Smart Grid Serialization Comparison

Comparision of serialization for distributed control in the context of the Internet of Things

Bo Petersen, Henrik Bindner, Shi You
DTU Electrical Engineering
Technical University of Denmark
Lyngby, Denmark
bspet@elektro.dtu.dk, hwbi@elektro.dtu.dk,
sy@elektro.dtu.dk

Bjarne Poulsen
DTU Compute
Technical University of Denmark
Lyngby, Denmark
bjpo@dtu.dk

***Abstract*—Communication between DERs and System Operators is required to provide Demand Response and solve some of the problems caused by the intermittency of much Renewable Energy. An important part of efficient communication is serialization, which is important to ensure a high probability of delivery within a given timeframe, especially in the context of the Internet of Things, using low-bandwidth data connections and constrained devices. The paper shows that there are better alternatives than XML & JAXB and gives guidance in choosing the most appropriate serialization format and library depending on the context.**

***Keywords*—Smart Grid; Internet of Things; Serialization; XML; JSON; YAML; FST; Kryo; JAXB; Jackson; XStream; ProtoStuff; Gson; Genson; SnakeYAML; MsgPack; Smile; ProtoBuf; BSON; Hessian; CBOR; Avro**

I. INTRODUCTION

In a future Smart Grid with a large share of Renewable Energy (EU 2020 & 2030 energy strategy), there will be problems caused by the intermittent nature of most Renewable Energy, especially solar and wind [1].

These problems primarily consists of times with either excess or lack of energy from renewable power sources.

Excess power will be wasted, transported to other regions or countries, stored or converted, all of which will cause a loss in energy.

Lack of energy will cause the use of more economically or environmentally expensive energy, in the form of non-renewable energy, bio-fuels or stored energy.

The most efficient solution to these problems, if done right, is Demand Response, which entails controlling consumption units, especially heating, cooling and production units.

In addition, control of production units, which have the capability to move their production can also help to solve these problems.

For the control of these Internet of Things Distributed Energy Resources (DER), both production and consumption units, communication between the units and the System Operators (Transmission System Operator, Distribution System Operator and Balance Responsible Party) is crucial.

The choice of technology for this communication (e.g. Web Services), called communication middleware is very important to ensure that the control messages are received within a given timeframe, depending on the needs of the power grid.

This need could be to avoid a fault, by initiating load shedding, with a timeframe of seconds to minutes, or moving the consumption of energy from peak hours, by initiating load shifting, with a timeframe from hours to days.

Communication in the scope of power system services lies between the physical hardware that is needed and basic communication protocols like TCP/IP, and the business logic in the form of control algorithms (fig. 1).

Another important part of ensuring that the control messages are received within the given timeframe is the choice of serialization format and library, which affects the size of the message and the serialization time.

Even though there is no guaranty of delivery for messages sent over the internet within a given timeframe as oppose to dedicated lines, the probability of delivery within the given timeframe is improved by reducing the size of the transmitted message.

Furthermore the serialization time becomes especially important to consider when the processing device of the DER is a System on Chip, for instance Beagle Bone [2] or Odroid [3], with limited processing capabilities, as this will also improve the probability of delivery within a given timeframe, because the sending and receiving devices will be able to process the message quicker.

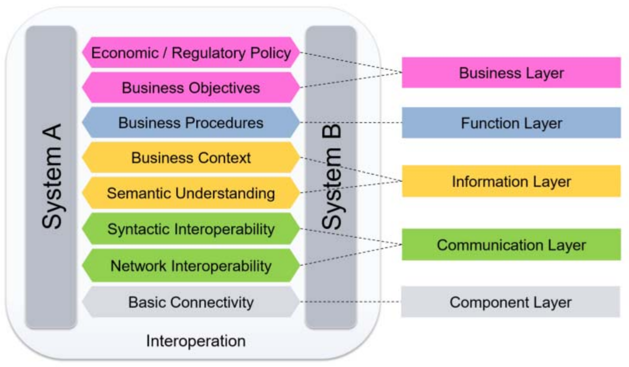

Fig. 1. CENELEC SGAM Model [28]

Sponsored by the PROActive INtegration of sustainable energy resources enabling active distribution networks (PROAIN) project

Moreover, in the case of a System on Chip with limited memory, the memory consumption has to be considered, to ensure that the control system can be executed without fault.

In cases were the DER is communicating over a low bandwidth data connection like EDGE (cell phone network) or Power Line Communication, the size of the message after serialization, and potentially also compression strongly affects the probability of delivery within the timeframe.

The choice of serialization format and library is often affected by the fact that most communication middleware uses a certain format and library, which is more of a convenience than a hindrance, as almost all communication middleware is capable of transmitting binary or text serialized messages.

In the area of power system communications, the choice made by prevalent communication standards should also be taken into account.

These standards are IEC 61850 [4], OpenADR [5] and CIM [6], which uses SCL (extension to XML), XML and RDF (extension to XML) respectively.

The current state of the art is online benchmarks for serialization formats and libraries, which does not take into account the requirements of the Smart Grid, the use of Smart Grid communication standards, the possibility of using compression after serialization, and does not give recommendations as to choosing a serialization format and library for the use in Smart Grid communications.

The hypothesis is that there are many better alternatives to using the XML format and the JAXB library for serialization in the context of Smart Grids, especially for applications with low bandwidth data connections and constrained processing & memory devices.

The aim of the paper is to give guidance in choosing the most appropriate serialization format and library for Smart Grid communications depending on the context, and to compare prominent serialization formats and libraries to the XML format used by the prevalent communication standards.

II. Methods

The scope of serializers for this paper has been limited to Java serializers, because most serializers are available in Java, and because Java can run cross platform.

The included serializers were chosen by searching online for all java serializers, sorting out the ones, with few users, which has not been updated for years, or are in early beta versions (based on MvnRepository.com).

In addition, serializers that require manual serialization or schemas that cannot be generated from source code, where excluded, as it would require too much implementation work for most real world cases (this primarily includes Thrift and the Protocol Buffers library).

Of the 26 serializers picked, two of them failed to work (YamlBeans & ProtoBuf (Jackson)).

The quantitative comparison of the serializers measures the following:

- Serialization time.
- Deserialization time.
- Compression time.
- Decompression time.
- Memory use for serialization.
- Memory use for compression.
- Serialized message size.
- Compressed message size.

With compression being performed after serialization, and using the GZip compression library.

Faster or more compact compression could be used, but because GZip is the default compression used in communication and because a comparison of compression formats and libraries is outside the scope of the paper, GZip is used to give an idea of the impact of using compression.

The times have been measured by first performing a warm up that serialize, compress, decompress and deserialize all test messages 1000 times, then measures the time it takes to serialize 1000 times and taking the average, and then doing the same for compression, decompression and deserialization.

The memory consumption is measured by requesting the execution of the garbage collection and then saving the memory consumption, after setting op the test objects, but before doing the 1000 runs, then requesting the execution of the garbage collector after 999 serialization runs, and saving the memory consumption after all 1000 runs, to get the memory held by the serializer for all runs plus the memory held during 1 run, which gives the peak memory consumption during 1000 runs if the garbage collection was as active as possible.

The times does not include initialization, because it only has to be performed on startup, and therefore will not affect the average serialization time of a message.

The test messages consists of IEC 61850 data model classes because it gives a good idea of the messages being transmitted for Smart Grid use cases, and because CIM does not specify fixed classes for energy systems as it can be used in many domains and OpenADR is a relatively new standard, and also does not exactly specify data model classes.

The IEC 61850 data model classes used are logical node classes, for which a unit uses one or more of them to describe its components, for instance the battery of an EV or a time schedule for production, used for measurement data and control commands respectively.

A logical node consists of many fixed classes, divided into 3 levels below the logical node in the hierarchy, so they can be relatively large.

For the tests all logical node classes specified in 61850-7-4 (2010) and 61850-7-420 (2009) are used.

The qualitative comparison includes serialization format and library characteristics for language neutrality, the required use of schemas or annotations and whether the serialized output is binary or text, but does not take into account whether version control is supported, as IEC 61850 specifies the version of all logical node classes.

The tests were run on Windows 10 (build 14393), using Java (Oracle 1.8.0_102 64bit), with an Intel Duel Core 2.1 GHz processor (i7-4600U), with 8 GB of memory.

The results for one serializer relative to the other serializers should be the same on any system, as long as the system does not run out of memory.

III. Results

The included serialization formats consists of java specific binary formats (Java Serialization API (JSA) [7], Fast-serialization (FST) [8] and Kryo [9]), human readable text formats (XML [10] [11] [12] [13], JSON [14] [15] [16] [13] [17], YAML [18] [19]), and language neutral binary formats (MsgPack [20] [21], Smile [22] [13], ProtoBuf [13], BSON [23], Hessian [24], CBOR [25], ProtoStuff [13] [26], Avro [27]).

These formats include multiple human readable text formats and multiple language neutral binary formats, which gives many options for choosing alternatives to XML and even includes two java specific binary format (Fast-serialization and Kryo) as alternatives to the built-in Java serialization API.

They also include formats that requires the use of schemas and/or annotations and without, many language neutral formats, the format used by prevalent communication standards (XML), and many popular serialization formats.

The libraries included are the ones needed for most of the formats, as they are single format libraries, and three multi format libraries (ProtoStuff, Jackson, XStream).

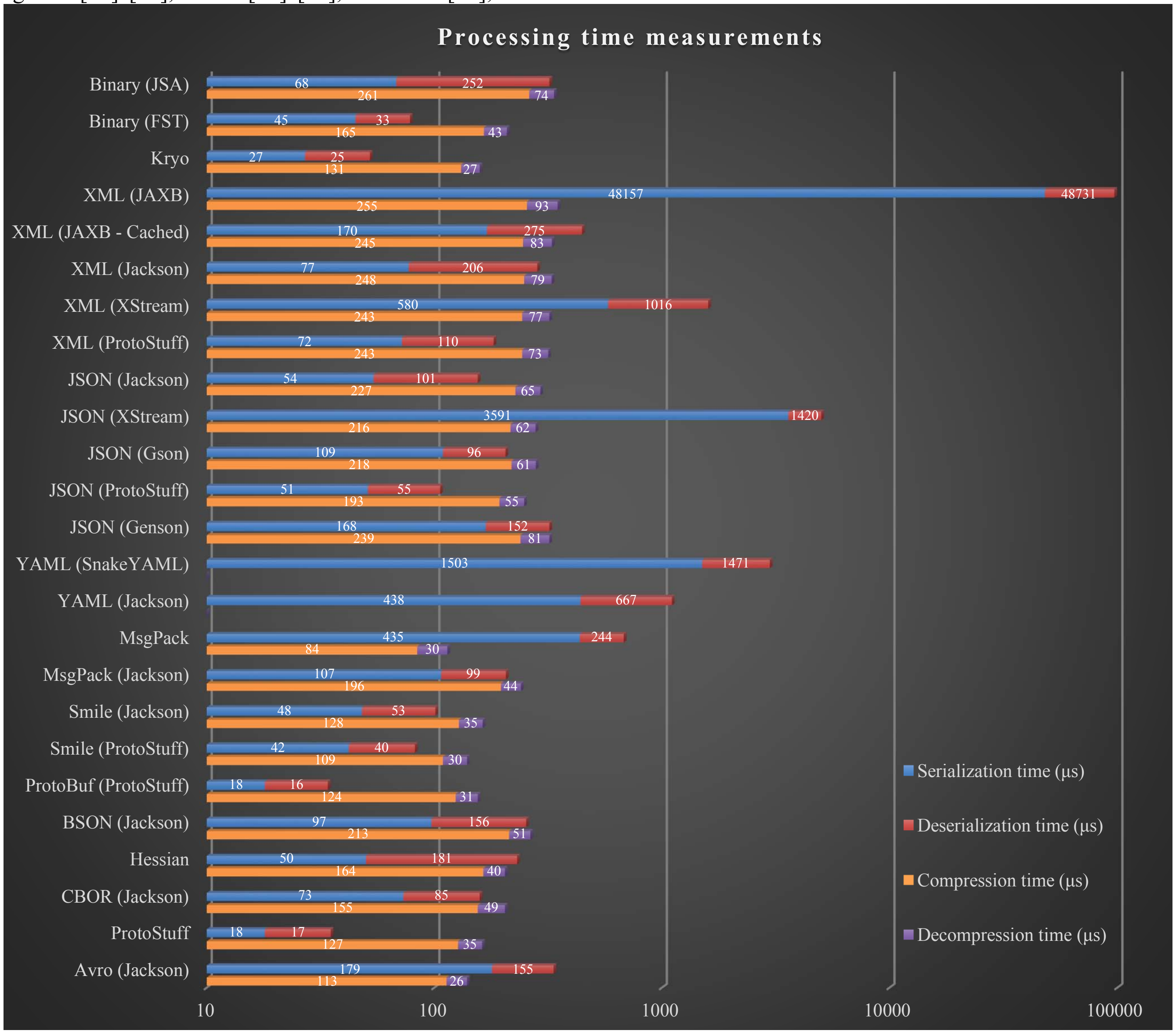

Fig. 2. Comparison of average processing time spent per message for serialization, deserialization, compression and decompression

The quantitative results of the comparison are the calculated average serialization, deserialization, compression and decompression times (seen in fig. 2), the serialized byte size and compressed serialized byte size (seen in fig. 3), and the memory consumption for serialization and compression (seen in fig. 4).

The JAXB serializer performs particularly bad when the context is not cached, which is why the performance has been measured both with cached context and without, which is a optimization and optimizations has not been performed for the other libraries.

A comparison of the XML format using the default java serializer JAXB with a cached context, and the most competitive serializers, based on size (Avro), speed (ProtoBuf-ProtoStuff, ProtoStuff), being human readable (Json-Jackson), and being java specific (Fast serialization), can be seen in fig. 5.

The results of the qualitative comparison, which includes the name, version, and library (if the library is not a single format library), whether the format is a human readable text format, whether the format enables the use of and/or requires a schema, annotations or inheritance, and whether the format is language specific or language neutral (seen in table 1).

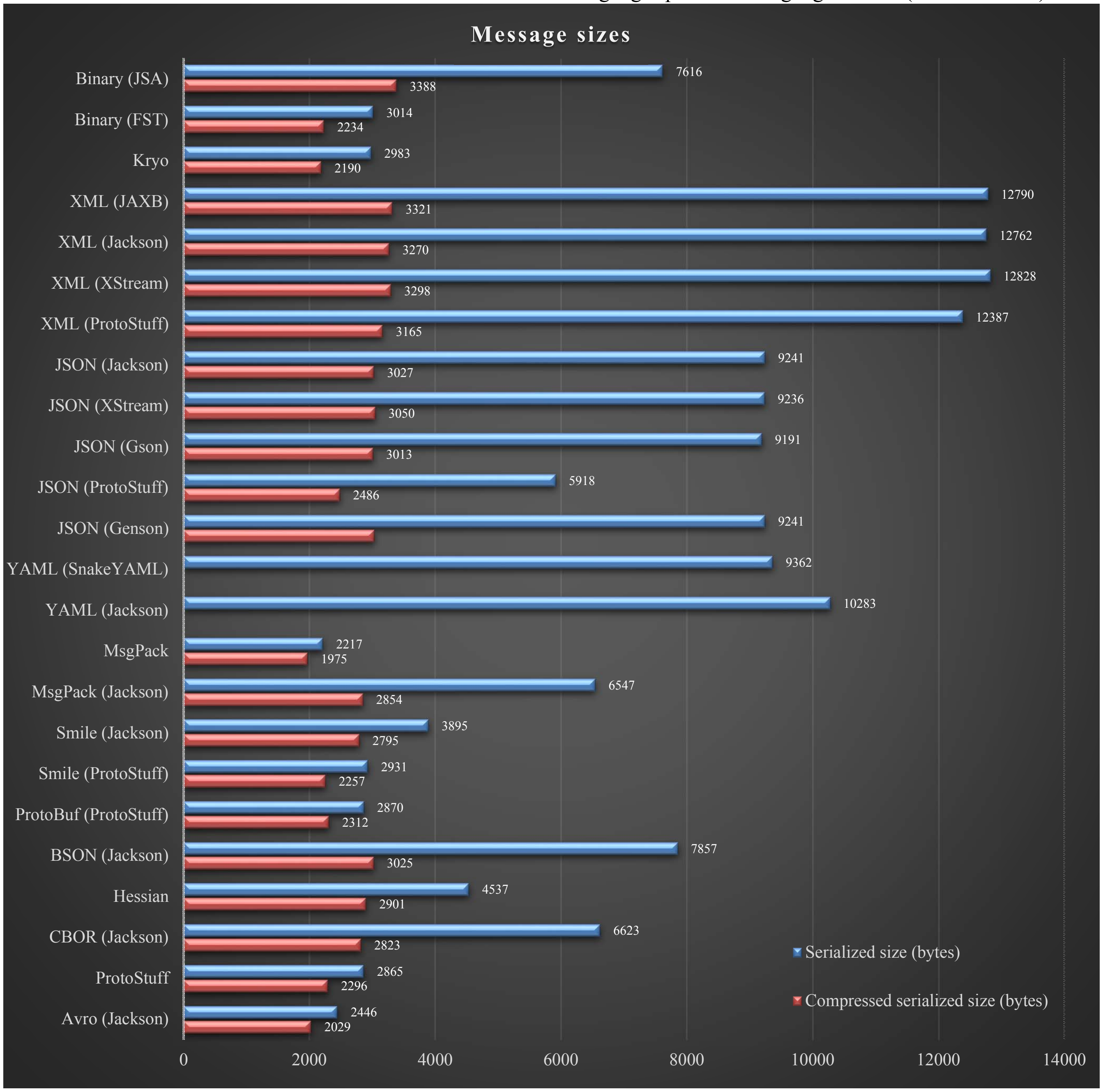

Fig. 3. Comparison of serialized size and compressed serialized size

978-1-5090-5443-5/17/$31.00 ©2017 IEEE

IV. Discussion

The first thing to consider when choosing a serialization format is whether the serialized output needs to be human readable text, and for instance with configuration files, the data often needs to be human readable so it can be changed in a text editor.

However, with Smart Grid communications, it mostly only needs to be human readable for debugging, which means that for most use cases it might as well be binary.

Another important thing to consider is whether the message will be compressed either by the communication middleware or before that, because depending on the chosen serialization it might affect the size of the message and the time it takes to serialize and deserialize, differently.

Moreover, it is important to use a communication middleware that does not serialize the message if it has already been serialized.

Note that even though the compressed serialized byte size is shown in fig. 3 for the human readable text formats (except YAML, which is problematic with compression because of the semantic use of whitespace), it mostly does not make sense to compress these formats, because it removes their primary characteristic, that they are human readable.

Memory consumption is important to consider when using a System on Chip for the Internet of Things, which in the case of a Beagle Bone Black only has 512 MB of memory, which is quickly exhausted by the operating system, and the control system.

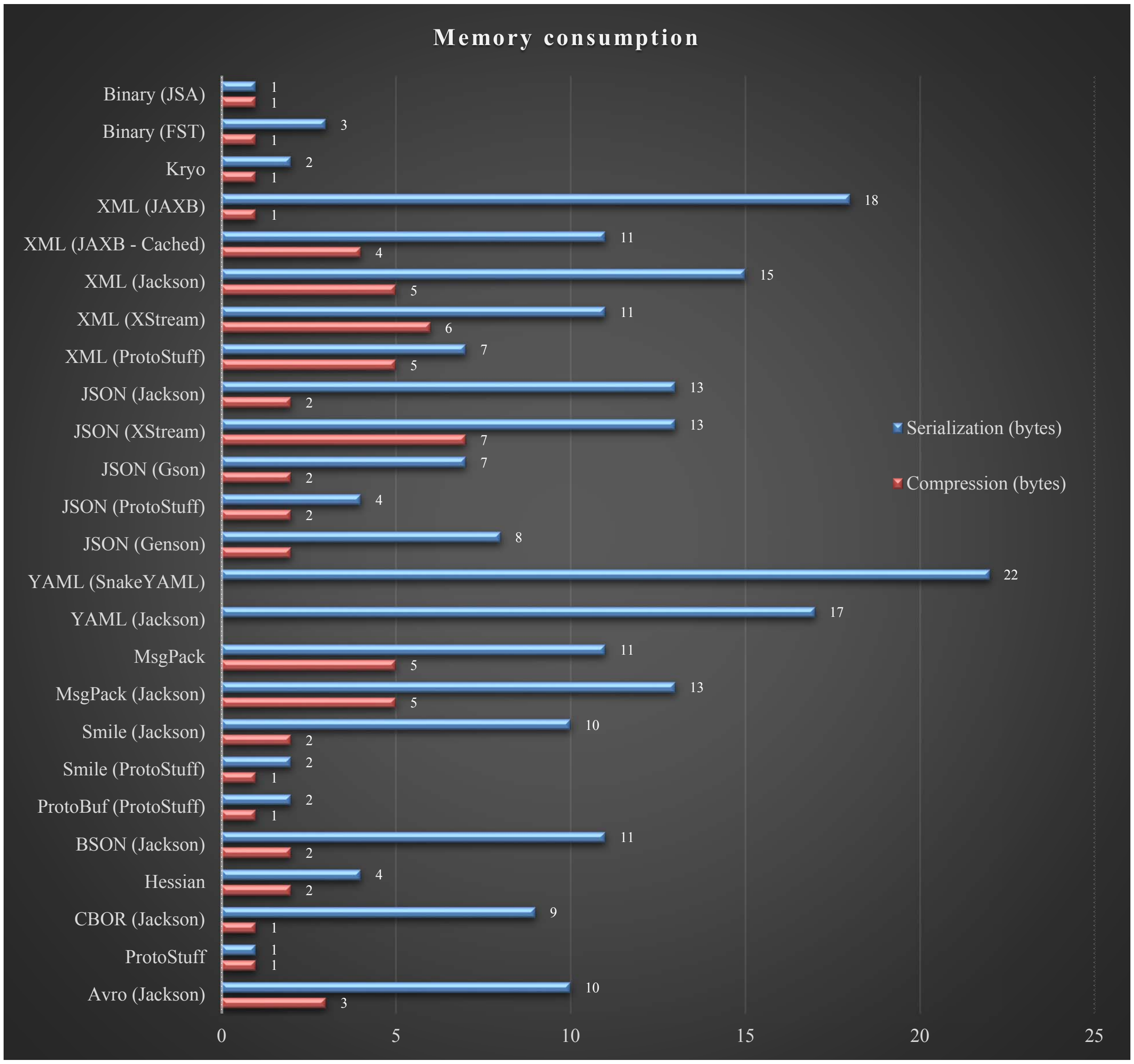

Fig. 4. Comparison of memory use

Looking at the quantitative result however, it can be seen that the memory used by the serializers range from 1 to 22 MB, with many using less than 5 MB. This should make it possible to choose a serialization format and library that can run on a System on Chip.

Even if the serialization format has already been chosen it is important to note that the speed of the serialization library might differ a lot, for JSON, it could be more than 40 times as long.

TABLE I. SERIALIZATION CHARACTERISTICS

Name (version) [(library)]	Serialization format / library characteristic		
	Binary / Text	*Schema / Annotations / Inheritance*	*Language neutral*
JSA (JDK 1.8.0_102)	Binary	Required Inheritance	No
FST (2.47)	Binary	Optional Annotate	No
Kryo (4.0.0)	Binary	Optional Annotate	No
XML (JDK 1.8.0_102) (JAXB)	Text	Optional Schema & Required Annotate	Yes
XML (2.8.1) (Jackson)	Text	Optional Schema & Annotate	Yes
XML (1.4.9) (XStream)	Text	Optional Schema & Annotate	Yes
XML (1.4.4) (ProtoStuff)	Text	Required or Generated Schema	Yes
JSON (2.8.1) (Jackson)	Text	Optional Annotate	Yes
JSON (1.4.9) (XStream)	Text	Optional Annotate	Yes
JSON (2.7) (Gson)	Text	Optional Annotate	Yes
JSON (1.4.4) (ProtoStuff)	Text	Required or Generated Schema	Yes[a]
JSON (1.4) (Genson)	Text	Optional Annotate	Yes
YAML (1.17) (SnakeYAML)	Text	Optional Annotate	Yes
YAML (2.8.1) (Jackson)	Text	Optional Annotate	Yes
MsgPack (0.6.12)	Binary	Required Annotate	Yes
MsgPack (0.8.8) (Jackson)	Binary	Optional Annotate	Yes
Smile (2.8.1) (Jackson)	Binary	Optional Annotate	Yes
Smile (1.4.4) (ProtoStuff)	Binary	Required or Generated Schema	Yes
ProtoBuf (1.4.4) (ProtoStuff)	Binary	Required or Generated Schema	Yes
BSON (2.7.0) (Jackson)	Binary	Optional Annotate	Yes
Hessian (4.0.38)	Binary	No	Yes
CBOR (2.8.1) (Jackson)	Binary	Optional Annotate	Yes
ProtoStuff (1.4.4)	Binary	Required or Generated Schema	Yes
Avro (2.8.1) (Jackson)	Binary	Optional Annotate	Yes

*The JSON like serialization format produced by ProtoStuff is language neutral but not compatible with other JSON serializers, because it uses property indexes instead of property names as keys

The differences between uncompressed serialized language neutral binary message sizes are more than 3 times as big, and the difference between speeds is more than 24 times as fast.

Between human readable serializers, the difference in speed is more than 70 times, and the difference in size could save more than 25 percent, which does not include the ProtoStuff library for JSON, because the way it saves a lot of space is by replacing property names with property indexes, which makes it incompatible with other JSON libraries.

For java specific serializers, Kryo is an impressive alternative to the Java Serialization API (JSA), with message sizes that are less than half as big for uncompressed messages, and 2.5 times as fast.

When the size of the messages are the most important thing, primarily with low-bandwidth data connections, Message Pack (MsgPack) & Avro produces uniquely small messages, but pays the price by being slower than most other language neutral binary serializers.

When it comes to speed, especially for constrained devices, Protocol Buffers (ProtoStuff), ProtoStuff, Kryo and FST perform particularly well and produce quite compact output.

Concerning memory, most serializers use little memory and it should therefore not be a problem, but some of them use much less memory than others, which in certain situations makes them a better choice.

Compression does make the message smaller, which for some use cases makes it worth using, but the price payed in processing time, is not worth it, for the most efficient serializers, in most cases.

The comparison of JAXB with the best serializers in different areas (fig. 5) shows that in every area there is a better choice, especially if a different format than XML is used.

When power system control messages are sent, it requires that measurements values have been received first by the controlling entity, which makes the message sizes used in the tests relevant, even though they are bigger than most control messages, they corresponds with the average size of measurement value messages.

The use of a schema for a serialization format, only helps to generate programming language code, which can be helpful, but not necessary, as the code can be created from documentation instead.

Schemas can also be generated from programming language code, if the serialization library has that feature, which makes it possible to move implementations of data classes from one programming language to a schema and then to another programming language.

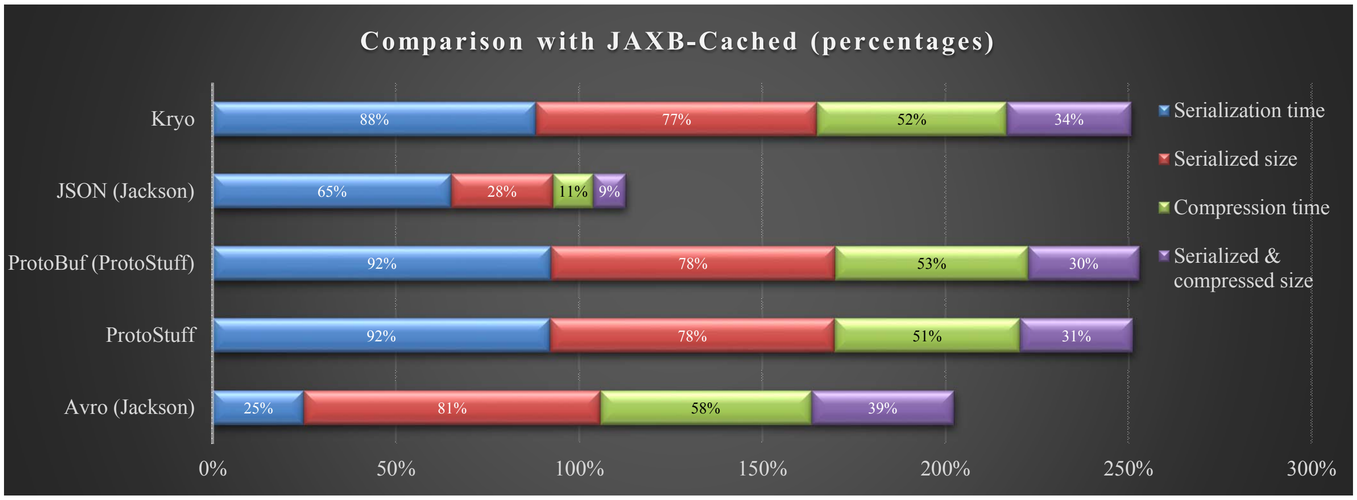

Fig. 5. JAXB comparison of size and speed improvements

A serialization format is language neutral if it is not tied to a particular programming language and supports cross platform applications if implementations exist in multiple languages.

The choice to use a language neutral or cross platform serialization format depends on whether other programming languages has to be supported for the distributed control application, and if so, it is important to check whether a format is language neutral and/or supports cross platform applications.

Some serialization libraries requires or allows the use of annotations, which might add additional work, in implementing the data model used, which in the case of IEC 61850 includes hundreds of classes, but it might allow certain implementations of data model classes that might otherwise not be possible.

In the case of IEC 61850, versioning can be handled by the application using the serialization as the version is specified by the logical nodes, but in other cases versioning could be an important characteristic of a serialization format and library, to allow the data model classes to change over time, while allowing an application to use multiple versions.

V. Conclusion

There are better alternatives to using XML, as JSON is also human readable and more compact, and binary formats, especially ProtoStuff, ProtoBuf, Kryo and FST, are faster and much more compact.

One thing that is special about XML and format extending XML, is the ability to specify new message parts, as part of the message.

But because this requires the system to know them in advance, which could have been done through documentation, or work with previously unknown message parts at runtime, this is only useful for rare complex cases.

When choosing a serialization format and library, it should be considered how active the development is, how big the community using it is, and how many resources are available, and seeing as this changes over time, is hard to quantify, and very subjective, this is outside the scope of the paper.

Further general information, not specific to power system, on pros and cons specific to a particular serializer can be found in online benchmarks.

Future work includes a comparison of compression formats and libraries, which could make the use of compression more useful, and a comparison of communication middleware, which together with this paper, could give a better overview over the possible Internet of Things Smart Grid power system services and applications, depending on the timeframe.

References

[1] "Scientific American," [Online]. Available: http://blogs.scientificamerican.com/plugged-in/renewable-energy-intermittency-explained-challenges-solutions-and-opportunities/. [Accessed 27 09 2016].

[2] "Beagle Bone Black," [Online]. Available: http://beagleboard.org/black. [Accessed 27 09 2016].

[3] "Odroid Specs," [Online]. Available: https://www.engadget.com/products/hard-kernel/odroid-u3/specs/. [Accessed 27 09 2016].

[4] R. E. Mackiewicz, "Overview of IEC 61850 and Benefits," IEEE PES Power Systems Conference and Exposition, Atlanta, GA, pp. 623-630, 2006.

[5] C. McParland, "OpenADR open source toolkit: Developing open source software for the Smart Grid," IEEE Power and Energy Society General Meeting, San Diego, CA, pp. 1-7, 2011.

[6] S. R. M. S. a. J. M. G. V. M. Uslar, "What is the CIM lacking?," IEEE PES Innovative Smart Grid Technologies Conference Europe (ISGT Europe), Gothenburg, pp. 1-8, 2010.

[7] "Java Serialization API," [Online]. Available: http://www.oracle.com/technetwork/articles/java/javaserial-1536170.html. [Accessed 27 09 2016].

[8] "FST," [Online]. Available: https://github.com/RuedigerMoeller/fast-serialization. [Accessed 27 09 2016].

[9] "Kryo," [Online]. Available: https://github.com/EsotericSoftware/kryo. [Accessed 27 09 2016].

[10] "JAXB," [Online]. Available: https://jaxb.java.net/. [Accessed 27 09 2016].

[11] "Jackson (XML)," [Online]. Available: https://github.com/FasterXML/jackson-dataformat-xml. [Accessed 27 09 2016].

[12] "XStream," [Online]. Available: http://x-stream.github.io/. [Accessed 27 09 2016].

[13] "ProtoStuff," [Online]. Available: http://www.protostuff.io/. [Accessed 27 09 2016].

[14] "Jackson," [Online]. Available: https://github.com/FasterXML/jackson. [Accessed 27 09 2016].

[15] "XStream (JSON)," [Online]. Available: http://x-stream.github.io/json-tutorial.html. [Accessed 27 09 2016].

[16] "Gson," [Online]. Available: https://github.com/google/gson. [Accessed 27 09 2016].

[17] "Genson," [Online]. Available: https://github.com/owlike/genson. [Accessed 27 09 2016].

[18] "SnakeYaml," [Online]. Available: https://bitbucket.org/asomov/snakeyaml. [Accessed 27 09 2016].

[19] "Jackson (YAML)," [Online]. Available: https://github.com/FasterXML/jackson-dataformat-yaml. [Accessed 27 09 2016].

[20] "MsgPack (v.6)," [Online]. Available: https://github.com/msgpack/msgpack-java/tree/v06. [Accessed 27 09 2016].

[21] "Jackson (MsgPack)," [Online]. Available: https://github.com/msgpack/msgpack-java. [Accessed 27 09 2016].

[22] "Jackson (Smile)," [Online]. Available: https://github.com/FasterXML/jackson-dataformats-binary/tree/master/smile. [Accessed 27 09 2016].

[23] "Jackson (BSON)," [Online]. Available: https://github.com/michel-kraemer/bson4jackson. [Accessed 27 09 2016].

[24] "Hessian Serialization," [Online]. Available: http://wiki4.caucho.com/Hessian_Serialize_Example. [Accessed 27 09 2016].

[25] "Jackson (CBOR)," [Online]. Available: https://github.com/FasterXML/jackson-dataformats-binary/tree/master/cbor. [Accessed 27 09 2016].

[26] "ProtoStuff Runtime," [Online]. Available: http://www.protostuff.io/documentation/runtime-schema/. [Accessed 27 09 2016].

[27] "Jackson (Avro)," [Online]. Available: https://github.com/FasterXML/jackson-dataformats-binary/tree/master/avro. [Accessed 27 09 2016].

[28] S. G. C. Group, "Smart Grid Reference Architecture," Smart Grid Coordination Group, 2012.