Analyzing .NET serialization components

Akos Nagy, Bence Kovari
Department of Automation and Applied Informatics
Budapest University of Technology and Economics
Budapest, Hungary
akos@aut.bme.hu, kovari@aut.bme.hu

***Abstract*—In this paper we present a number of available serialization libraries for the .NET platform, both integrated and 3rd party solutions and we analyze and compare their performance. Through this analysis and comparison we aim to deduct how the process of serialization can be further optimized and which parts of the process act as a bottleneck. The comparison also gives insight into how the data exchange format itself affects the whole process.**

Keywords—serialization; deserialization; XML; JSON; binary; .NET; performance measurement

I. INTRODUCTION

By the start of the 21st century a number of changes had happened in the IT industry which revolutionized both the way we use and design software solutions. These changes include the cost decrease of bandwidth, the general availability of network connections, the spread of smart mobile devices and the emergence of the cloud. The concept of distributed systems, while definitely not new, received renewed attention with this period.

Distributes systems are generally defined as a number of separate computational nodes that communicate with one another in order to achieve a common goal. During this communication messages are exchanged between the nodes. In order to produce the messages, the memory contents of the sender must be converted into a format that can be sent over the network as an array of bytes. This process is known as serialization. The receiver must be able to reconstruct the original memory content from this series of bytes. This is the process of deserialization.

While ultimately the message is only a series of bytes, there are a number of different formats designed for a number of different purposes. The first generation of distributed systems mostly consisted of remote procedure calls (RPC). This technology is strongly coupled to the programming platform used to realize the architecture. As platform-independency is not a requirement, these architectures popularized the spread of binary data exchange formats. These represent the memory content as a series of bytes using every available platform dependent information to make the communication process more robust or faster.

In the years following the emergence of RPC a new requirement came along that needed to be satisfied: platform independency. RPC, being a platform-specific method of distributed communication was no longer an option. This brought along the emergence of the service oriented architecture and its web based realization, the web services. Web services use the HTTP protocol stack for communication with XML messages. XML has the benefit of being platform independent, but it takes more resources to both produce and process the messages.

Another step in the evolution of distributed systems was the REST architecture. Although it is not a new concept, the spread of mobile devices acted as a catalyst for this architectural style to become an industry-wide first choice when dealing with web based communication. While not confined to only one data exchange format, the traditionally used JSON format is more compact then XML but retains most of its capabilities, including platform independency. As mobile devices have an important weakness of low battery life, making messages as small as possible is a crucial criterion in these systems.

During this evolution, the importance of binary serialization has decreased, but it is not without merit. Binary formats still have a number of domain specific applications. The Avro [1] binary format is used to interact with Hadoop solutions. The Thrift [2] protocol stack is used by Facebook for communication. The Protocol Buffers (also known as ProtoBuff) [3] designed by Google has the explicit goal of being smaller and faster than XML. While these are binary formats, they are also very well specified and are platform independent.

In this paper we analyze the serializer libraries mentioned before: the Microsoft Avro library [4], the .Net implementations of Thrift and Protocol Buffers. Along these 3rd party solutions integrated components are also included in the analysis. The serializer of choice for .NET RPC is the BinaryFormatter. For the web services scenario, the default solution is the DataContractSerializer and in the case of REST services, the de facto choice is the JSON.NET component [5]. These are all compared to each other and the 3rd party solutions. The aim of this analysis is to shed light on how the data exchange format itself affects the serialization performance and to propose optimization possibilities.

978-1-5090-2380-6/16/$31.00 ©2016 IEEE

II. Related work

A. Related research

A number of authors have published papers with the same goal: to analyze the performance of different serializer engines and evaluate their performance.

In [6] the author analyzes 12 different libraries from both a quantitative and a qualitative approach He arrives at the same conclusion as we do: text-based formats are not as impressive from a quantitative point of view as binary formats when it comes to compactness. From a qualitative point of view however, the author prefers the readability of JSON and XML formats. In [7] the author continues his work to evaluate the serializers regarding other nonfunctional requirements.

The authors of [8] carry out a series of measurements about XML serialization in the Java platform and use linear regression to define a formula that can be used to predict serialization costs. As a further step we aim to use our data the same way for the .NET platform.

As discussed in the previous section, the spread of mobile devices is a strong propelling force towards implementing faster and more compact serialization mechanism. Authors of [9] carry out measurements and evaluate the results from the very specific point of view of mobile devices. They arrive at the conclusion that JSON is a superior alternative to XML for communication purposes. For storage purposes they also recommend binary formats, as their speed and storage requirements are better compared to the text-based formats. In case of JSON vs. XML we have the same conclusion, but the speed of the binary solution is not as impressive on the .NET platform,

Authors in [10] compare the XML and binary serialization techniques of the Java and .NET platforms. They arrive at the conclusion that Java has a superior binary serializer mechanism. Though the reasons they give are very platform specific, this still points to the fact that the binary serializer of the .NET platform can still be improved. Our results also show that the all-inclusive .NET binary serializer underperforms when compared to most of the other components.

The authors of [11] study binary serialization of C++, .NET and Java. They conclude that all platforms have the necessary capabilities to perform efficient binary serialization.

The main contribution of this paper is twofold. First, the measurement system is more focused regarding the programming platform, as we only discuss .NET components. On the other hand, the measurement system is broader when it comes to the analyzed components. We examined more components, in the case of some formats even more than one. The measurement framework implemented to carry out the experiments is a flexible system allowing for easy integration of even more components in the future.

Our goal with this paper is not only to add performance measurements of .NET components to the existing literature and elaborate on them. We also think that the measurements provided in this paper can serve as a basis for an all-inclusive knowledgebase that developers can turn to when they have to decide between communication formats and components.

B. Serializer components

In the paper .NET serializer engines are discussed, some of them integrated and some of them are 3rd party libraries.

The BinaryFormatter serializer is the all-inclusive binary serializer solution of the .NET framework. As its primary function is to provide serialization for RPC scenarios, the output format is totally incompatible with other platforms. It also includes .NET specific type information in the output stream, thus enforcing the strongly typed nature of the .NET platform even over the wire. This feature is useful in .NET-.NET communication scenarios, but makes the whole process slow and brittle. Including this information takes time and takes up some space and also introduces compatibility problems, such as version handling.

The DataContractSerializer is part of the Windows Communication Foundation (WCF) extension library. The aim of this library is to introduce a unified framework for all communication related tasks in the .NET framework. Although not confined for web services only, its design is strongly affected by the concept. The DataContractSerializer is the default component for web services built in WCF. As such, it produces an output that is XML based, platform-independent and can be embedded in a SOAP envelope (the communication protocol for web services). The platform-independent nature of the serialization makes it a little more robust, but the XML format is still verbose. This serializer is designed to be extensible, but extension have an additional performance cost.

The JSON.NET serializer is a 3rd party component for JSON serialization, but over the years it has become the de facto serializer component for the Web API library, which is used to build REST services in the .NET framework. Credited as being a high performance serializer, it is often used as an etalon for JSON measurements. It is also highly extensible and completely compatible with the latest JSON RFC [12].

Apache Avro, designed by the Apache Software Foundation is a binary serialization system. The binary protocol is strongly specified, in theory it can be implemented on any platform. Currently a number of popular platform have Avro libraries implemented (e.g. Java, .NET, Python). Avro uses dynamic typing, so no prior code generation is needed to handle objects and Avro streams. In case when persistence is also a requirement, the schema of the objects can be stored along the contents. Contrary to a lot of other popular solutions (e.g. Protocol Buffers) it does not rely on tags or identifiers. Schema changes are also handled and resolved automatically. Avro is mainly used in big data scenarios with Hadoop solutions.

Protocol Buffers are a structured binary format developed by Google. It relies on user assigned tags to serialize the data. The specification calls for a predefined schema file to be present (called .proto files) that describe how data is to be serialized, but fully automatic versions are also implemented. It is claimed by Google that it is 3-10 times smaller and 20 to 100 times faster than XML [3].

Thrift is also developed by the Apache Software Foundation. It is not only a serializer library like Avro or ProtoBuff, but also a framework for distributed software engineering scenarios. Like ProtoBuff, it also relies on

predefined schemas to serialize data. The default binary serialization format can be further processed with built-in codecs but also completely changed to JSON.

III. Contributions

A. Measurement framework

In order to measure the performance of serializers a measurement framework has been implemented. The framework defines measurable interfaces which must be implemented by measurable components. These measurements provide the time of the serialization operation and the length of the serialized data. Also, to check if the operation produced the correct results, the serialized bytes and the deserialized objects are both available. Every serializer library must be wrapped into a class that implements this interface.

The input of the system consists of the number of top level objects in our object graph that has to be serialized, the type of the serializer, the number of times the operation has to be run and the operation itself (serialization or deserialization). From this we generate an object graph that is to be serialized. This graph is generated so the object content values (e.g. string and list lengths) have the same statistical properties as the original data source we used to build the object model (categories and products of the Northwind sample database [13]). The output is the time of the operation and the length of the output stream (in case of serialization only). These values are calculated as the average of all runs except for the best and the worst values.

In order to compensate for any bias that can arise from platform specific issues, the core component of the system runs in a separate application domain and can only complete one measuring session (i.e. serialization of 20 objects to JSON, or deserialization of 50 Avro objects). To further automate the system and run measurements for all serializers and different object sizes, the core system can be run from scripted environments.

The following section shows our results for different numbers of objects and serializers. The section also contains the first, the best and the worst run for every combination of serializer and object number.

B. Measurement results

This section discusses the measurement results. We measured the performance of every serializer solution running both the serialization and the deserialization 20 times and averaging the time and output lengths, with omitting the worst and the best results. The best and worst results are also shown at the end of this section, as well as the result from the first run. The results are discussed in two parts: for small number of objects (max 500) and for large number of objects (more than 1000).

Fig. 1 shows the average speed for each serializer in the smaller measurement domain:

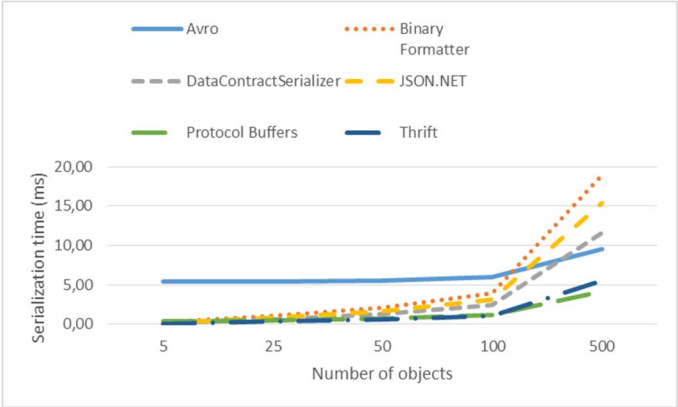

Fig. 1 - Average serialization speed in the smaller measurement domain

From Fig. 1 it can be seen that the examined serializers can be grouped into three categories regarding average speed. The first category contains the BinaryFormatter, the JSON.NET serializer and the DataContractSerializer. This concludes that the text-based serializers are slower than the binary serializers in general. The reason for BinaryFormatter being in this group is that it includes .NET specific metadata into the output stream. This extra data has to be processed and written to the output stream which results in the extra time.

The second group contains the Thrift and the Protocol Buffer serializers. Being binary formats they are expected to be the fastest.

The third group contains only the Avro serializer. The characteristics of the diagram indicate that this module scales the best from all the analyzed solutions.

Fig. 2 shows the average output length for the same measurement domain:

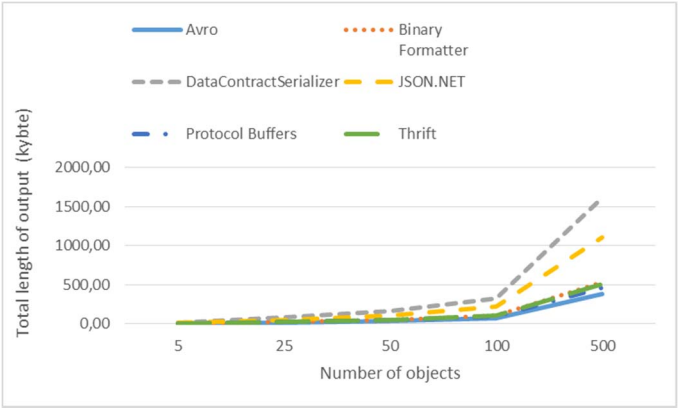

Fig. 2 - Average output length in the smaller measurement domain

From Fig. 2 it can be seen that regarding average output length, the serializers can be grouped into two categories. Group one contains the text-based serializers, group two the binary serializers. As expected, the text-based formatters produce longer output.

Fig. 3 shows the results for average speed in a larger measurement domain.

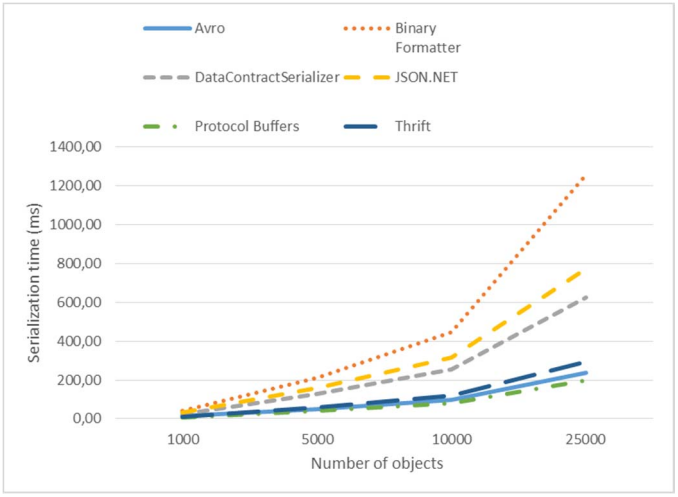

Fig. 3 - Average serialization speed in the larger measurement domain

The average speed characteristics are similar to the ones in the smaller domain, with the scalability of the Avro serializer becoming a real advantage making it one of the best solutions. The scalability of the BinaryFormatter also becomes apparent, though it is not as strong as it is for the Avro.

Fig. 4 shows the output length for the same larger domain:

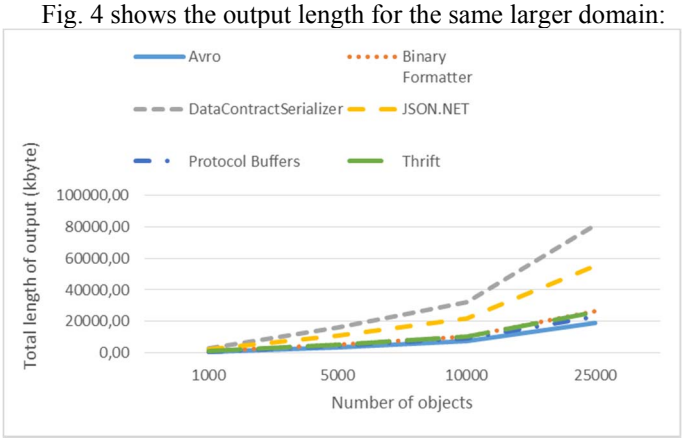

Fig. 4 - Average output length in the larger measurement domain

The characteristics are consistent with the measurements from the smaller domain.

Fig. 5 and Fig 6. show the average deserialization times in the same arrangement. For the smaller domain similar conclusions can be drawn for the average speed as in the case of the serialization. In the larger domain, the scalability of the Avro serializer again becomes apparent, but the scalability of the BinaryFormatter is worse than it is in the case of serialization.

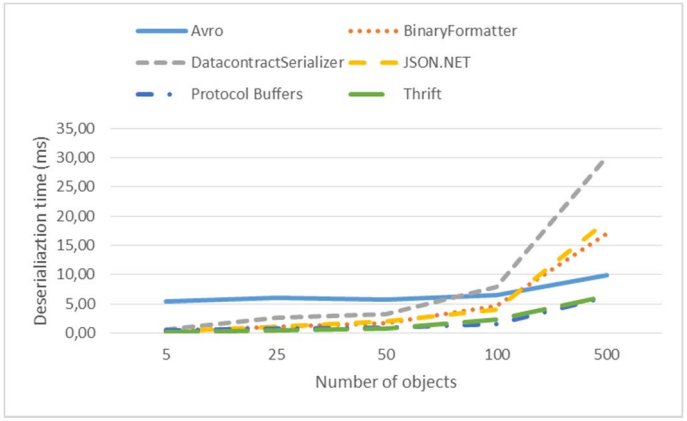

Fig. 5 - Average deserialization speed in the larger measurement domain

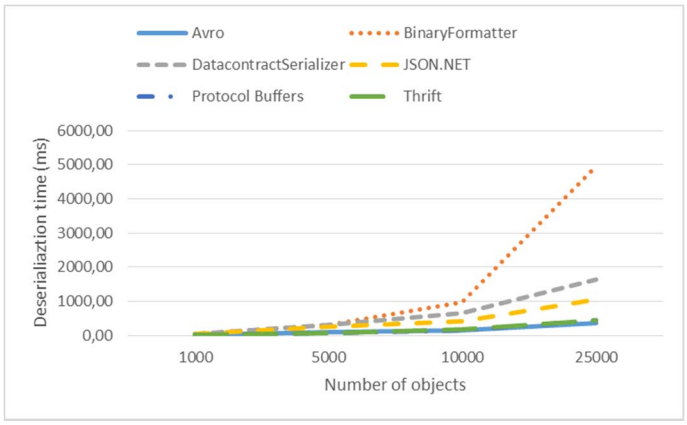

Fig. 6 - Average deserialization speed in the larger measurement domain

Table 1 contains the aggregated views of the serialization measurements, extended with the first, best and worst speeds and also the average data length for smaller measurement domain.

TABLE 1

	No. of objects	First (ms)	Best (ms)	Worst (ms)	Average (ms)	Average (kbyte)
Avro	5	96.65	4.95	96.65	5.40	3.89
	25	55.95	5.13	55.95	5.36	19.36
	50	55.81	5.36	55.81	5.47	38.96
	100	57.05	5.85	57.05	5.96	76.86
	500	60.45	9.37	60.45	9.59	386.72
Binary Formatter	5	2.20	0.18	2.20	0.25	6.39
	25	2.35	0.81	2.35	1.04	27.23
	50	4.63	1.74	4.63	2.05	53.53
	100	5.30	3.67	5.30	3.87	107.58
	500	20.23	18.33	20.23	18.97	529.33
DataContract Serializer	5	18.54	0.11	18.54	0.18	16.13
	25	9.62	0.53	9.62	0.60	80.43
	50	10.74	1.15	10.74	1.23	164.78
	100	11.92	2.14	11.92	2.37	323.84
	500	21.59	10.94	21.59	11.64	1612.25
JSON.NET	5	101.89	0.13	101.89	0.17	11.08
	25	123.11	0.69	123.11	0.79	54.26
	50	127.87	1.45	127.87	1.59	110.25
	100	99.57	2.90	99.57	3.10	222.88
	500	107.67	14.92	107.67	15.36	1101.96
Protocol Buffers	5	58.55	0.36	58.55	0.40	4.64
	25	77.40	0.49	77.40	0.53	23.19
	50	91.05	0.68	91.05	0.73	45.98
	100	57.83	1.05	57.83	1.12	91.10
	500	60.71	3.93	60.71	4.18	460.72
Thrift	5	2.71	0.04	2.71	0.06	5.06
	25	5.84	0.25	5.84	0.34	26.08
	50	6.45	0.50	6.45	0.54	51.40
	100	4.09	1.00	4.09	1.10	103.83
	500	8.64	5.17	8.64	5.48	513.24

From the measurements it can be seen that without exception the worst run of is always the first. This can be explained with the fact that each serializer component has to be built-up. When the component first encounters an object, the type of the object has to be discovered for type information. From that point on, the same type information can be cached and reused later.

The ratio of the results of the first runs and the average of the runs is also an indicator of scalability. As a general rule it can be seen that as the number of objects grow, the average speed becomes close to the speed of the first run. But in the case of the BinaryFormatter and the DataContractSerializer, the ratio is close to 1. For JSON.NET, ProtoBuff and Thrift it is around 0.8 and for the serializer that scales the best, Avro, it is the least, 0.74. This also indicates that Avro is the one that scales the best.

IV. Conclustion and Future work

The paper presented performance analysis for a number of .NET serializer engines, both integrated and 3rd party. In order to measure the performance we designed a measurement framework that is used to automate the process. From the results we can see that while binary formats tend to have a better performance both in time and in compactness. We can also see that for most cases, the first run of the module produces the worst time performance result. Also as a general rule, there is a strong connection for every module for serialization and deserialization performance.

In order to further optimize the serialization process, we can deduct that in order to realize an efficient module, both the serialization and the deserialization has to be effectively implemented. Binary formats have a better performance both in speed and in compactness, so an efficient serializer should probably use a binary format. While platform independency is a strong drive towards text-based formats, binary formatters can also be used in these scenarios and it does not necessarily diminish their performance, as shown for the Avro, ProtoBuff and Thrift formats.

It is also notable that for most serializers, the first run is the worst, so optimizing the first run can result in notable performance improvement altogether.

For future work, we plan to include even more serializers into the analysis and also examine the current ones with different customizations and parameterizations. Also, measuring how much time is spent writing to the output/ reading from the input could be measured. With these data we plan to propose an efficient serializer algorithm.

Acknowledgment

This work was partially supported by the TÁMOP-4.2.1.D-15/1/KONV-2015-0008 project.

References

[1] „Apache Avro Specification," [Online]. Available: https://avro.apache.org/docs/1.7.7/spec.html. [Accessed: 16 january 2016].

[2] M. Slee, A. Agarwal and M. Kwiatkowski, „Thrift: Scalable cross-language services implementation," *Facebook White Paper,* volume 5, issue 8, 2007.

[3] „Protobol Buffers - Developers Guide," Google, [Online]. Available: https://developers.google.com/protocol-buffers/. [Accessed: 16 january 2016].

[4] „Microsoft Avro Library," Microsoft, [Online]. Available: https://hadoopsdk.codeplex.com/wikipage?title=Avro%20Library. [Accessed: 16 january 2016].

[5] „JSON.NET Home site," Newtonsoft, [Online]. Available: http://www.newtonsoft.com/json. [Accessed: 16 january 2016].

[6] K. Maeda, „Performance evaluation of object serialization libraries in XML, JSON and binary formats," in *Digital Information and Communication Technology and it's Applications (DICTAP), 2012 Second International Conference on*, 2012.

[7] K. Maeda, „Comparative Survey of Object Serialization Techniques and the Programming Supports," *Journal of Communication and Computer,* volume 9, issue 8, pp. 920-928, 2012.

[8] G. Imre, H. Charaf and L. Lengyel, „Estimating the Cost of XML Serialization of Java Objects," in *Engineering of Computer Based Systems (ECBS-EERC), 2013 3rd Eastern European Regional Conference on the*, 2013.

[9] A. Sumaray and S. K. 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*, 2012.

[10] M. Hericko, M. B. Juric, I. Rozman, S. Beloglavec and A. Zivkovic, „Object serialization analysis and comparison in java and. net," *ACM Sigplan Notices,* volume 38, issue 8, pp. 44-54, 2003.

[11] C. J. Tauro, N. Ganesan, S. Mishra and A. Bhagwat, „Object Serialization: A Study of Techniques of Implementing Binary Serialization in C++, Java and .NET," *International Journal of Computer Applications,* volume 6, issue 45 2012.

[12] T. Bray, „The javascript object notation (json) data interchange format," 2014.

[13] „Northwind database," Microsoft, [Online]. Available: https://northwinddatabase.codeplex.com/. [Accessed: 16 january 2016].