EPJ Web of Conferences **245**, 05017 (2020) https://doi.org/10.1051/epjconf/202024505017
CHEP 2019

dCache – Efficient Message Encoding For Inter-Service Communication in dCache

Evaluation of Existing Serialization Protocols as a Replacement for Java Object Serialization

Lea Morschel[1,*], *Olufemi* Adeyemi[1], *Vincent* Garonne[2], *Dmitry* Litvintsev[3], *Paul* Millar[1], *Tigran* Mkrtchyan[1], *Albert* Rossi[3], *Marina* Sahakyan[1], *Juergen* Starek[1], and *Sibel* Yasar[1]

[1]Deutsches Elektronen-Synchrotron DESY, Hamburg, Germany
[2]Nordic e-Infrastructure Collaboration (NeIC), University of Oslo, Norway
[3]Fermi National Accelerator Laboratory, Batavia, USA

Abstract. As a well established, large-scale distributed storage system, dCache is used to manage and serve huge amounts of data collected by high energy physics, astrophysics and photon science experiments. Based on a microservices-like architecture, dCache is built as a modular distributed system, where each component provides a different core functionality. These services communicate by passing serialized messages to each other, a core behavior whose performance properties can consequently affect the entire system.
This paper compares and evaluates different data serialization protocols in computer science with the objective of replacing and improving upon *Java Object Serialization* (JOS), which has increasingly presented itself as no longer being sufficiently performant for encoding messages. The criteria for choosing a new framework are collected, analyzed and formalized. The primary motivation for replacing Java serialization for encoding dCache messages is increasing the general speed of message-passing and thereby reducing the round-trip time for user requests. Emphasis is also placed on schema evolution capabilities and framework usability. Approaches for generalizing (de)serialization speed and size measurements based on data structure complexity are introduced, criteria for measuring documentation, learning curve, maintainability and introduction effort are defined. Finally, several selected serialization protocols are evaluated and compared accordingly, concluding with a recommendation for a suitable JOS replacement.

1 Introduction

The *dCache* software [1] is an open-source distributed storage system written in *Java*, which uses a microservices-like architecture to provide location-independent access to data. It is designed to support a wide range of use cases, from high-throughput data ingest, being dynamically scalable to hundreds of petabytes, as well as deployable in heterogeneous systems and on commodity hardware. It is easy to integrate with other systems, because it can communicate over several protocols for accessing data and enabling authentication, and supports

*e-mail: lea.morschel@desy.de

© The Authors, published by EDP Sciences. This is an open access article distributed under the terms of the Creative Commons Attribution License 4.0 (http://creativecommons.org/licenses/by/4.0/).

different qualities of data storage, including tertiary storage support, for which it is able to use disks as a caching layer [2]. Within the system, a significant portion of the time needed for internal communication between services is spent on serializing and deserializing messages. The primary motivation for replacing Java serialization is increasing the general speed of message-passing and thereby reducing the round-trip time for user requests. Serializers are compared in a more general sense in order to understand their strengths and weaknesses. This paper is the summary of a larger scientific thesis [3].

2 Related Work

As one of the first, Evans et.al. describe why *object serialization in Java* (JOS) is inappropriate for providing persistence [4]. In a 1999 paper, Philipsen, Haumacher and Nester evaluate JOS as being too slow and introduce a design for a new Remote Method Invocation as a drop-in replacement, especially for use in high performance computing [5].

Today, many different stand-alone libraries for serialization exist. In [6], twelve different object serialization libraries are compared for the Java language with regard to many metrics of interest for the scope of this paper. However, the authors do not implement necessary best practices for Java benchmarking as described by Goetz [7] and consequently present unreliable results, which are very likely influenced by *Java Virtual Machine* (JVM) optimizations. Bittl et al. analyze different data serialization schemes to evaluate performance with regard to their usage in wireless Car-to-X communication [8]. Although the results are partially of interest, they primarily focus on aspects related to their specific context. Most importantly, these results were not obtained in a Java based environment and its focus on areas of memory consumption and encoding size is only of secondary interest to the scope of this paper.

Therefore, a thorough evaluation of Java-specific, speed optimized serialization protocols is necessary with regard to the specific requirements of the dCache system.

3 Current Message Serialization in dCache

Within the dCache architecture, a microservice, called cell, represents the most fundamental executable building block. Each cell fulfills a specific task and can be grouped into a certain type category, for example *pool* as a storage element or *door* for enabling access to data over a specific protocol, like HTTP. The cells communicate with each other via messages. Like in IP packages, messages in dCache consist of an envelope carrying meta information and a message payload. An envelope may be (de)serialized independently from the payload for routing purposes, which significantly reduces the processing time. There are a total of 159 different non-abstract message classes at the time of writing.

Within dCache, the Java object serialization is used to serialize these messages to a binary format. It has been added to the Java language in 1996 [9] and has the advantage of being trivial to adopt in a Java-based software, because annotated objects can be serialized automatically and invisibly to the programmer. It has undoubtedly played a large role in the success of the Java programming language, but has increasingly and repeatedly proven to be problematic in many ways due to inherent design flaws [7]. Besides making life difficult for developers by having to consider hidden operations, that might require an object's structure to remain backwards compatible or may leave it open to injection of malicious code by violating encapsulation, Java serialization is also not as performant as modern frameworks and serializes exclusively to a Java-specific format that cannot be easily reconstructed outside of the JVM.

Mean	Median	Rating Count	Property Key	Property
9.00	8, 10	6	*A*	Run in parallel with JOS
8.00	8	7	*B*	Speed improvements compared to JOS
7.57	7	7	*C*	Support for schema evolution
6.71	6	7	*D*	Introduction effort and maintainability
6.29	6	7	*E*	Documentation and gentle learning curve
5.86	5	7	*F*	Framework independence of a schema/an encoding format
5.43	5	7	*G*	Platform and language independence
3.86	4	7	*H*	Smaller serialized format than with JOS

Table 1: Results of the *dCache* developers' ratings of serialization framework properties

4 Criteria for a New Serialization Protocol in dCache

In addition to the traditional batch analysis, there is a trend towards increasingly interactive usage of dCache via protocols such as NFS or WebDAV, where small latencies are much more significant. The motivation for wanting to replace JOS with a different framework in dCache primarily results from a request's round-trip time no longer being sufficiently fast for the individual user in this scenario.

In order to assess additional requirements of a new serialization protocol, an analysis was conducted among the currently active dCache developers regarding the importance of different criteria concerning system functionality and development ease. Each of the eight defined properties was rated on a scale between 1 and 10, with the size of the number being positively correlated with the relevance of the rated criterion. The criteria and their resultant ratings are shown in Table 1 ordered by assessed importance.

Criterion *A* concerns the ability of the protocol to be able to work in parallel with JOS, which is important for backwards compatibility. It is not a technology selection criterion but purely depends on the implementation. The most important property thereafter are performance improvements (Criterion *B*), as is frequently the case with serialization, closely followed by schema evolution (Criterion *C*). Soft criteria concerning maintainability (Criterion *D*) and learning curve (Criterion *E*) are moderately important. The size of an encoding (Criterion *H*) is almost irrelevant, however. Because this is usually of significant importance in most of the published comparisons of serialization protocols, this creates a more unusual profile of requirements.

5 Evaluation

In the following sections the evaluated protocols are presented, followed by an introduction of the evaluation scenarios and test approaches as well as the employed tools and environment of execution.

5.1 Serialization Protocols to be Evaluated

The compared serialization protocols include the native Java serialization to establish the baseline. Additional frameworks were chosen due to their ease of integration as full object

Framework	Plot Legend	Type	Serialization Format	Platform Dependence
Apache Avro (Avro)[10]	AV, AVJ	SBS	binary, JSON	agnostic
Fast-Serialization (FST)[11]	FST	FOGS	binary	agnostic
Java Object Serialization (JOS)[12]	JOS	FOGS	binary	JVM bound
Kryo[13]	KRY	FOGS	binary	JVM bound
Protocol Buffers (Protobuf)[14]	PB	SBS	binary	agnostic
Protostuff Runtime (PSR)[15]	PSR	FOGS	binary	agnostic

Table 2: Important features of the evaluated serialization protocols

graph serializers or because they are promising representatives of modern schema-based serialization approaches, have been shown to be high-performing, well supported and feature rich. The frameworks are additionally required to be open source in order to be usable in the dCache project. The selected frameworks are shown in Table 2 alongside their most relevant features with respect to this analysis. It includes whether they are a *schema-based serializers* (SBS) that require the manual definition of data structures or *full object graph serializers* (FOGS) that can traverse the data structure and dynamically infer a schema at runtime. The serialization formats are almost exclusively binary, which is usually faster.

5.2 Evaluation Scenarios

These frameworks are evaluated according to the main criteria of interest as described in Section 4. They need to be formalized in order to be able to compare different frameworks.

Performance

It is only possible to compare the performance of different serialization protocols per data structure. Because different features of a data structure may be (de)serialized with divergent approaches and overheads, there is no definitive order on data structures that is based on their complexity with regard to the (de-/)serializing speed or encoding size independent of a specific serialization protocol, which may be used. A common and practical strategy to address this problem is to evaluate protocols tightly coupled to the requirements of the respective context by selecting and focusing on common representatives of data structures therein. In a more general approach, it may be analyzed how different data types scale with growing content, especially composite and container types like *lists* or *maps*, and testing deeply nested structures.

In order to obtain results that do not merely consider the current dCache structures in use, three sets of data structures are defined, which each aim to generalize certain aspects or represent the specific use case and lend themselves to compare protocols in different regards:

- *TypeList*: Lists (Int, Double, String) of different sizes (10, 50, 100, 1000, 1000, 100000)
- *Composites*: Six small, composed classes that vary object- and primitive-typed members
- *dCache-like*: Representative message classes from dcache

Support for Schema Evolution

The support for schema evolution is rated based on the support for forward and backward compatibility of message format changes. Additionally, the amount of effort and difficulty of usage is taken into account.

Qualitative Framework Features

With regard to several aspects, a comparison of frameworks was conducted by describing different qualitative indicators that are defined by the presence of certain features or the fulfillment of requirements and using them as a basis for assessment. For this analysis, the frequently used *Likert Scale* [16] was used for defining and rating criteria. The selected approach defines ten ratable items for the area of documentation and learning curve and ten items for rating the introduction effort and maintainability.

5.3 Environment and Tools

In order to test the performance of different serialization protocols independently of the complexities of the dCache system, a simple, Java-based testing software was created [17]. The *Java Microbenchmark Harness* (JMH) was used for benchmarking the serialization and deserialization code of each of the protocols for every test object. Attention was payed to avoid the common pitfalls of Java benchmarking that stem from *Just In Time Compilation* features and smart optimizations by the Java Virtual Machine such as loop unrolling, dead code elimination and constant folding. Redundant testing in different environments showed that the relative performance within environments was preserved, so that results on a single machine can be representative for discovering performance differences.

6 Results

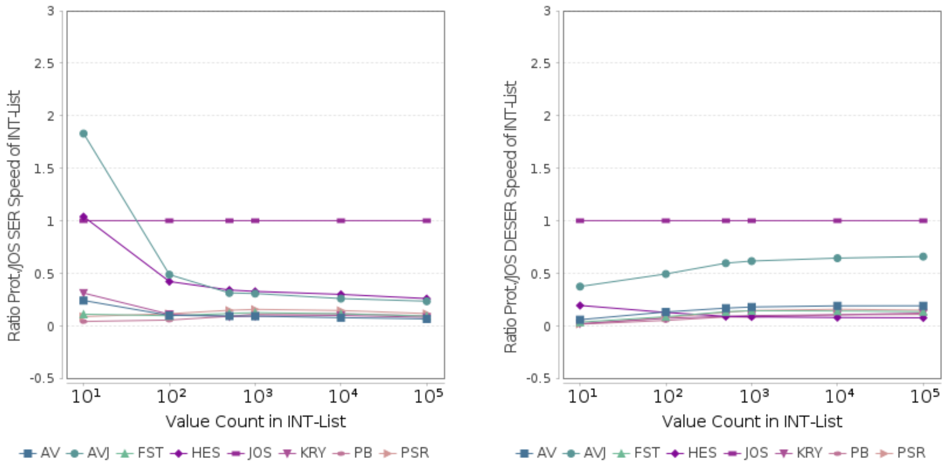
In the following, the results of evaluating the selected protocols are analyzed with regard to each of the defined categories of inquiry. The performance analysis took about 140 days of converted computation time but was effectively highly parallelized.

Performance

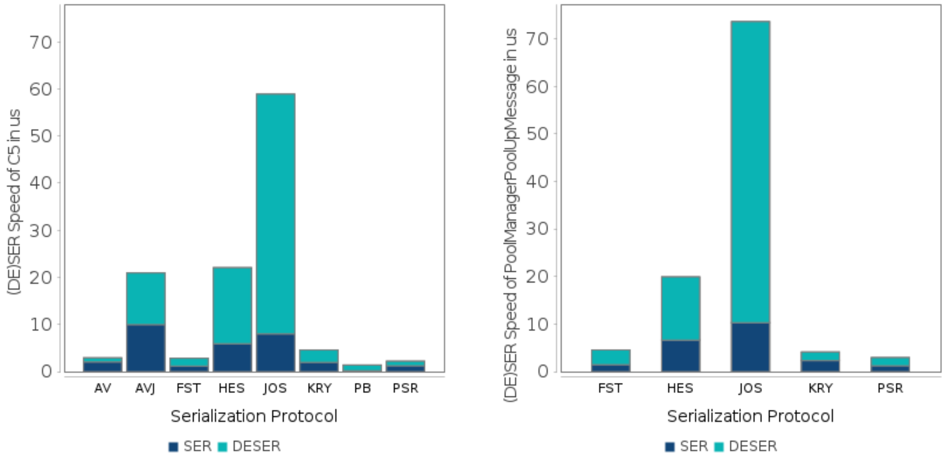
It was discovered that the lightweight message envelope is already being (de)serialized very efficiently, so that replacing it with different serializers did not result in any significant improvements. Encoding and decoding the occasionally very large payload messages, however, could be improved with almost every analyzed protocol. Figure 1a shows the serialization speed of IntList objects as the ratio of each protocol's speed over the JOS performance. The x-Axis shows the number of pre-generated values contained in the list object. It was found that JOS deserialization was slower than any other protocol as shown in Figure 1b, especially with increasing list sizes.
Similar trends can be observed for objects from the Composites category, of which the results for the most complicated ones are shown in Figure 2a. The performance of JOS deserialization is especially bad, while Protobuf is fastest in general. Because the complexity of existing dCache messages was observed to be too large to be able to consider adopting Protobuf in the near future, only the FOGSs were evaluated with regard to the dCache-like objects. Figure 2b shows this evaluation with similar results as in the composite case.

Support for Schema Evolution & Qualitative Framework Features

According to this evaluation, Protobuf supports schema evolution best with the least effort involved, closely followed by Protostuff-runtime, Avro and JOS on the second place. FST is insufficient and Hessian poor. Regarding the qualitative features, Kryo (95 %) has earned the most points, closely followed by Protostuff (92 %) on the second place. They are followed by Avro, Protobuf, FST and Hessian.



(a) Serialization Performances of IntList Objects (b) Deserialization Performances of IntList Objects



(a) The most Complicated Composite Object (b) The dCache `PoolManagerPoolUpMessage`

Figure 2: (De)Serialization Performances of Composite Messages

7 Summary and Outlook

While Protobuf was the fastest serializer with the best schema evolution support, the conducted analysis uncovered the problematic complexity and large number of messages within dCache, which currently prevents the adoption of a schema-based serializer. It is hoped that this complexity will gradually be reduced and may eventually allow a migration to Protobuf or similar. For immediate improvements, the FST serializer as one of the three fastest FOGS was easiest to integrate in the current situation and enabled the reduction of an isolated message round trip time by about 10 % in a test instance. The problem with schema evolution was solved by falling back to JOS in case of different dCache versions. FST will be selectable via configuration file to be used as the default serializer in dCache starting from version 6.1.

References

[1] www.dCache.org, *dCache Website*, https://www.dcache.org, accessed: 2019-07-10

[2] P.A. Millar, O. Adeyemi, G. Behrmann, P. Fuhrmann, V. Garonne, D. Litvinsev, T. Mkrtchyan, A. Rossi, M. Sahakyan, J. Starek, 2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP) pp. 651–657 (2018)

[3] L. Morschel, *Efficient Message Serialization for Inter-Service Communication in dCache*, https://bib-pubdb1.desy.de/record/436675 (2019)

[4] H. Evans, Tech. rep., Department of Computing Science, University of Glasgow (2000)

[5] M. Philippsen, B.H. andChristian Nester, CONCURRENCY: PRACTICE AND EXPERIENCE **12**, 495 (2000)

[6] K. Maeda, 2012 Second International Conference on Digital Information and Communication Technology and it's Applications (DICTAP) pp. 177–182 (2012)

[7] B. Goetz, *OpenJDK: Towards Better Serialization*, http://cr.openjdk.java.net/~briangoetz/amber/serialization.html (2019), accessed: 2019-07-14

[8] S. Bittl, A.A. Gonzalez, M. Spaehn, W. Heidrich, International Journal On Advances in Telecommunications **8**, 48 (2015)

[9] R. Riggs, J. Waldo, A. Wollrath, K. Bharat, Computing Systems **9**, 291 (1996)

[10] www.avro.apache.org, *Welcome to Apache Avro*, https://avro.apache.org/, accessed: 2019-07-28

[11] www.github.com/RuedigerMoeller/fast serialization, *FST: fast java serialization drop in-replacement*, https://github.com/RuedigerMoeller/fast-serialization, accessed: 2019-07-21

[12] Oracle, *Java Object Serialization*, https://docs.oracle.com/javase/8/docs/technotes/guides/serialization/index.html, accessed: 2019-07-28

[13] www.github.com/EsotericSoftware/kryo, *Java binary serialization and cloning: fast, efficient, automatic*, https://github.com/EsotericSoftware/kryo, accessed: 2019-07-21

[14] www.developers.google.com/protocol buffers, *Protocol Buffers | Goodle Developers*, /urlhttps://developers.google.com/protocol-buffers/, accessed: 2019-07-27

[15] www.github.com/protostuff/protostuff, *Protostuff – A java serialization library with built-in support for forward-backward compatibility (schema evolution) and validation*, https://github.com/protostuff/protostuff, accessed: 2019-07-21

[16] I.E. Allen, C.A. Seaman, Quality Progress **40**, 64 (2007)

[17] L. Morschel, *Codebase for Benchmarking Common Java Serializers*, https://github.com/lemora/serializer-benchmarking (2019)