
```

Here the compiler had to scan the otherwise uninteresting string in order to count the characters, and the string could not contain single quotes.

In situations where hardware are to interpret data, such as in CPUs and network equipment, it is common to use fixed format layouts, as illustrated by the header part of IPv4 packets[6] in Figure 1. Also, here the length (in this case of the packet with payload) is given explicitly as in Standard FORTRAN, but note that the "Total Length" attribute is limited to 16 bit numbers and the address fields are limited to 32 bits. Although practical in situations with dedicated hardware, such limitations create problems over time when the length of the fields become too short and new extended protocols such as IPv6 needs to be introduced. It is thus wise to avoid fixed formats and limited lengths of size fields.

IPv4 Header Format

Offsets	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Version				IHL				DSCP						ECN		Total Length															
4	32	Identification																Flags			Fragment Offset											
8	64	Time To Live								Protocol								Header Checksum															
12	96	Source IP Address																															
16	128	Destination IP Address																															
20	160	Options (if IHL > 5)																															

Figure 1. IPv4 header.

More recent formats, geared towards human readability and easy construction, such as XML and JSON, have chosen representations in line with the FORTRAN 77 format, with explicit delimiters around strings. This notation has the drawback that all user data need to be parsed in order to understand the structure of the information. Additionally, binary information, e.g. images, which are often large by comparison (several MB) needs to be converted to readable text (e.g. through base64 encoding) before inclusion in such messages which increase their size by 1/3, adds more work for the programmer and slows down generation and parsing of messages.

With PON we set out to define a format for structured data that combines the structure and compactness offered by JSON with more efficient handling for both textual and binary data.

1.2. Previous work

Since JSON was popularized, several new formats have been designed that borrow from or build upon its structure, but simultaneously try to improve certain aspects where JSON falls short. The perhaps most apparent shortcoming of JSON, as suggested by the numeral binary JSON-like formats, is its lack of direct support for handling binary information. BSON[11], BJSON[12] and Universal Binary JSON[13] are all examples of such formats.

These formats share the general structuring of data with JSON, but are fully binary rather than textual, which enabled them to makes use of byte or bit manipulation to greatly decrease the number of bytes needed to represent the same data and to speed up processing. Like in Standard FORTRAN, these formats include the length of strings and other data types before their actual value, meaning that they do not need to parse the data itself, looking for an ending delimiter.

2. The PalCom Object Notation

The PalCom Object Notation is also a JSON-like format, and can easily be translated to and from it. Like FORTRAN and the previously mentioned binary JSON-like formats, each value is prepended with its own length. Unlike those formats, however, PON is a fully textual format, like JSON, and can therefore be seen as mix of the two variations. Due to knowing the length of values and thus not needing to parse them, PON can directly handle binary information, while still maintaining a textual and thus readable format.

2.1. PON structure and node syntax

The overall structure of PON is borrowed directly from JSON, and data is thus organized by objects and arrays, where objects are lists of key-value pairs and arrays are list of values.[2] Objects and arrays are themselves also considered values, which enables nesting. In PON all values are represented as nodes that consists of three fields; the length of the value, a type identifier and the actual data value itself:

<Length><Type><Data>

Length is an integer stating the length of the data, the type identifier is a one byte ASCII character that determines the value type and on which the format of the data field depend. The three fields are written in sequence without spaces or delimiters, as are all nodes themselves. For instance, three consecutive string values are written:

15sMy first string16sMy second string15sMy third string

2.2. Objects, arrays and keys

Since objects and arrays are also values they too follow the node structure. An array is a list of values, and its data field is thus a list of nodes, written in sequence, and their combined length is that of the array's length field. Similarly, objects are lists of key-value pairs. Keys are the only exception to the node structure, instead being more closely represented as they are in JSON, i.e. as a string ending in a special delimiter (colon).

This is motivated by keys typically being short, and would thus provide little to no speed increase for not needing to be parsed. Also, not having to include the node length and node type slightly reduces the size of their representation and generally makes the format more compact since keys are often numeral.

As an example, an object and an array, each containing two strings can be written:

26{key1:6svalue1key2:6svalue2

16[6svalue16svalue2

2.3. Conserving compactness

Though the node structure does provide performance benefits, not having to parse the payload, it does come with an added overhead (length and type) which increases the representation's size. Especially for short data values, this overhead can make up a significant percentage of the entire node. To compensate for this and to keep PON's compactness comparable to that of JSON, we have shortened the representation in other areas. For value types of fixed length, the length field of the nodes is omitted; this applies to single characters and bytes. For value types with a predetermined set of values, and no actual payload, the data field of the nodes is omitted as well, thus representing them as a single ASCII character; this applies to boolean and null.

The types that are supported by PON are meant to directly mirror those of modern programming languages - in particular Java. All the supported types are listed in Table 1, together with their respective type identifiers, their equivalent type in Java and a short example.

Table 1. Types supported by PON and their type identifiers.

PON type	Java type	Length	Type identifier	Data	Example
string	String	yes	s	yes	16sThis is a string
integer	Integer or Long	yes	i	yes	2i49
float	Float or Double	yes	d	yes	4d17.3
binary	byte[]	yes	y	yes	412y<Data> where <Data> is binary data of length 412 bytes
char	Character	no	c	yes	cA for the single character 'A'
byte	Byte	no	b	yes	bA for the ASCII character 'A'
boolean	Boolean	no	t or f	no	t for true, f for false
null	null	no	u	no	u for null
object	Map<String, Object>	yes	{	yes	26{key1:6svalue1key2:6svValue2
array	List<Object>	yes	[	yes	16[6s;value16svalue2

3. Analysis and performance testing

We evaluate PON through a comparison with JSON, first by analyzing their size for different data types, then comparing their readability, followed by a performance test in which both formats generate and parse data taken from real use of the PalCom framework.

3.1. Size analysis

We can examine the size difference of the two formats by looking at their overhead, by which we here will mean the extra number of characters needed for their format in addition to the payload itself. For objects, arrays and strings JSON adds two characters ({}, [] and ""), whereas PON adds one ({, [and s) plus the number of digits needed to represent the length of the data. For short values (length 0–9) the two formats are thus equal in size. For increasingly long values, PON adds more overhead than JSON. However, the added overhead from the length field is roughly given by the logarithm of the length, which becomes tiny in comparison as the length increases, and thus insignificant for large values. For instance, a value in the size range of a few MB (millions of characters), e.g. an image, the length field will be seven digits. For binary values, PON's overhead is equal to that of strings, whereas JSON has no way of representing it. If base64 encoded and represented as a string, the overhead of JSON becomes 2 plus 1/3 of the length, greatly outweighing that of PON. For numeric values PON's overhead is equals to that of strings, whereas JSON has none. Data containing a large number of numeric values may thus not be as compact in PON as in JSON. For all other types (char, byte, true, false, null) PON's representation is significantly shorter, since they are all represented by one or two characters, as was shown in Table 1.

For messages with largely string or numeric values, we would thus expect a somewhat comparable size for the two formats, whereas for messages with a mix of values or few string and numeric values PON should be noticeably smaller.

With regards to generation and parsing speed, we would expect comparable performance from the two formats for short to medium data values, whereas for larger values (long strings or binary data) PON should be noticeably faster due to not having to parse the payloads.

3.2. Readability and human interaction

Being a textual format, PON is also human readable. However, compared to JSON, some of its readability is lost in favor of compactness and performance. This is largely due to nodes being written without spaces or separators and having no ending delimiters. The beginning of, e.g. an object, is marked by a length and a { character, but there is no way to know where the object ends, and thus where the next node begin, apart from counting its length characters forward. For short values this is easy to do, even for humans, but for long or nested values it is not.

The node lengths also makes it hard to write PON representations by hand, since the length of inner values affect the length of their containers. One must thus start with the length of the innermost values of nested structures and recursively sum their lengths to calculate the length of their containers. This makes PON a bad choice for some tasks where JSON might be used, e.g. configurations files. PON is designed as a data interchange format between devices.

3.3. Performance test

For the performance test we took various example data from real usage of the PalCom framework, and had both formats generate their representations of it and parse it back into its original form. We measure the size of their respective representations for the same data as well as their generation and parsing speed. Each measurement was run separately, with care taken to only include actual generation and parsing time in the measurement, a three second warmup for letting the virtual machine load necessary classes and with 100000 iterations to get consistent average values. For the JSON format, we ran the test using two popular JSON libraries that we have seen perform well in benchmarks[14, 15, 16], Gson[9] and Jackson[10].

The test was run with data from three distinct categories:

- A PalCom *ServiceListRequest* message, which is a short message that consists of two string values and one integer.
- A PalCom *ServiceListReply* message, which consists of a list of descriptions of, in this case, fifteen services, each of which has several string, integer, byte and boolean parameters and some null values as well as a few nested structures with similar value type.
- A PalCom service command from the itACiH project for fetching an image. It consists of a couple of string and integer values representing information about the image, as well as the image itself (binary data). We include measurements for two such messages, with image sizes 50 kB and 500 kB (larger image sizes yielded roughly the same results). For JSON the binary image data was base64 encoded (not included in time measurements).

Test results for these messages are shown in figures 2, 3, 4 and 5 respectively.

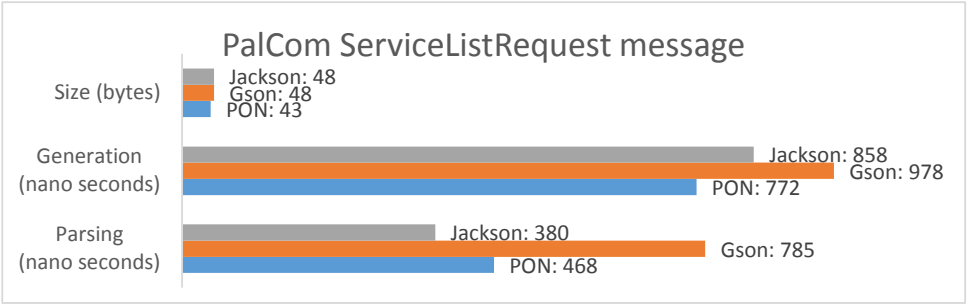

Figure 2: Shows size, generation time and parsing time for a ServiceListRequest message. PON is slightly more compact than JSON and is faster than Gson and roughly equal to Jackson (faster generation but slower parsing).

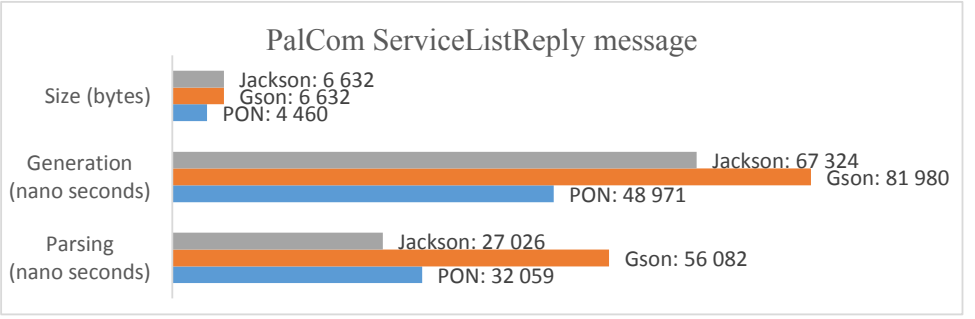

Figure 3: Shows size, generation time and parsing time for a ServiceListReply message. PON is significantly more compact than JSON and in total faster than both Gson and Jackson.

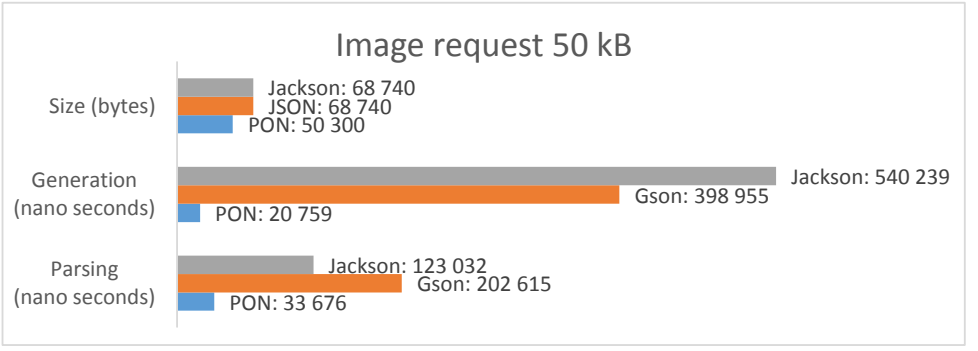

Figure 4: Shows size, generation time and parsing time for a 50 kB image request. PON is significantly more compact than JSON and greatly outperforms both Gson and Jackson.

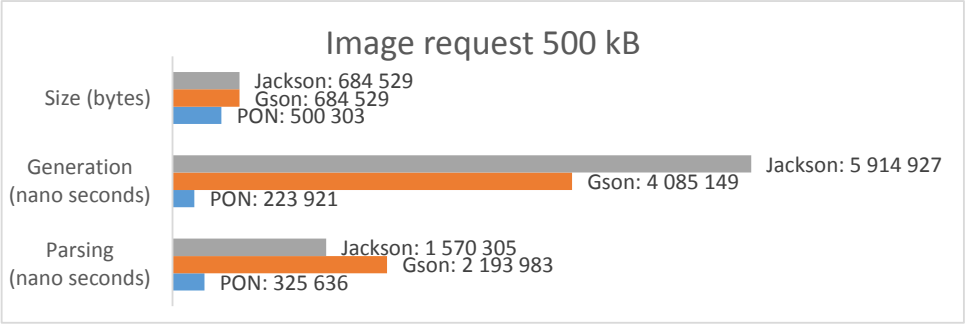

Figure 5: Shows size, generation time and parsing time for a 500 kB image request. PON is significantly more compact than JSON and greatly outperforms both Gson and Jackson.

3.4. Results

From the results of the short PalCom service message we see that the PON representation is just slightly shorter than JSON. We also see that PON is noticeably faster than Gson and about equal in performance to Jackson; slightly faster generation and slightly slower parsing (a net difference of 2 nanoseconds). If we divide the difference of their total times (generation plus parsing) with the total time of PON, we calculate that PON is more or less equal to Jackson and 42 % faster than Gson.

The longer PalCom service message shows similar results, but starts swinging more in favor of PON. The JSON representation is roughly 1/3 longer than the PON representation and we calculate that PON is 16 % faster than Jackson and 70 % faster than Gson.

Finally, for the image command with binary data we see a large size difference due to JSON being base64 encoded and a substantial performance advantage for PON. For the 50 kB image, PON is twelve times as fast as Jackson (1100 % increase) and eleven times as fast as Gson (1000 % increase). For the 500 kB image the difference between PON and the two JSON libraries grows slightly larger still.

Note again, that the time for base64 encoding the data was not included in the measurements, but some simple measuring using the 50 kB image data suggested that it could slow down the JSON libraries with an additional 70 % (300 000 nanoseconds to encode, 170 000 nanoseconds to decode).

3.5. Comment

Although PON can handle binary information directly, it is of course possible, should it be necessary, to handle it like JSON; i.e. by base64 encoding the data and representing it as a string. In this case, PON still provides faster processing, since the payload does not need to be parsed. Re-running our test for PON with the 50 kB image data, now base64 encoded, resulted in a total processing time of 120278 nanoseconds, still 450 % faster than Jackson and 400 % faster than Gson.

4. Conclusions

We have presented a new data interchange format PON, which supports combinations of text and binary information to be included in the same message. Like JSON, it is a textual, compact format and have direct support for a number of value types. PON supports some additional value types, being closely coupled with the types of the programming language Java, and in particular supports direct handling of binary information.

The performance when generating or parsing PON representations is at least comparable to popular, efficient JSON implementations and for large data values and binary information it outperforms them with a factor of ten or more.

Also the representation is more compact with PON, as observed using test data from IoT systems. For longer mixed type messages, and in messages with binary information, a difference of 25–30 % has been observed.

PON is designed for IoT situations where communication is between devices, but is still textual and thus human readable, apart from when used with binary payloads. It borrows its structure from JSON and can easily be translated to and from it.

References

1. D. Svensson Fors, B. Magnusson, S. Gestegård Robertz, G. Hedin, and E. Nilsson-Nyman. Ad-hoc composition of pervasive services in the palcom architecture. In Proceedings of the 2009 international conference on Pervasive services, ICPS '09, pages 83–92, New York, NY, USA, 2009. ACM.
2. RFC 7159 – The JavaScript Object Notation (JSON) Data Interchange Format.
3. XML Media Types, RFC 7303. Internet Engineering Task Force. July 2014.
4. Ansi x3.9-1966. USA Standard FORTRAN. American National Standards Institute. Informally known as FORTRAN 66..
5. Ansi x3.9-1978. American National Standard – Programming Language FORTRAN. American National Standards Institute. Also known as ISO 1539-1980, informally known as FORTRAN 77
6. RFC 791 – Internet Protocol (IPv4)
7. Fielding, Roy T.; Gettys, James; Mogul, Jeffrey C.; Nielsen, Henrik Frystyk; Masinter, Larry; Leach, Paul J.; Berners-Lee (June 1999). Hypertext Transfer Protocol -- HTTP/1.1. IETF. RFC 2616.
8. Thomas Sandholm, Boris Magnusson, Björn A Johnsson: The Palcom Device Web Bridge, Technical report LU-CS-TR:2012-251, ISSN 1404-1200, Report 100, 2012.
9. Google Gson library, https://github.com/google/gson
10. Jackson JSON library, http://wiki.fasterxml.com/JacksonHome
11. BSON, http://bsonspec.org/
12. BJSON, http://bjson.org/
13. Universal Binary JSON, http://ubjson.org/
14. The Ultimate JSON Library: JSON.simple vs GSON vs Jackson vs JSONP, http://blog.takipi.com/the-ultimate-json-library-json-simple-vs-gson-vs-jackson-vs-json/ (accessed 2015-06-05)
15. jvm-serializers, https://github.com/eishay/jvm-serializers/wiki (accessed 2015-06-05)
16. Top 7 Open-Source JSON-Binding Providers Available Today, http://www.developer.com/lang/jscript/top-7-open-source-json-binding-providers-available-today.html (accessed 2015-06-05)