Scientific Thesis in the Context of the Practical Semester

Efficient Message Serialization for Inter-Service Communication in dCache

**Evaluating a Replacement for Java Serialization in the Distributed
Storage System dCache**

Handed in on:

August 22, 2019

Author:

Lea Morschel

Deutsches Elektronen-Synchrotron DESY

Notkestraße 85

D-22607 Hamburg

Supervisor (University):

Prof. Dr. Dennis Säring

University of Applied Sciences Wedel

Feldstraße 143

D-22880 Wedel

Supervisor (Company):

Tigran Mkrtchyan

Deutsches Elektronen-Synchrotron DESY

Notkestraße 85

D-22607 Hamburg

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 thesis 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 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.

Contents

Abstract	III
List of Figures	VI
List of Tables	VII
List of Listings	VIII
1 Introduction	1
1.1 Introduction to the Problem Area	1
1.2 Related Work	2
1.3 Structure of the Thesis	3
2 Serialization Overview and Classification	5
2.1 Data Structures As Directed Graphs	6
2.2 Flexibility of Data Representation	9
2.3 Data Types, Type Safety and Security	10
2.4 Format of Encoded Data	12
3 dCache and its Use of Java Serialization	15
3.1 Current Serialization of Messages Within <i>dCache</i>	16
3.2 Problems and Advantages of Using Java Serialization	17
3.3 Criteria for a new Serialization Protocol within <i>dCache</i>	18
4 Comparing Different Serialization Protocols	21
4.1 Serialization Protocols to be Evaluated	21
4.1.1 Java Native Serialization	22
4.1.2 Kryo	22
4.1.3 Fast-serialization	23
4.1.4 Hessian	23
4.1.5 Protocol Buffers	24
4.1.6 Protostuff	25
4.1.7 Apache Avro	25
4.2 Criteria For Evaluation	26
4.2.1 Performance Relative to Data Structure Complexity	27
4.2.2 Qualitative Framework Features	29
4.3 Evaluating Performance and Encoding Size	30
4.3.1 Environment and Tools	31
4.3.2 Data Structures	33
4.3.3 Benchmarking Setup	40
4.3.4 Evaluating Individual Serialization Protocols	41
4.3.5 Comparison of Serialization Protocols	49
4.3.6 Evaluating the Influence of the Computing Environment	54
4.4 Evaluating Schema Evolution Support and Qualitative Features	56
5 Discussion	59
5.1 Interpretation and Evaluation of Results	59
5.2 Outlook	60
6 Appendix	61
Bibliography	65

List of Figures

1.1	Object serialization and deserialization	2
2.1	Defining components of a serialization protocol	5
2.2	Graph representation of example data structure	9
2.3	References converted to copies after serialization and deserialization	11
3.1	<i>dCache</i> domain topology: multi-star	16
3.2	Serialization of <i>dCache</i> messages and envelopes	17
3.3	Detailed ratings of the serialization framework criteria	20
4.1	<i>JOS</i> (de)serialization speed for <i>typelist</i> objects	42
4.2	<i>JOS</i> and <i>Kryo</i> (de)serialization speed for <i>composite</i> objects	43
4.3	<i>Kryo</i> (de)serialization speed for <i>typelist</i> objects	43
4.4	<i>FST</i> (de)serialization speed for <i>typelist</i> objects	44
4.5	<i>FST</i> and <i>Hessian</i> (de)serialization speed for <i>composite</i> objects	44
4.6	<i>Hessian</i> (de)serialization speed for <i>typelist</i> objects	45
4.7	<i>Protobuf</i> (de)serialization speed for <i>typelist</i> objects	46
4.8	<i>Protobuf</i> and <i>Protostuff-runtime</i> (de)serialization speed for <i>composite</i> objects . .	46
4.9	<i>Protostuff-runtime</i> (de)serialization speed for <i>typelist</i> objects	47
4.10	<i>Avro</i> (de)serialization speed for <i>typelist</i> objects	48
4.11	<i>Avro</i> and <i>Avro-Json</i> (de)serialization speed for <i>composite</i> objects	48
4.12	<i>AvroJson</i> (de)serialization speed for <i>typelist</i> objects	49
4.13	Ratio comparison of IntList (de)serialization speeds	50
4.14	Ratio comparison of DoubleList (de)serialization speeds	51
4.15	Comparison of StringList (de)serialization speeds	51
4.16	Comparison of (de)serialization speeds of C0 and C1	52
4.17	Comparison of (de)serialization speeds of C2 and C3	53
4.18	Comparison of (de)serialization speeds of Composites C4 and C5	53
4.19	Comparison of <i>PMPUM</i> (de)serialization speeds	54
4.20	Comparison of encoding sizes of IntList and DoubleList objects	55
4.21	Comparison of encoding sizes of StringList and <i>PMPUM</i> objects	55
4.22	Comparison of <i>Protostuff</i> (de)serialization speed on different machines	56

List of Tables

2.1	Characteristics of a serialization protocol	6
2.2	Examples of zigzag encoded signed integral values	13
2.3	Examples of variable length encoded signed integral values	13
3.1	Results of the <i>dCache</i> developers' ratings of serialization framework properties .	19
4.1	Criteria for qualitative evaluation of serialization frameworks	30
4.2	Comparison of test-machines	33
4.3	Schema evolution support of serialization frameworks	57
4.4	Qualitative evaluation of serialization frameworks	58

List of Listings

2.1	Declaration of example data type "User" (Java)	8
2.2	Instantiation of example data type "User" (Java)	8
4.1	Category <i>Typelist</i> : class IntList	34
4.2	Category <i>Composite</i> : class Msg (<i>C0</i>)	35
4.3	Category <i>Composite</i> : class Msg_rw01 (<i>C1</i>)	35
4.4	Category <i>Composite</i> : class Msg_rw02 (<i>C2</i>)	36
4.5	Category <i>Composite</i> : class Msg_rw03 (<i>C3</i>)	36
4.6	Category <i>Composite</i> : class Msg_rw04 (<i>C4</i>)	36
4.7	Category <i>Composite</i> : class Msg_rw05 (<i>C5</i>)	37
4.8	Category <i>dCache-like</i> : class Message	38
4.9	Category <i>dCache-like</i> : class PoolManagerPoolUpMessage	38
4.10	Category <i>dCache-like</i> : class PoolV2Mode	39
4.11	Category <i>dCache-like</i> : class PoolCostInfo	39
6.1	Singularity recipe file for benchmarker deployment	61
6.2	<i>Typelist</i> schema descriptions for <i>Protobuf</i>	61
6.3	<i>Typelist</i> schema descriptions for <i>Avro</i>	62
6.4	<i>Composites</i> schema descriptions for <i>Protobuf</i>	62
6.5	<i>Composites</i> schema descriptions for <i>Avro</i>	63

1 Introduction

This thesis compares and evaluates different data serialization protocols in computer science with the objective of replacing and improving upon *Java Object Serialization (JOS)* [16] in the distributed storage system *dCache* [21].

The criteria for choosing a new framework are collected, analyzed and formalized. 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. 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.

1.1 Introduction to the Problem Area

Distributed systems are becoming more and more prevalent [7]. A *distributed system* consists of multiple independent logical or physical components that are connected by a network and work together by communicating with each other [19].

Ideas of modularity [12] have in recent years evolved into a new fundamental approach to software architecture, wherein software is constructed as a composition of fine-grained, loosely-coupled, independently deployable service modules. This concept has been termed *microservices*. The properties of such a system can notably improve scalability, reliability and agility [6], while problems may arise from difficulties in determining the granularity of services to create, ensuring security, managing distributed storage, application testing and resource consumption [17].

Microservices represent a way to implement a distributed system. The rising number of potentially heterogenous services in such a system leads to an increased message exchange. Each of them might be received by a service written in another programming language and running in a different environment than the sender's, and must be understood nonetheless.

In other words, these message objects need to be *serialized* – translated into a format that can be transmitted or stored – by the sender in an efficient, language- and system-agnostic way, then passed over to the recipient and *deserialized* – reconstructed – to a format that he can interpret (see figure 1.1).

While some of the most prevalent serialization formats nowadays are plain-text, human- and machine-readable, most prominently Extensible Markup Language (*XML*) and the more compact JavaScript Object Notation (*JSON*). They are usually slower and larger in size than binary formats. The native *Java object serialization* serializes to such a binary format. It has been added to the *Java* language in 1996 [16] 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 [5]. Besides making life difficult for developers by having to consider hidden

Figure 1.1: Object serialization and deserialization

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 serialization frameworks and serializes exclusively to a *Java*-specific format that cannot be reconstructed outside of the *Java Virtual Machine (JVM)*.

This thesis evaluates different serialization frameworks with the aim of replacing the native *Java* serialization for message encoding in *dCache*, a distributed system of microservices written primarily in the *Java* programming language [30]. The *dCache* software is an open source project that is developed to manage the storage of and efficient access to large amounts of scientific data. The *dCache* project commenced in the year 2000 and is now a joint effort between the *Deutsches Elektronen-Synchrotron (DESY)* in Hamburg, the *Fermi National Accelerator Laboratory* near Chicago and the *Nordic e-Infrastructure Collaboration (NeIC)*.

Many research centers around the world use *dCache* to store their scientific data, *DESY* alone hosts seven instances for different experiments or applications on the campus. One of the larger among them, which stores data for the *CMS* experiment of the *Large Hadron Collider* at *CERN*, contains about 8.7 PiB of data in 31×10^6 files (spring 2019). In the time from January until the mid of April 2019, the *CMS* billing data recorded 282,108,449 data transfers, which does not even include deletions or other requests. These commonly occur in bursts of ingesting large amounts of oftentimes small files in a short period of time. Each one of those operations is internally realized by sending several messages between different services.

Within *dCache*, 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. Being able to easily communicate with external services due to usage of a platform-independent serialization format is of additional, albeit secondary interest.

1.2 Related Work

The first description of a serialization method in a programming language was introduced by Herlihy and Liskov [8] and implemented as an example in the language *CLU*. It already discusses aspects such as safety and abstraction from the encoding format. The *Java* object serialization as described in [16] is clearly rooted in *CLU*. It serializes not merely object graph states but their full definitions. It also does not separate the serialization process from the resultant format.

In [4], Evans describes why object serialization in *Java (JOS)* is inappropriate for providing persistence. The primary motivation is that although it is initially simple to work with, the challenges of advanced usage are difficult to foresee and handle. The paper also evaluates the complexity of using *JOS* as a means of persisting objects in a distributed environment when evolving classes. It finished with discussing alternatives to using *JOS* for persistence. Already in the year 1999 Philipsen, Haumacher and Nester designed a new Remote Method Invocation

(RMI) as a drop-in replacement for the native *Java* one, which is analyzed as being too slow, especially for use in high performance computing [14].

In [10], twelve different object serialization libraries are compared for the *Java* language with regard to language support, schema necessity and compiler availability as well as encoding format. It additionally measures encoding sizes and (de)serialization speeds, but does not implement necessary best practices for *Java* benchmarking and consequently presents unreliable results, which are very likely influenced by *Java Virtual Machine* optimizations. The compared serialization frameworks range from plain text encodings such as *XML* or *JSON* to binary encoders such as *Protocol Buffers* or *Thrift* and aim to compare their performance in a general sense. Bittl et al. analyze different data serialization schemes to evaluate performance with regard to their usage in wireless Car-to-X communication [2]. They compare the *Efficient XML Interchange (EXI)* format with *ASN.1* encodings and the frequently used alternative *Protocol Buffers*. Because of bandwidth limitations, the encoding size is of crucial importance, but they also evaluate the memory footprint on computation and the (de)serialization time using three different processor technologies. They conclude, that the results for the relation between processing speeds is the same on the different platforms and that *Protobuf* encoding sizes are too large to be used in car-to-x communication. It is also in most cases outperformed by *ASN.1*. These results were, however, not obtained in a *Java* based environment, are not based on pre-existing object structures but use examples of different standardized *ETSI ITS* messages used in automotive communication and focus on areas of memory consumption and encoding size that are only of secondary interest to the scope of this paper. There are different serialization frameworks for different specific use cases, such as one for resource-constrained devices in wireless sensor networks [13], which is implemented in *Java* and supports code generation in different languages. Most are, however, optimized for specific features that might not be optimal in other scenarios. Therefore, a thorough evaluation of *Java*-specific, speed optimized serialization protocols is necessary with regard to the specific requirements of the *dCache* system.

1.3 Structure of the Thesis

The thesis is organized into several chapters. After the introductory part, chapter 2 will be giving an overview of serialization and the problems that are aimed to be solved. Different existing approaches will be introduced.

Chapter 3 introduces the way serialization is currently implemented in the *dCache* software, especially with regard to its current use of *Java serialization*. The advantages and disadvantages of the current approach are analyzed, then several characteristics of serialization protocols/frameworks are introduced. A survey is conducted among the active *dCache* developers and the criteria ranked by importance accordingly.

Chapter 4 will then commence to compare several selected serialization protocols, by first introducing each of them, describing their core features and approaches as well as explaining why they were chosen. The next section will define criteria, by which these frameworks will be evaluated. Afterwards, the evaluation of (de)serialization performance commences with descriptions of the used environment and tools, then the implemented data structures and benchmarking setup. Subsequently the results for individual benchmarkers are introduced, followed by comparisons of protocols and different environments. The last section presents the results of analyzing schema evolution support and qualitative features.

These results are summarized, discussed and evaluated in chapter 5, which concludes with an outlook.

2 Serialization Overview and Classification

Serialization is the process of translating data structures into a format that can be transmitted or stored. *Deserialization* is the process of reconstructing a serialized data representation (see figure 1.1) [18].

The data structures to be serialized are usually the at-runtime representations of data in a software program, for example in the form of C-type structs or object states in Object Oriented Programming (*OOP*). When (de)serializing objects, the process is frequently referred to as (*un*)*pickling*. Data structures are in this context defined as instances of a data type, denoting that they are filled with concrete values, and will also be called *message* or *data object*. The serialized data representation, in the further progression of this thesis referred to as *encoded* or *serialized object/data (structure)*, may be human-readable or binary.

Serialization is commonly used for tasks such as transferring data, triggering remote procedure calls (*RPC*), creating a deep copy of an object, wherein all referenced objects are cloned, or encoding data for persistent storage.

We define a *serialization protocol* as a combination of the *format of the initial data structure*, that can be serialized, the *format of the resulting encoded data*, and the *method of translating* a data structure into its respective encoded representation (see figure 2.1).

Figure 2.1: Visualization of the defining components of a serialization protocol

Important characteristics of a *serialization protocol* are listed in Table 2.1.

A *serialization framework* is a software framework, which contains one or more implementations of *serialization protocols*. It can additionally be evaluated with regard to its programming language bindings, when an abstract schema is used for defining data structures, and its general *usability* characteristics. This chapter aims to analyze properties of *serialization protocols* and will therefore not discuss these aspects.

The *speed* of serializing or deserializing is not a fixed value for each serialization protocol, but represents the time it takes for serializing or deserializing a concrete data structure using a specific serialization protocol in a certain environment. The same applies to the *size* of encoded data: it is the size of the encoded form of a concrete data structure, that was serialized with a specific serialization protocol.

Characteristic	Explanation
Speed	The (de)serialization speed
Encoding size	The size of the produced encoded data
Data structure flexibility	The flexibility of the (de)serializable data structures
Data structure format	The supported format, data types and type safety of the (de)serializable data structures
Encoding format	The encoding format and its possible platform independence

Table 2.1: Characteristics of a serialization protocol

In general, these quantities can be conceptualized as functions, requiring the following parameters: the *serialization protocol*, the used *data structure instance*, which is filled with values, and in the case of duration evaluation, also the properties of the *computing environment*.

The size of the resultant encoded data, $Size_{ser}$, depends on the *serialization protocol* and the *computing environment*. It is a fixed value for each pair of parameters and independent of the properties of the machine it is calculated on. An encoding may, however, vary in the ordering of serialized elements.

The speed of serialization, $Speed_{ser}$, and deserialization, $Speed_{deser}$, is additionally coupled to the hardware and software environment that the evaluation is performed in. A value is always bound to a running software instance, which is influenced by a large number of system parameters. The time it takes to execute a sequence of operations can be *measured*, but this is an inherently imperfect process, that is subject to inaccuracies of many kinds. These range from unknown systematic effects such as scheduling overhead, CPU hopping or additional load generated by other tasks, to statistical fluctuations of the measured sizes. Means of mitigating these problems are addressed in a following chapter.

If we assume all parameters to be fixed and the results of these measures to consequently be a static value for each combination of parameters, then their sizes (de)serialization speed and encoding size) are primarily governed by a selection of trade-offs between different desired features in a serialization protocol (disregarding a possibly sub-optimal implementation).

The other features from Table 2.1 are discussed in more detail in the following sections. These are the flexibility of the (de)serializable data structures, its supported data types and associated type safety aspects, as well as the format of the encoded data, which includes the property of platform independence and certain encoding approaches. As a first step, it is necessary to understand the structure of serializable data. Therefore, ways of representing and characterizing complex data structures are introduced and analyzed, which are then discussed with regard to the process of serialization and accompanied difficulties.

2.1 Data Structures As Directed Graphs

In order to analyze data structures and their representations, the following definitions and notations will be used. A *graph* is an ordered pair $G = (V, E)$ with a set of vertices V , also called *nodes*, and a set of edges E , which are connections between two vertices. An *undirected graph* is a graph, whose edges are unordered pairs of vertices: $E = \{ \{u, v\} \mid u, v \in V \}$. A *directed graph* or *digraph* is a graph, whose edges are ordered pairs of vertices: $E \subseteq V^2$. An undirected graph can be described as a directed graph with E being symmetrical. A *loop* is defined as an edge that joins a vertex to itself. If a graph has no loops, the following holds: $\forall (u, v) \in E \implies u \neq v$.

A graph having parallel edges is called a *multigraph*. A graph that has no loops and no parallel edges is called a *simple graph*.

In a directed tree, the *predecessors* of a vertex are defined as $V_{\text{pred}}(v) = \{p \in V \mid (p, v) \in E\}$, the *descendants* of a vertex are defined as $V_{\text{desc}}(v) = \{d \in V \mid (v, d) \in E\}$. A *walk* in a graph $G = (V, E)$ is a sequence of alternating vertices $v_1 \dots v_n \subseteq V$ and edges $e_1 \dots e_m \subseteq E$, with $n, m \in \mathbb{N}$ and $n = m + 1$, starting and ending with a vertex: $v_0 e_0 v_1 e_1 \dots e_m v_n$ where $e_x = (v_x, v_{x+1})$ and $x \leq m$. It is said to be *closed*, if $v_n = v_0$, otherwise it is called *open*. A *circuit* is a closed walk with every edge being distinct, a *cycle* is a closed walk with every vertex being distinct (except for the first and last one). A *path* is a walk with no repeated vertices. A graph is said to be *connected*, if there is a path between every pair of vertices, *acyclic* if it has no cycles. A *tree* is a simple graph that is connected and acyclic. A *directed acyclic graph* (DAG) is a digraph with no directed cycles.

A labeled graph is a graph, whose vertices or edges are labeled, which means that there exists a mapping $\alpha : V \rightarrow M$ or $\beta : E \rightarrow N$, where M, N are the respective label sets, often containing numbers.

A data structure is a collection of organized, hierarchical data values. It can be represented as a labeled, directed graph, which may contain cycles and loops.

A *node* shall be defined to be a vertex that is labeled `<datatype>`, which can be any type other than a basic one. A *node* n may have predecessors $V_{\text{pred}}(v)$ and descendants $V_{\text{desc}}(v)$.

A *leaf* will be defined to be a vertex that is labeled with `<datatype>:<value>`, where `<datatype>` is a string representing a supported data type and `<value>` being a string representation of a value that is valid according to the aforementioned `<datatype>`. A `<datatype>` may either be a basic type or else its corresponding `<value>` must be `null`. A *leaf* l will always have predecessors $V_{\text{pred}}(l) \neq \emptyset$ but never descendants $V_{\text{desc}}(l) = \emptyset$. It can be understood as a typed field value in a data structure.

A directed node connecting two vertices a to b is labeled with `<fieldname>`, which can be understood as the name of the field by which a knows b or the unique position within an a type that is made up of an ordered list of values.

A data structure according to this definition always includes at least one *node*, which serves as the outermost data container. The graph is assumed to be externally named, so that the initial node can exist without being pointed to by a `<fieldname>` tagged edge.

In programming languages, the *declaration* of types as the building blocks of hierarchical data structures are separated from the *instantiation*, where they are combined and filled with values. In object oriented programming, there are basic and class types. A class may contain data and methods. A class b (subclass), that *inherits* from a class a (superclass), contains everything of a but can add fields and methods or override the latter. *Subtype-polymorphism* describes the concept that a subclass-typed object can be assigned to a superclass-typed field, as described by the *Liskov substitution principle* [9]. In this way, the general structure of an object tree can only be described abstractly by its class definition and the collection of all the classes and subclasses of fields therein, which will be referred to the *schema* of a data structure. The object tree of a concrete *instantiation* of such a class can only be determined dynamically at runtime.

To illustrate how such an object tree can be represented as a graph according to this definition, an example `User` data structure is declared and instantiated in *Java*, then translated into its graph representation. In Listing 2.1, the data type a constructor is assumed to be present and set the values accordingly, which is used in the instantiation shown in Listing 2.2. There, a `User` object is instantiated in a way that creates several interesting features that need to be translated

to a graph representation, which can be seen in Figure 2.2. The two `File` fields are assigned the same reference, which is correctly represented in the graph. The `List<File>` points to this value with its first element field, to another with its second one. Both of them have a `User` field, one of them is assigned the `null` type and thereby creates a non-basic field *leaf* node, the other points to the initial `User` object, creating a cycle in the graph. Finally, the `User` object contains a reference to itself, creating a loop.

```

1 class User {
2     String username = "anon";
3     int userid;
4     User favUser;
5     File favFile;
6     File leastFavFile;
7     List<File> viewedFiles;
8     // ...getters and setters ...
9 }
10
11 class File {
12     User owner;
13     String filename;
14     int permissions;
15     String content;
16
17     File(User owner, String filename, int permissions, String content) {
18         ...
19     }
20     // ...getters and setters ...
21 }

```

Listing 2.1: Declaration of example data type "User" (Java)

```

1 class Instantiator {
2     User createUser() {
3         User user = new User();
4         user.setUsername("anon");
5         user.setUserId(34);
6
7         File ambivFile;
8         ambivFile = new File(null, "myAmbivFile", 777, "ambivalent file content");
9         user.setFavFile(ambivFile);
10        user.setLeastFavFile(ambivFile);
11
12        File otherFile = new File(user, "otherFile", 777, "other file content");
13
14        List<File> viewedFiles = new ArrayList<>();
15        viewedFiles.add(0, ambivFile);
16        viewedFiles.add(1, otherFile);
17        user.setViewedFiles(viewedFiles);
18
19        user.setFavUser(user);
20        return user;
21    }
22 }

```

Listing 2.2: Instantiation of example data type "User" (Java)

Figure 2.2: Graph representation of instantiated example data type "User"

2.2 Flexibility of Data Representation

There are two types of structured information, which are needed in order to fully represent a data structure: an abstract *schema*, which stores the type hierarchy, and the *instantiation*, storing the values with which it is filled.

Many languages have built-in support for *serialization*, as it is called in *Java*, or *pickling*, which is the name given to the native *Python* protocol, both of which support compact binary encoding formats [18]. These mechanisms serialize *invisibly* by traversing object or type structures dynamically at runtime, building up an internal representation of its schema. We will be referring to frameworks with this feature as *full object graph serializers*. They are therefore flexible with regard to a change of a data structure, serializing what they find. They need to remain backwards compatible, however, if there is a chance that they might in the future be asked to deserialize data that was encoded using an old data structure layout.

Most external serialization frameworks do not have a language integration, that is similarly deep. There are a few *full object graph serializers*, however, which automatically generate a schema by traversing a data structure graph at runtime, such as *Hessian* [29], *Kryo* [24], *Protostuff* [25], *fst* [26] and *JBoss Marshalling* [31]. They aim to invisibly serialize data structures, in these cases primarily *Java* objects, in order to create a drop-in replacement for the problematic native *Java* serialization. They oftentimes continue to suffer from many of its problems themselves, by mostly addressing and improving upon certain aspects. In *Java*, classes and member variables can be annotated, which might then be interpreted by a framework, that can use this information to

generate a respective *XML* or *JSON* representation of an object at runtime or reconstruct it from an encoded version.

Most frameworks, however, are based on a schema description language, or *interface description language* (IDL), which allows the abstract definition of a data structure schema. Based on this schema, an instance may be created, that can be filled with values and encoded into a supported serialization format. Those might include human-readable variants like *JSON* or *XML*, but also often binary formats, that are optimized in different ways. Defining a schema often has the advantage of enabling faster (de)serialization, because the transient data structure in the program's execution context does not need to be traversed every time to infer a schema. It might also be desired, that the fields, which should be serialized, can be individually selected and accessed in a controlled way over the specified *getter* methods, rather than possibly being retrieved by invasive, encapsulation-violating routines a *full object graph serializer* might use.

Both approaches for creating an internal representation of the structure of data, that is to be serialized, may try to solve the problem of schema evolution: the ability of a deserializing routine to deserialize encoded data, that was based on an older (*backwards compatible*) or newer (*forwards compatible*) version of the data structure's schema. This is often done by including a schema-version specific identifier, which is never reused, tracking the current and previous schemata with their identifiers and updating the current version ID and structure with a change to the schema. The encoded data is then mapped to the respective deserialization function, when the schema version has been identified. Implementing support for *schema evolution* may lead to a larger $Size_{ser}$ by having to store field identifiers or increased $Speed_{ser}/Speed_{deser}$ due to handling changes requiring additional logic.

Schema evolution can include support for different types of changes for different serialization protocols. It can include compatibility with just the last/next version of a schema, or all previous/following ones, making it *fully transitive*. It can include support for deletion, addition or modification of any fields, just those marked as being *optional* or merely specific types, for example *integer* being transformed to *long*. It can enforce a certain sequence of updating *serializers* and *deserializers*. There are also frameworks, which do not support schema evolution at all and delegate the version management to the software developer.

2.3 Data Types, Type Safety and Security

The supported data types of a serialization protocol must be defined, in order to be able to encode and reconstruct values of different length and content. *Full object graph serializers* often support all the data types of the language, whose data structures they traverse, without the user having to manually translate them. By mapping native data types to internally defined ones, language interoperability may be achieved, where the respective API invisibly translates received types to their native representations and vice versa.

Within the scope of this thesis, *basic data types* will mean single-valued types that serve as the basic building blocks in a programming language. Having many differentiated size varieties of *basic data types* potentially allows a more compact encoding by reducing the number of unnecessary stored bytes. If there was only one large numerical data type, which can be used to encode the largest occurring numbers in a program, serializing a small one would take up equally as many Bytes:

Figure 2.3: References converted to copies after serialization and deserialization

$$\begin{aligned}
 (4\,294\,967\,295)_{10} &\hat{=} (1111\,1111\,1111\,1111\,1111\,1111\,1111)_2 \\
 (1)_{10} &\hat{=} (0000\,0000\,0000\,0000\,0000\,0000\,0001)_2.
 \end{aligned}
 \tag{2.1}$$

Here, a 32 bit signed integral data type is used to store the largest and one of the smallest numbers possible therein. In the second case, this results in many prefixed zeros, which are unnecessary for interpreting its value. In most programming languages, numerical data types, (un)signed integral and floating point, are each represented by a 32 and 64 bit variant. The integral types are usually called *integer* (32 bit) and *long* (64 bit), the floating point types named *float* (32 bit) and *double* (64 bit). Sometimes there is also a 16 bit version of the integral type, called *short*, and some serialization frameworks even support an eight bit numerical type. Most programming languages furthermore have a *char* type requiring 16 bit, that is commonly used for storing or a Unicode character, a *byte* data type, which is stored using 8 bit, and a *boolean* type for representing the values *true* or *false*, which can therefore be stored in a single bit. *Enumerations* are sometimes understood as composite data types, a field whereof can contain a single value from a predefined set of values.

A *composite data type* in this context refers to a data type, which can be composed of basic or other composite data types. The perhaps most widely implemented type therein is the *structure* or *struct* type, a simple container including a number of fields of a defined data type, including other structs. This theoretically enables arbitrarily deep nesting. Every serialization library supports such a *struct* concept. Some other widely used structures in this category are *lists* or *sets* containing an at runtime varying number of sorted or unsorted values, respectively, that are all of the same type. A *string* data type may, for example, be represented as a sequence of *char* typed values. Most serialization frameworks define a way of representing lists/arrays or sets. Additionally, a *map* data type is sometimes known, which can contain a set of unique *key* values of a specific type, each associated with a value of another predefined type. Most programming languages and serialization frameworks include the concept of an empty type, frequently called *null*, to serve as a placeholder for an empty field of a composite data type.

In *object-oriented programming* (OOP) languages, an instantiated class, a so-called *object*, is an example of an instantiated *struct* composite data type. Within the runtime environment of such a program, *references* to objects, rather than the objects themselves, are assigned to fields of their class data type, which may result in multiple fields pointing to the same exact object, which exists only once. Most serialization frameworks do not implement such a concept. They would serialize the object corresponding to the graph *a* on the left side of Figure 2.3, which has three references to an object *b*, as having three copies of the same object as displayed on the right side. This results in taking up three times as much space for object *b* as would be necessary.

Serialization and especially deserialization is a well-known and often abused security risk. Reconstructing an object or other data structure from an encoded stream of bytes can be dangerous, when the data structure schema is unclear. If a malicious third party has access to serialized

data, they can read and manipulate it. Deserialization of external input by processing it within one's software can potentially lead to abuse of application logic, obtaining references to private members, denial of service or in the worst case arbitrary code execution. Input validation is crucial, but highly difficult. Deserializing untrusted data should be entirely avoided if possible, a good approach includes using cryptography in combination with authentication. Enforcing *type safety* should be implemented by knowing, expecting and validating input against a specific data structure schema or set of supported types.

Many serialization frameworks require the definition of a data type's schema include a *code generation* feature for several programming languages. It allows generating code modules from a schema definition for type-safe writing and reading of fields to and from a respective (encoded) data structure.

2.4 Format of Encoded Data

The internal representation of an instantiated data structure needs to be encoded into a sequential format, that can be transmitted or stored. A format can be optimized for a specific, desired feature, such as having a small size of the encoded data or desirable speed of the encoding/decoding process. Minimizing the size of the encoded data often requires more time for processing, while high speeds frequently result from reducing compression efforts, leading to redundancies in the written data and consequently increasing sizes. Many frameworks support more than one encoding format, which can be chosen from, each for a different purpose, such as a human and machine readable quasi-standard format like *JSON*, that can easily be interpreted by another application or used for debugging, or a more efficient binary one.

Some serialization formats are platform dependent, which often leads to better performance and smaller encoded size than a corresponding platform independent format. The reason behind this is that platform-specific optimizations can not be used and the characteristics of each type need to be redefined, such as its used *endianness*. An encoding can be self-describing, meaning it can be interpreted without additional external information, or may achieve a small encoding size by not including type information or field names at all, but requiring knowing the writer's schema for that data structure when reading it. Some serializers need to read in a data structure completely in order to extract a value, some can read parts directly without needing to parse and unpack the structure to a secondary representation.

When serializing to a binary format, fields in a data structure can be stored in combination with their locally-unique identifier, which enables trivial field name changes, not having to store a single byte for unused optional fields and more simple integration of schema evolution support by assuming that such an identifier is never reassigned. If merely stored sequentially without a field identified, the encoding will be more compact, but evolving the schema will be more difficult.

The size of the binary encoding of different basic and composite data types is often optimized using similar approaches. One important strategy is foregoing the typically implemented byte-alignment of values, so that no padding is necessary, which will likely shrink an encoding's size. Additionally, optional and unused fields, as well as potentially even those filled with their respective default value, can take up zero bytes, instead of encoding it as a specific magic number or the explicit default value. Storing *enum* values by integral numbers, that are mapped to the defined types, is also more space-efficient than storing string-representations.

If numerical data types can be put into one of several differently sized type containers, which was shown to be more space-efficient in Equation 2.1, they might still suffer from unnecessarily stored bits. Although there may be a small, discrete number of integral data types in a schema

Original (decimal)	Encoded As (decimal)
0	0
-1	1
1	2
-2	3
2	4

Table 2.2: Examples of zigzag encoded signed integral values

Original (decimal)	Encoded As (binary)
1	0000 0001
-1	0111 1111
63	0011 1111
-63	0100 0001
300	1010 1100 0000 0010

Table 2.3: Examples of variable length encoded signed integral values

definition or programming language, integral type encoding can be further optimized by internally storing them in a variable-length format (*varint*), which means reserving the high-order bit of each byte to indicate whether the following byte is also part of the current number. The number itself may then be stored as the *two's complement* representation in the remaining 7 bit of each byte, starting with the least significant group (see table 2.3, where the further bytes indicator is colored orange). To avoid storing the maximum amount of bits in the case of negative numbers, which are in two's complement notation indicated by a 1 as the most significant bit, a *zig-zag* coding may be used. This technique maps signed numbers to unsigned ones, so that those with a small *absolute value* have a small encoded value as well (see Table 2.2 for examples). Numbers of 32 bit (n_{32}) or 64 bit (n_{64}) are encoded using

$$\begin{aligned} n_{32} &\rightarrow (n \ll 1) \hat{~} (n \gg 31) \\ n_{64} &\rightarrow (n \ll 1) \hat{~} (n \gg 63), \end{aligned} \tag{2.2}$$

where $\hat{~}$ is the bit-wise *XOR* operation and $\gg$ an arithmetic shift.

Floating point numbers are more difficult to optimize and are often just converted to their equivalent bit-count sized integral representations.

List types need to store the number of included values, which is often simply prepended as a fixed-size integral number, thereby statically limiting the number of storable values. The same is frequently used for encoding strings, which are represented as a list containing a certain number of characters, or the often included *byte list* type for storing arbitrary data. Encoding *struct* types may just include storing the values of its included fields.

The choice of an encoding format and optimization strategy heavily depends on the use case, which can vary from relying on static, rarely changing data structure formats, focus on readability or universality of the encoding and interoperability with external modules, to highly time- and/or space-critical applications, for example in the case of *wireless Car2X communication* [2].

3 dCache and its Use of Java Serialization

The *dCache* software 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 different qualities of data storage (quality of service, *QOS*), including tertiary storage support, for which it is able to use disks as a caching layer [11].

Within the *dCache* architecture, a microservice is termed *cell* and represents *dCache*'s most fundamental executable building block. Each cell fulfills a specific task and can be grouped into a certain type category, for example *pool* for storage or *door* for enabling access to data over a specific protocol, for example *NFS*, which then gets translated into internal instructions. A *domain* is a container for cells that runs in a single *Java Virtual Machine (JVM)* instance. The included cells, each running as a separate thread therein, can be arbitrarily chosen but are usually combined in a way that is deemed useful to achieving a certain objective together. A domain's name is required to be unique within a *dCache* installation.

A minimal *dCache* instance consists of a single domain housing specific cells. A *door* and a *pool* type are necessary for accessing and storing data. Two management components are required; a *pnfsManager* that supervises the metadata namespace, and the heart of the *dCache* system, a *poolmanager* that manages all the pools in a system and regulates data dispatch. As an external dependency, an *Apache Zookeeper* [39]) instance is also necessary as a distributed directory and coordination service. New cells can easily be added to an instance. Growing the storage capacity, which often also improves the performance of an instance, just requires the addition of new pools. In order to provide resilience of an instance to node failures, *dCache* supports running in *high availability (HA)* mode, wherein redundant cells for a service can be run in parallel, such as having multiple *pnfsManagers*.

The cells within *dCache* communicate with each other via messages, addressing other cells with their globally unique identifier `cellName@domainName`. If the `domainName` suffix is missing, the addressed cell is either *well known* and globally unique, or a local one, in which case one could also use the suffix `local`. Within a domain, the *location manager* cell is responsible for creating *tunnels*, which are cells that can establish a TCP connection to another tunnel. When a connection is established, the corresponding *domain route* is added to each of the involved domains's *routing tables* by the *routing manager* cell and removed, when the tunnel shuts down. There are different types of routes for different use cases, for example *topics* that enable *publish-subscribe* messaging, in which each *subscriber* receives each published message, or *queues*, in which every *published* message is only consumed by a single *subscriber*.

For the sole purpose of message routing, we distinguish two types of domains: *core domains*, which register themselves in *zookeeper*, act as message hubs and forward messages from *satellite domains*. Every core domain is connected to every other domain, every satellite domain is connected to every core domain. This results in a star-topology with multiple, fully interconnected *core domain*

Figure 3.1: *dCache*'s domain topology: multi-star with fully interconnected centers

centers (see figure 3.1). In this way, every domain is connected to every other one by at most two hops via a single core domain.

3.1 Current Serialization of Messages Within dCache

There are many message classes in *dCache*, which are used for communication between cells. Not all of them continue being used and consequently sometimes one of them is removed. The following summary of existing message classes was undertaken using *dCache* version 5.2.3. The largest group of messages is made up of the ones inheriting from the **Message** class, which has 57 direct sub-classes, many of which have sub-classes themselves. Including itself, there are a total of 130 non-abstract message classes, which inherit from **Message**. There are additionally the isolated classes **CellMessage** and **CellPath** without sub-classes, as well as **JobInfo** and **DoorInfo**, each having one sub-class. There are also 5 non-abstract classes inheriting from the **StorageInfo** interface, 12 inheriting from the **ProtocolInfo** interface (two anonymous implementations) and 6 inheriting from the abstract **InfoMessage** class. Summarized, there are a total of 159 different non-abstract message classes, that might be serialized by a cell, sent to another one, then deserialized.

However, only objects of the **CellMessage** type are sent between cells. They serve as an envelope for the actual distinct message. In addition, they carry meta information about the message's source and destination (both of type **CellPath**), creation time, time to live (*tll*) and the timestamp of when it was received by the local domain. Also included is the unique message identifier (*umid*) and last *umid* (both of type **UOID**), if the message is persist (indicating that it is a *reply*) and its session information (type **Object**). The carried message object can either be stored in binary form in a *byte array* or in an **Object** type field, which is indicated by a **mode** flag and can therefore be serialized and deserialized independently of the **CellMessage** object it is contained in. In this way, deserializing a **CellMessage** object to access its meta information in order to decide what needs to be done with it, does not require deserializing the actual message object within (depicted in Figure 3.2), which significantly reduces processing time, when a **CellMessage** object just needs to be passed on.

The **CellMessage** class contains methods for serializing and deserializing the message content it is carrying. If the carried message is encoded, the **CellMessage** object can write its fields to a

Figure 3.2: Serialization of *dCache* messages and envelopes

`DataOutput` stream, or a new one can be created from a `DataInput` stream. Currently, *Java*'s inbuilt serialization is used for in these cases, which is problematic in several ways.

3.2 Problems and Advantages of Using Java Serialization

Java's inbuilt, JVM independent object serialization (*JOS*) or *pickling*, that is used for its remoting capabilities (*RMI*), has undoubtedly contributed to the language's popularity and prevalence. At the same time it has also lead to many problems, both for the language's developers as well as its users, that still negatively impact it to this day. In June 2019, an article by *Brian Goetz*, titled *Towards Better Serialization*, was published on the *OpenJDK* website, that extensively describes these problems and explores possible ways of improvement [5]. This section will give a brief overview of some on the main advantages and disadvantages of using *JOS*.

In [16], the automatic pickling of object states in the *Java* system was introduced for the first time. Its goals included simplicity, extensibility and type safety. It not only aims to serialize *data* structures, but also an object's *methods* and strives to do so completely transparently for the user. Implementing these goals has made *JOS* powerful and simple to use.

For a class to be serializable, it merely needs to implement the `java.io.Serializable` marker interface along with all the classes that it might reference at runtime as part of its object graph. `ObjectOutputStream` extending `java.lang.OutputStream` and `ObjectInputStream` extending `java.lang.InputStream` are classes that contain methods for serializing and deserializing graphs of objects and primitive types. The resulting stream of bytes is written to an `OutputStream` or read from an `InputStream`, respectively. When serializing an object by calling the `writeObject` method, it first pickles the specified object itself, then traverses the included references to other objects. In this way, it can recursively build up a serialized representation of the complete graph. References to the same object are preserved and will be reconstructed as a single object during unpickling.

There are many problems with the way *JOS* is implemented, however. One of the most obvious insufficiencies concerns the stream format, which is at the same time very *Java*-specific, not human readable and neither performant nor compact. It is also highly coupled to the pickling mechanism itself and consequently difficult to replace, without having to re-implement much of the underlying logic as well.

The way *JOS* was designed to work without much effort by the user, hiding its underlying mechanisms, is responsible for most of its other problems. It internally makes use of privileged, extra-linguistic, encapsulation-violating routines, that ignore accessibility restrictions and constructors. This makes it very difficult to check and validate the integrity of data, that is reconstructed, as well as profoundly complicating reasoning about security in general.

JOS also includes a mechanism for handling simple object structure evolution, which uses an implicit `serialVersionUID` field for each class. Evolving serializable classes is only very restrictively possible, however, making changes to a large software and especially library unnecessarily difficult, where backwards compatibility might need to be ensured for a very long time.

Finally, *JOS* also tries to do too much by serializing an entire program, including all methods and constructors, even though most use cases for serialization today merely involve encoding data. This leads to a larger overhead, both with regard to performance and size of the encoded data, but also generally unnecessary complications and risks by having to solve a more complex problem.

It is possible to exclude certain fields from being serialized via an annotation. The `readObject` and `writeObject` methods used for serializing an object can be overwritten to use a different encoding format. This, however, requires constructing and managing the versioning of the format as well.

Using a different library for serializing the contained *data* in objects can forego many of these problems. By opting to explicitly use field accessors, like getters, setters and constructors while deserializing fields and reconstructing objects, their validity can be better controlled by using Java's own fundamental concepts of encapsulation. Many serialization libraries additionally support schema evolution and can be configured to use one of several encoding formats, that may each be optimized for different desirable qualities.

3.3 Criteria for a new Serialization Protocol within dCache

The motivation for wanting to replace *JOS* for a different framework in *dCache* primarily results from a request's round-trip time no longer being sufficiently fast. This may be due to a combination of the system's ever-growing complexity and the increasing demands placed upon it, in the form of larger numbers of requests through different ways of accessing data and the generally increasing sizes of installations.

The *dCache* system has also in recent years expanded its integration of third-party software, most prominently *Apache Kafka* [33]. It may be used to collect logging messages for transactions within *dCache* in *JSON* format as an alternative to parsing the more idiosyncratically formatted and very slowly evolving logging files. *Transactions* include the **request** for moving a file in a communication between the client and *door*, the **transfer** of a file between client and pool, the **store** reporting a file being flushed to tape, the **restore** of a file from tape back to disk or the **remove** of a file. Currently, the messages sent to *Kafka* include just these few, hand-crafted ones, but a language-agnostic serialization format might allow directly passing on internal messages, or automatically converting them into a desired format.

Because there are many different parts within *dCache*, that may benefit from different properties of a serialization protocol and framework, a survey was conducted among the active *dCache* developers with the aim of ranking a created list of criteria by reported importance. Eight attributes were defined, that were considered to be most relevant for their serialization protocol specific properties, and those regarding a framework's maintainability and ease of integration into an existing system.

The developers were asked to rate each of the eight defined properties on a scale between 1 and 10, with the size of the number being positively correlated with the relevance of the rated criterion. Comments were desired but not required. It was furthermore not necessary to rate every item, if one did not feel confident in judging its relevance.

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

Table 3.1: Results of the *dCache* developers’ ratings of serialization framework properties

Seven of currently eight developers, not including the author, have partaken. The defined properties with their associated, averaged ratings are showcased in Table 3.1, all ratings are visualized in Figure 3.3.

The most important property was rated to be the ability of a new serialization protocol to be able to work in parallel with the currently used JOS (*A*). This is not surprising, because on the one hand, *dCache*’s upgrade model enforces backwards compatibility with the last two major versions, in which breaking changes are introduced, it would additionally also continue enabling rolling updates, in which parts of the distributed system run an older version, and on the other hand, because it would ease migration by allowing a smooth transition to the new technology through being able to add support for new messages one by one. However, this is not a technology selection criterium, but concerns demands on the way of integrating the chosen framework into the *dCache* architecture, independent of its nature.

Corresponding to expectations, however, the initial primary motivation for evaluating a replacement for JOS, namely improving the speed of message (de)serialization, was thereafter judged to be the most important property (*B*). Many developers described it as the highest priority and main reason for considering a change in the first place. Lower ratings were either uncommented, explained by not knowing of current performance impacts, or reasoned as serialization indeed being one of the main areas, that currently hurt *dCache*’s performance, but should not be prioritized at the cost of *A* or *C*. The deserialization speed is the slightly more important one of the two, because it occurs more often, for example when using *topics*.

Closely followed, achieving third place, is a serialization protocol’s ability to support schema evolution (*C*). The main arguments were based on backwards compatibility: the potential need to back-port field changes, maintaining support for running multiple pool versions in the same instance and rolling updates, without which distributed *dCache*’s would need a ‘big bang’ upgrade. They are closely related to the arguments regarding *A*, but in this case tightly coupled to the framework to be chosen.

The places 4 and 5 were filled by soft criteria regarding introduction effort and maintainability (*D*), as well as documentation and learning curve (*E*), which were generally regarded as being *nice to have*, but other things being more important, just like framework independence of the new encoding format (*F*).

Rather surprisingly, however, the platform and language independence of a framework/protocol ended up on the second to last place. It was generally regarded as being irrelevant, not essential

Figure 3.3: Detailed ratings of the serialization framework criteria

or even important. It was sporadically described as being *attractive* and just in one case rated with the highest score possible.

Interestingly, having a smaller size of serialized messages was very clearly deemed as the most irrelevant property. It landed on the last place, prominently offset from the ratings of the other items. *dCache* does not score the messages and the speed of a transmission will not be significantly impacted by a marginally reduced message size. Currently it should not be a concern, unless it affects performance, which would only be the case if message sizes increased significantly. This was regarded as being unlikely and then in all probability due to too much information.

To summarize, performance improvements appear to be the most relevant factor, as is frequently the case with serialization, closely followed by schema evolution. Soft criteria as well as platform and framework independence are of lesser interest, *nice to have* but not crucial. The size of an encoding is almost irrelevant, however. Because it is usually a significant factor in most of the published comparisons of serialization protocols, this creates a more unusual list of requirements.

4 Comparing Different Serialization Protocols

This chapter will be concerned with comparing different *serialization protocols* and the respective frameworks they are contained in, as defined in chapter 2 (*Serialization Overview and Classification*).

Because there are many ways of serializing a data structure into a specifically encoded representation, even from the same initial format, there are several serialization protocols that translate into a common encoding such as *XML* or *JSON*. To distinguish different protocols, a protocol will always be introduced within the specific implementation context of a framework.

This thesis will only analyze protocols that can serialize data structures from Java, or those created from more abstract schema definitions, where there is a corresponding *Java API*, that can be used for instantiation. This is reasonable due to the overarching goal of replacing JOS in the Java-based *dCache* system.

4.1 Serialization Protocols to be Evaluated

In the following, the compared serialization protocols are introduced, starting with the to-be-replaced native *Java* serialization. The others were chosen, because the respective frameworks are either easy to integrate as *full object graph serializers* or contain promising representatives of modern schema-based serialization approaches, have been shown to be high-performing, well supported and feature rich. This has led to increased adoptions, many reports and informal reviews, which form a good basis for pre-selections and choosing evaluation approaches. The frameworks are additionally required to be open source in order to be usable in the *dcache* project.

The popular *Apache Thrift* framework was excluded from this evaluation, because it is not ideal for the project in several ways. It is primarily an RPC framework that comes with a relatively large number of dependencies for creating client/server instances, but also includes a schema-based, cross-language serialization protocol. There is no *maven* integration or downloadable jar file to easily allow the necessary code generation from defined data structure schemas. Attempting to build the compiler on a Linux Fedora 30 system resulted in errors that had already been reported in 2015 for a Debian 8 system¹. A simple workaround requires symlinking to the `/` directory. This does not create the impression of *Thrift* being a very reliable or stable system. A consequently created docker container was over 8 GB large. The official documentation for *Apache Thrift* is additionally very poor. To conclude, introducing these dependencies into *dCache* would be a bad idea and in large parts even unnecessary.

The selected serialization protocols are grouped and introduced by framework, which is the unit that will later be integrated into *dCache*. It may also have relevant advantages or disadvantages in addition to the features of the included serialization protocols.

¹<https://issues.apache.org/jira/browse/THRIFT-3327>

4.1.1 Java Native Serialization

JOS as introduced in [16] was already discussed in section 3.2 (*Problems and Advantages of Using Java Serialization*). It is the *Java* programming language’s inbuilt, JVM independent object serialization implementation, that was introduced to the language in 1996. It is not an independent framework, but a specific serialization protocol.

One of its main features includes being a *full object graph* serializer that supports traversing an object’s dynamic reference graph at runtime, inferring its schema and directly encoding it. JOS is trivial to use and first introduced in a software, but hard to maintain because of its limited data structure evolution support and complex security vulnerabilities. It not only encodes an object’s state, but the entire program contained therein.

Compared to other protocols, JOS is not performant, its serialized format neither compact, despite being a binary encoding that is not human readable, nor language independent. This will become apparent in the following runtime and size comparisons. It can, however, communicate across JVMs and the JVM-based *Kotlin* [34] programming language can also use JOS’s routines, but non-JVM based languages cannot decode its binary format and reconstruct an equivalent representation.

4.1.2 Kryo

Kryo [24] is an open source, full object graph serialization framework for the *Java* language. It is developed by the *EsotericSoftware* [23] company that primarily develops *Spine*, a 2d skeletal animation framework for games. *Kryo* was first released to *GitHub* in 2013 with the aim of creating an easy to use, fast and efficient alternative to *JOS*. It is used in many large software projects, such as *Apache Spark* [37], the analytics software for big data. In its documentation it is recommended to use *Kryo* rather than *JOS* where possible. The only reason for it not being the default option in *Spark* is due to *Kryo* requiring the custom registration of classes, that should be serialized.

Kryo exclusively serializes to a *JVM*-specific binary format, that can be deserialized by *JOS*. There are different implementations for reading and writing data, called *serializer*. These are pluggable, but many commonly used ones are provided initially. *Kryo* can automatically encode objects that are not marked as `Serializable`, but does not support all types out of the box. Classes that should be serialized need to be registered in order to achieve better performance, wherein an object is associated with an internal identifier. When deserializing an object, it needs to have the same identifier it had during serialization.

Internally, variable length encoding can be used for `int` and `long` types, which decreases encoding size but also the serialization speed. It supports *chunked encoding* for writing data, which fills and flushes a small memory buffer multiple times instead of using a single, potentially very large one. *Unsafe buffers* may be used for increased serialization performance at the cost of encoded data being influenced by features of the system performing the serialization, such as its endianness.

By default, *Kryo* does not support references, meaning it will encode multiple references as multiple copies of that object. It will fail when encountering circular references. Support for references can be enabled, which impacts performance by needing to track every occurring object in a graph.

The `VersionFieldSerializer` enables backwards compatibility, wherein fields can be added, but retaining the ability to deserialize previously encoded data. Using the `TaggedFieldSerializer`,

fields can be tagged with an integral version identifier, enabling backwards compatibility including support for adding, removing or renaming of fields and optionally also forwards compatibility.

4.1.3 Fast-serialization

The framework *fast-serialization* [26] or *FST*, is a full object graph serialization framework for the *Java Development Kit* (*JDK*) that is intended as a faster replacement for *JOS*. It was created by Ruediger Moeller under an *Apache 2.0 license* and was first released in 2014. It reports to be used in many projects, such as the backend middleware of a stock trading company.

FST primarily supports serializing data to a *JVM*-compatible binary encoding and additionally a *JSON* format. Many serializers for different standard classes are included, falling back to using the standard *JOS* in cases of coming across unknown types. Custom serializers can be implemented. In order to improve (de)serialization speed, used classes can be preregistered. Serializer configurations need to be identical between receiver and sender, including having the same preregistered classes.

In its default configuration, *FST* detects and serializes cycles in both encoding formats. This feature can be disabled to improve performance, but breaks compatibility with *JOS*. Because an encoded object is only written once the used stream is flushed and always needs to be read entirely before being decoded, there is a limit on the size of object graphs that can reasonably be handled with *FST* of about 1,5 GB. Several other optimizations can be achieved by means of annotations, such as handling **String** type fields that are used like enumerations accordingly or conditionally decoding streams by skipping marked object fields. *FST* supports a concept called *structs*, which allows storing structured data as a packed, continuous chunk of memory that can be accessed directly without (de)serializing, resulting in significantly reduced encoding sizes and speeds. When not handled correctly, however, this unsafe direct memory access can lead to *JVM* coredumping.

By adding an integral valued **Version** annotation, backwards compatibility can be achieved, so that streams encoding an older version of an object can be decoded by the decoder of the current version, but not vice versa.

4.1.4 Hessian

Hessian [29] is a binary Web service protocol developed by *Caucho Technology, Inc.* under an open source *Apache Licence 2.0*. It includes a messaging remote procedure call layer and an object encoding protocol. The latter constitutes a full object graph serializer that is dynamically typed and portable across languages. Official implementations are available for *Java*, *Python* and *ActionScript*, and several others exist as third-party creations.

Hessian was designed to be self-describing, without needing an external schema for (de)serializing.

The *Hessian* binary serialization protocol defines several types, that are mapped to the respective native types of each supported language in the corresponding application programming interface. The basic types are **boolean**, the integral types **int** (32 bit) and **long** (64 bit), the floating point type **double** (64 bit), a **null** and a millisecond precision **date** (64 bit) type, as well as a *UTF-8* encoded **string** type. Supported composite or recursive data types include **list**, **map** and **object**. **Ref** items are used for encoding references.

In order to enable language independence, integral types that are shorter than or are as long as an **int**, such as *Java* **short** or **byte**, are internally mapped to the *Hessian* **int** type, floating

point numbers to its `double`. When deserializing such fields, they are extracted to these larger types, resulting in lossy conversions that reduce type safety.

Integral types can be serialized using variable-length encoding. Lists can be encoded as having a variable or fixed length, being typed or untyped. *Hessian* allows encoding objects in a compact way by only having to encode a class and its field names once, so that all following objects only need to serialize their values. *Hessian* uses three internal reference maps that allows it to store arbitrary graphs. They are used to store

- A. Objects/lists/maps
- B. Class definitions
- C. Type/class names.

The reference to each object that is encountered in a graph is added to A, its class definition, if not already present, is added to B. B and C improve efficiency by not having to store repetitions of common string items.

4.1.5 Protocol Buffers

Protocol Buffers, also called *Protobuf*, is a serialization library, encoding format and interface description language (*IDL*) compiler [22]. It is developed by *Google*, was initially released in 2001 internally, where it is still used heavily, and made available to the public as an open source project in 2008. There is also an RPC protocol stack based on Protobuf called *gRPC* [27].

Details about the *Protocol Buffer* framework can be taken from the official documentation pages [22].

Protobuf is language- and platform-neutral. It supports three encoding formats: its primary, binary one, a JSON representation and a textual format for debugging purposes. It was designed to be better than XML with regard to speed, simplicity and encoding size. Additionally, schema evolution support was one of the main design focuses, allowing backwards and forwards compatibility. Working with Protobuf involves using its specific interface description language to define the structure of data (*message definition*), which may be stored in a `*.proto` file, from which generated code can be used to write and read encoded data having the specified structure. Several languages are supported for code generation, *Java* is among them. A *protocol buffer*, a data object encoded in Protobuf's native binary format, is not self-describing and requires the message definition for interpretation.

We will be evaluating the newest and third major version of Protobuf, that has added support for different programming languages, some new features and included a new version of the protocol buffer language for message definition called *proto3*, which is not fully compatible with the predecessor *proto2*.

The structure of data is described using *proto3*, which has the `message` type as one of its most important building blocks, which functions as a *struct*-like container for different fields, where each of them is defined as a name-type pair. Messages can be nested and used as field values. Each field in a message is also assigned a unique field number that is used to identify the field in the message's binary format. The field numbers 1 to 15 are encoded using one byte, those between 16 and 2048 use two, etc., so it is a good strategy to assign low numbers to frequently occurring fields and to possibly even reserve a few for future additions. Fields can be defined as being `singular`, which is the default field rule that demands that each concrete message has zero or one instances of these fields, or `repeated`, wherein the field can occur zero or any other number of times and the ordering of them is preserved.

The *proto3* language supports different basic data types, called *scalar types*, that can be used in data structure definitions. In addition to `double`, `float`, `bool`, `string` (limited to 10^{32} *UTF-8* or 7 bit ASCII encoded characters) and `bytes`, which can store sequences of up to 10^{32} bytes, there are several optimizations for integral data types to choose from. Next to the normal, variable-length encoded `int32` and `int64` types, which are inefficient for encoding negative values, there are unsigned variants, explicitly signed ones that are more efficient for encoding negative values than their normal equivalents, and fixed-size 32 and 64 bit types that are more efficient if the values to be stored are frequently larger than 2^{28} and 2^{56} , respectively. There are defined default values for each type.

Additionally, *proto3* supports *enumerations* (`enum`), `oneof` semantics for specifying that only one in a certain set of singular field will be set, and `Any` types, which act as undefined embedded messages that may contain an arbitrary serialized message as bytes along with a URL for resolving its type. While `repeated` fields can serve to represent lists, *proto3* also contains an associated `map<key_type, value_type>` type, that cannot be repeated, where the key can be any integral or string type.

4.1.6 Protostuff

Protostuff is a framework that makes use of *Protobuf*. It enables generating a schema, either at compile time or at runtime. A schema is a class belonging to a specific data structure, that encapsulates its logic for serialization, deserialization, validation and construction of the data structure object. The module called *protostuff-runtime* allows generating such a class from existing *Java* objects by using reflection. This has the disadvantage of foregoing the explicit control of field identifiers provided by manually generated schemata, which is useful for schema evolution. In version 1.5.9, the newest at the time of writing this thesis, `@Tag` annotations can be used on class fields to control field numbers. To remove existing ones from being serialized, they can be annotated by using the `@Deprecated` tag.

4.1.7 Apache Avro

Apache Avro, referred to as *Avro*, is a data exchange and serialization library [20] that was developed in the context of the *Apache Hadoop* project [28] concerned with scalable, distributed computing, which it joined in 2009 and where it is still most prominently used for both storage and communication.

A detailed description of the *Apache Avro* framework can be found in the official documentation pages [20].

Avro is a platform- and language agnostic system that supports two serialization encoding formats: binary and *JSON*. *Avro* relies on data structure description *schemas*, which are required to be present when writing or reading data, making the combination fully self describing. This permits serialization with no per-value overhead, which can be particularly small and fast. *Avro* schemas are defined in *JSON* or by using its more human-specific *Avro IDL*. Through *dynamic typing*, *Avro*, as opposed to *Protobuf* or *Thrift*, does not require code to be generated from a schema definition in order to write or process encoded data, because it is accompanied by its schema definition, which facilitates the construction of generic data-processing systems. A code generator for several languages is included, however, because it may be used as an additional optimization for statically typed languages, such as *Java*.

A used schema may be stored along with a single serialized message in *single-object encoding* or a list of them in a so-called *object container files*, which may be required for long-term storage. Schema evolution is possible without the developer having to manually assign field identifiers, because differences can be resolved symbolically using field names when both schemas are known.

The structure of data that can be serialized is declared in the form of a schema definition and will be outlined for the *JSON* format. It is constructed from *JSON* strings, which contain one of the defined data types, *JSON objects* as *struct*-like containers for storing key-value pairs and *JSON arrays* as a union of other types. There are several basic types, referred to as *primitive*, of **null** for no value, **boolean**, **int** and **long** for signed integral values, **float** and **double** for *IEEE 754* floating-point values, **bytes** for a sequence of unsigned bytes and **string** for representing a sequence of unicode characters.

Composite or complex types supported by *Avro*, are **records**, which include its name, type and default value and can be used like any other type, **enums** for enumerations, **arrays** for lists of values, **maps**, which always have keys of type **string** and values of another specified type. There are also the aforementioned **union** types that are represented by an array of other types, which are all supported for a field of that **union** type. The most common **union** is that of a type with **null**. Finally, there are **fixed** types, **logical types** for representing derived types, and special types like **UUID**, **Date**, **Time** and **Duration**. This wide array of types supports rich data structures.

For evaluation, both its native binary encoding is tested, as well as its supported *JSON* format.

4.2 Criteria For Evaluation

The characteristics based on which different protocols will be compared are based on the general serialization protocol features defined in Table 2.1 and more specifically the *dCache* specific ones ranked by its present developers in Table 3.1.

One of the most important characteristics was determined to be the framework support for *schema evolution*. This is not a criterion that is either fulfilled or not, but may vary in several ways, from requiring both schema versions for deserialization, supporting only specific trivial type changes, to renaming, dropping or adding fields in different contexts.

Less important, but worth examining nonetheless, are the features regarding interoperability, including the *framework*, *platform* and *language* independence of an encoding.

The most important property of *speed improvement* with regard to (de)serialization requires careful preliminary considerations, careful model building and selection of data structures to allow generalizability, analysis of systematic effects and thorough measurement, consolidation and evaluation. The following section will be introducing the model for categorizing different protocols, which will also be applicable to determining a protocol's encoding size.

Subsequently, several categories are introduced and described, that will be used for comparing the qualitative features of a framework: *introduction effort and maintainability* as well as *documentation and learning curve*.

4.2.1 Performance Relative to Data Structure Complexity

According to the definition in Chapter 2 (*Serialization Overview and Classification*), a serialization protocol consists of three parts: a specific *initial data format*, *translation method* and resulting *encoding format* (see Figure 2.1). The speed of serializing, $Speed_{ser}$, and deserialization, $Speed_{deser}$, was defined in Chapter 2 (*Serialization Overview and Classification*), as being dependent on the concrete data structure instance, the serialization protocol, and the computing environment. (De)serializing performance is evaluated with regard to a certain protocol within a concrete computing environment, whose influence is aimed to be minimized. Theoretically, the only remaining variable in these equations is the data structure that is (de)serialized, which is consequently the deciding factor for performance (and equivalently encoding size) differences.

Problem 1 – Performance: Complex, Uncontrollable Computing Environment It is not feasibly possible to completely control the computing environment for evaluation. A varying influence will also always be present and unmanageable in any real-world scenarios, especially in the case of distributed systems. [2] came to the conclusion that the relative performance of an encoding protocol to another is independent of the used processor technology. Along with the load on a system it will therefore only be peripherally analyzed in this thesis, but assumed to have little bearing on different measurements on the same machine. It needs to be kept in mind and taken into account nonetheless as a possible source of inaccuracies when planning, measuring and evaluating.

Problem 2 – Data Structure Complexity: Comparing Protocols, not Just Examples It is possible to compare the performance of different serialization protocols for serializing a certain data structure, which is modeled equivalently according to their respective initial data formats. In order to compare the (de)serializing speed or encoding size of two serialization protocols *in general*, every possible data structure would need to be (de)serialized with each protocol and the results compared, which is obviously infeasible. 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.

There are different approaches to address this second problem. Commonly, an evaluation of serializers is conducted in a specific known context and use case. A common and practical strategy is to evaluate protocols tightly coupled to the requirements of this context, by selecting and focusing on common representatives of data structures therein. Another strategy concerns evaluating how different data types scale with growing content, especially composite and container types like *lists* or *maps*, and testing deeply nested structures.

Additionally, several attempts to define a general complexity metric will be made, which maps an object to a complexity value. The goal is that results of (de)serializing examples may be plotted as (discretely) connected graphs, that can be understood as a function of (de)serialization time/encoding size over data structure complexity. Their slopes could be evaluated and functions of different protocols compared. However, defining such a low-dimensional metric is problematic due to the different features a structure may have, which does not lend itself to such a simplification. Nonetheless, a number of ideas are explored and evaluated.

Complexity measure 1 – Based on time for serializing data structures with a certain protocol

One can try to evaluate all basic building blocks of the *initial data format* of a serialization protocol, by analyzing and collecting the time it takes to serialize each one and differently sized

contents, thereby creating a sort of *virtual construction kit*. From this kit, the time for serializing a more complex, composite data structure could be estimated by summing up the time it takes for serializing all the constituent parts. This would result in a function, that would map a data structure instance to a numerical time value, which can be regarded as its *complexity* with regard to its serialization time using this specific protocol. A few example objects could be created, creating a linear increase of needed time over complexity for this protocol, against which several other protocols could be evaluated as well. Ideally, the serialization protocol, against which the metric is established, should be the one we wish to replace.

Complexity measure 2 – Based on graph features and content A complexity metric for a data structure can be based on the mere structure of a data object, for example by evaluating the size for representing different building blocks present in most serialization protocols. These sizes can be taken from an imaginative language, the representation in a programming language, a small encoding format or a human- and machine-readable one like JSON. One may simply use the number of bytes necessary for encoding such a structure or assign values to different kinds of structures and adding them together with the size of their contents. One may also use existing metrics for the complexity of graphs or data or a combination thereof.

Definition 4.1. We define a complexity measure C_2 as function that maps a weighted graph to a scalar value in the following way:

$$C_2(G) = |E| + \sum G_{\text{weights}} \quad (4.1)$$

Where $G = G(V, E)$ is a weighted graph, $\sum G_{\text{weights}}$ the sum of all weights in the graph G and $|E|$ the total number of its edges.

The chosen complexity defined in Equation 4.1 assumes, that a concrete data structure is represented as a weighted, directed tree according to the descriptions in Section 2.1 (*Data Structures As Directed Graphs*). It merely adds together all the weights present in the graph taken from each field value's byte count, and the total number of edges, which correspond to the total number of fields. In this way, it takes into account both the overhead for encoding fields, which is therefore assumed to be the same for all data types, as well as the size of the content, which can have a large impact on the resultant encoding size.

Complexity measure 3 – Two-dimensional, based on graph features and content It can be speculated, that a graph's complexity with regard to (de)serialization depends on the overhead for each field and their cumulative respective contents. Therefore a two-dimensional complexity metric for data structures is introduced, where the first variable denotes the cumulative number of bytes stored in all fields, and the second one the number of fields contained in the entire structure. Two-dimensional plots can still be created by means of slicing the three-dimensional plot. The effect of each variable can be measured by only changing one in a sequence of measurements.

Definition 4.2. We define a complexity measure C_3 as function that maps a weighted graph to a tuple in the following way:

$$C_3(G) = (w, e) = (\sum G_{\text{weights}}, |E|) \quad (4.2)$$

Where $G = G(V, E)$ is a weighted graph, $\sum G_{\text{weights}}$ the sum of all weights in the graph G and $|E|$ the total number of its edges.

The chosen complexity defined in Equation 4.2 assumes, that a concrete data structure is represented as a weighted, directed tree according to the descriptions in section 2.1 (*Data Structures As Directed Graphs*). The metric is a tuple (w, e) with w being the sum of all the weights present in the graph, which are taken from each field value’s byte count, and e the total number of edges, which correspond to the total number of fields. In this way, it takes into account both the overhead for encoding fields, which is therefore assumed to be the same for all data types, as well as the size of the content, which can have a large impact on the resultant encoding size, but does so separately.

Complexity measures C_2 and C_3 are likely to only correspond to the encoding sizes of a serialization protocol, not the speed with which it (de)serializes, whereas C_1 is explicitly coupled to one of these speeds. Because all methods of encoding may include different features of a data structure, do so in distinct ways and differently optimized, these used parameters are not sufficient for describing encoding sizes or speeds. Internal parameters of each serialization protocol would need to be considered.

All these ideas are therefore also problematic with regard to the choice of selected examples that are compared, because there may be many structurally different ones that have the same complexity value. They might each lead to drastically different results when serialized by a specific, other protocol. One may try to alleviate this problem by picking several representatives per complexity value, showcasing them all for the other protocol or possibly averaging them. This would, however, eliminate the usefulness of such a metric in the first place. A more general way of representing these different results might be as a distribution curve that encompass the encoding sizes/(de)serialization speeds of all possible data structures for a certain complexity value, which may be calculated.

A suitable metric would, for a specific serialization protocol and any object with the same complexity score, result in a single value for the (de)serialization speed or encoding size, whichever it is defined against. This should hold for applying the metric to any serialization protocol. It is therefore easy to disprove the validity of a proposed metric by counterexample. Metrics C_1 , C_2 and C_3 are too simple to accurately describe (de)serialization speeds or sizes and will therefore not be further considered.

A useful but non-general complexity metric might be defined for the concrete use case that this study is conducted for. It might include typical features of the messages that are sent within *dCache* and weigh their significance based on the frequency of occurrence combined with the importance of the respective messages. In this thesis, three sets of data structure definitions were chosen for evaluating different aspects. The first one contains lists of different data types for evaluating how they scale for increasing numbers of values. The second one contains a number of selected composite types for different scenarios. The third one encompasses typical *dCache* specific features in an example message. Together they may be used to make statements about the general performance of a framework for the *dCache* system.

4.2.2 Qualitative Framework Features

Evaluating a framework’s documentation and learning curve as well as its introduction effort and maintainability, as defined and rated in Table 3.1, is in many ways a matter of subjective opinion. With regard to several aspects, however, a comparison can and will be 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. The resulting rating can serve to impose a relative order on evaluated serialization frameworks/protocols.

Category	Item Key	Ratable Item Description
Documentation: Organization	DL-01	Well organized, structured
	DL-02	Graphics/tables/lists/bullets/color
Documentation: Descriptions	DL-03	Simple and sufficient descriptions
Documentation: Content	DL-04	Explanation of general concepts
	DL-05	Schema language guide
	DL-06	Description of encodings
	DL-07	Usage guide
General concepts	DL-08	Intuitive to understand
Schema language	DL-09	Intuitive to understand
<i>Java</i> API	DL-10	Intuitive to understand
Introducing a framework	IM-01	Pre-compiled/ maven/ few dependencies
Writing a schema file	IM-02	Easy and lightweight syntax
	IM-03	Many types supported
Writing (de)serializers	IM-04	No need for code generation
	IM-05	Can use local language's types
	IM-06	Minimal implementation overhead
Handling schema changes	IM-07	Minimal effort
	IM-08	Forward and backward compatible
Maturity and support	IM-09	Long-standing, well maintained
	IM-10	Large community and support

Table 4.1: Criteria for qualitative evaluation of serialization frameworks

For this analysis, the frequently used *Likert Scale* [1] is used for defining and rating criteria, wherein quality is usually ranked on a five or seven level scale. The selected approach defines ten ratable items for the area of documentation and learning curve (item keys *DL*-*) and ten items for rating the introduction effort and maintainability (item keys *IM*-*), as can be seen in Table 4.1. Each item can be ranked on a scale from one to five, with one indicating strong disagreement with the described item and five denoting strong agreement or inapplicability of the item. In this way, a score less than five is only given when the criterion is applicable, resulting in a rating system, wherein points are being deducted from the maximum for the imperfect fulfillment of relevant criteria.

4.3 Evaluating Performance and Encoding Size

An increase in (de)serialization speed was evaluated to be the most important aspect in choosing to introduce a new serialization protocol, a reduction of the encoding size the least important (see Table 3.1). The latter will therefore primarily be evaluated in order to gain potentially more comprehensive insights into the reasons for different performance results.

While an encoding size can easily, reproducibly and accurately be identified, determining a time interval by measuring can merely be estimated and is inherently subject to inaccuracies and deviations. The complexity of a computing environment needs to be controlled as much as possible and a single measurement is not sufficient. Planning accordingly serves to reduce the fluctuations around the pure value, with the aim of minimizing them as much as possible. Once

a systematic approach to selecting data structure examples is created, that enables generalizing results, the respective (de)serialization speeds and resulting encoding sizes for different protocols can be measured.

4.3.1 Environment and Tools

In order to test different serialization protocols independently of the complexities of the *dCache* system, a simple, Java-based testing software was created. It was developed on a Linux *Fedora 30 (Workstation Edition)* system with the kernel version *5.1.20-300.fc30.x86_64*, using the *JRE 1.8.0_212-b04 amd64* and JVM *OpenJDK 64-Bit Server VM by Oracle Corporation* with the IDE *IntelliJ IDEA 2019.1.3 #UI-191.7479.19*.

Maven was used to include different libraries, most importantly the serialization frameworks *Protobuf* (`com.google.protobuf`, artifact `protobuf-java`, version 3.7.1), *Protostuff* (`io.protostuff`, artifacts `protostuff-core` and `protostuff-runtime`, both with version 1.5.9) *Avro* (`org.apache.avro`, artifact `avro`, version 1.9.0), *Kryo* (`com.esotericsoftware`, artifact `kryo`, version 5.0.0-RC4) and *FST* (`de.ruedigermoeller`, artifact `FST`, version 2.56). *JUnit* (version 4.12) was used for testing implemented serializers, *JFreeChart* [32] for plotting results (`org.jfree` version, artifact `jfreechart`, version 1.5.0). The framework *Hessian* was included as a jar file in the version 4.0.60.

Measuring the performance of a specific section of code is commonly referred to as *benchmarking*. Even if when disregarding of the computing environment outside of the used software, it is difficult to do this correctly, which is especially true for the *Java* programming language.

A naive benchmarking approach might entail measuring the time before and after the execution of the code to be tested and calculating the difference. This process would usually be iterated a large number of times to better capture fast operations, avoid outliers and rather describe the central tendency of the collected data set by calculating the arithmetic mean, percentiles or standard deviations. This approach, however, does not take into account several important aspects that might affect the measured duration. Even the called method calls have a possibly significant overhead that cannot be separated easily from the benchmarked code (like `System.currentTimeMillis()` for retrieving timestamps).

Most problems, however, originate from the JVM being an adaptive virtual machine that silently performs several kinds of optimizations. *OpenJDK* uses the *HotSpot* Virtual Machine, which at first executes a line of code in interpreted mode and only translates it into machine code after a certain time of execution, which makes a warmup period of several thousands of operations for benchmarking obligatory, as well as staying aware of deoptimization and recompilation effects. A list of optimizations performed by the Hotspot JVM can be seen in OpenJDK's wiki [38].

Some of these pitfalls were analyzed and discussed in [15], such as the bad practice of mixing similarly naive benchmarkers in the same JVM run. This will frequently lead to results that are dependent on the ordering of starting benchmarkers due to the JVM making and updating assumptions, that lead to (re)generating differently efficient byte code at runtime. Isolated runs are equally affected by different mechanisms. The JVM usually tries to guess the execution of the entire program run and attempts to optimize it by tricks such as loop unrolling, lock coalescing, constant folding, dead code elimination, method in-lining and on-stack replacements.

To conclude, writing a good *Java* benchmark is extremely difficult, because it requires a lot of knowledge of dynamic code generation, optimization and JVM implementation details.

Java Microbenchmark Harness (JMH) *JMH* [35] is a *Java* benchmarking harness, that is part of the *OpenJDK* project. It aims to help developers build, run and analyze benchmarks while avoiding being affected by the mentioned problems within the context of a *JVM*, which it does not, however, prevent.

JMH is especially designed to facilitate *microbenchmarking*, which targets a specific, small piece of code, but it can also be used for larger benchmarks. It was chosen for this project because it is a popular, well supported and integrated framework, that is easy to use and can be included via maven (`org.openjdk.jmh`, artifacts `jmh-core` and `jmh-generator-annprocess`, both of version 1.21).

JMH can be configured through annotations on benchmark classes or methods to operate in different ways. It supports running in several `BenchmarkModes`, two of which are `throughput` as the default option, which measures how many operations are executed per specified time interval, and `average time`, wherein the average time for executing a benchmark method is evaluated. Several `TimeUnits` are supported, ranging from `NANOSECONDS`, `MICROSECONDS`, to `MINUTES` or even `DAYS`. The number of `Warmup` and `Measurement` iterations, as well as the time per iteration (with respective time unit) can be specified in addition to the number of `Forks`, which determines how often the combination of `Warmup` and `Measurement` is repeated.

If a benchmarking code needs an external state, that should be initialized outside of it and provided as a parameter, this is possible by so-called `state` variables. They can be annotated to have a specific scope, such as existing once per `Thread`, `Thread-Group` or `Benchmark`, so that all threads share this object.

JMH generates a report that is most thorough when requested in *JSON* format. It contains information for each benchmarked method: the *JMH* version and configuration, the *Java* environment and the benchmark method with possible state parameter values. It also includes the results of the measurements, such as `score`, `scoreError`, `ScoreConfidence` as well as `scorePercentiles` and `rawData`, which contains an average value for each `measurementIteration`.

Unfortunately, bad practices in *JMH* benchmarking are nonetheless prevalent and can easily lead to measuring something different than intended [3]. In this comparison of serializers, care was taken to not make those frequent mistakes. Empirically reasonable values for `Fork`, `Warmup` and `Measurement` were chosen, with `Fork` not being zero. Loops are never used to repeat the measured code parts several times per called benchmarking method. The result of (de)serializing data is always returned from a benchmarking method, so that the compiler will not simply eliminate it as unused code, which could also be achieved by feeding the value into an instance of *JMH*'s `blackhole` class. Parameters for benchmarking methods did not include `final` primitives. *Fixture* methods for setting up/tearing down benchmarks or `state` objects were not run at the costly level of benchmark `Invocation`, but `Trial` (at the start of each `Fork` run).

Singularity Containers The created benchmarking software is run as a *jar* file on different *Linux* machines. Instead of creating a different *jar* per benchmark class, a command line interface was implemented for configuring the type of benchmarker that should be run, the path for the resultant *JSON* file, and optionally its name. In order to simplify starting and stopping a benchmarker on different systems, as well as more easily controlling and reproducing its environment such as the used *Java* version, *Singularity* containers were chosen to package and deploy the created benchmarking software. *Singularity* [36] is an open-source, cross-platform tool for enabling *containerization* as an operating-system-level virtualization. It is most commonly used in scientific computing within the *high-performance computing (HPC)* world. *Singularity* is advantageous compared to other containerization solutions, because it can run with user privileges and does not need to communicate with an intermediary `root`-owned daemon process.

Feature	dcache-dot1	dcache-dot13
Linux Kernel	3.10.0-862.14.4.el7.x86_64	3.10.0-957.21.3.el7.x86_64
HEP-SPEC06 v1.2	125.58	363.03
Processors	12 Intel(R) Xeon(R)	20 Intel(R) Xeon(R)
	CPU E5-2440 0 @ 2.40GHz	CPU E5-2660 v2 @ 2.20GHz
Memory in kB	49243160	131812036

Table 4.2: Comparison of test-machines

The created *Singularity* recipe file for building the benchmarker container image is shown in Listing 6.1.

To be able to easily and reproducibly spawn and start multiple container instances, a script was created that accepts a *Singularity* image and a configuration file with a line per benchmark. When started with superuser privileges, the script will create Linux `cgroup` files to lock each instance to a separate CPU, if enough are available. It is hoped, that in this way several benchmarks can reliably be run in parallel without influencing each other. *Multithreading* should be disabled in order to further minimize (virtual) CPU hopping.

Test Machines The benchmarking tests were conducted on two machines, *dcache-dot1* and *dcache-dot13*, which belong to the *dCache operations team (DOT)*. On *dcache-dot1*, other activities are conducted, such as running up to three *Spark* masters. This provides a more realistic but unreliable environment. Machine *dcache-dot13* was solely reserved for work regarding this thesis, multithreading was disabled.

Table 4.2 compares the most relevant features of both machines. They run the same *Linux* flavor, but a slightly different kernel version. Machine *dcache-dot1* has only 12 physical processors running at 2.4 GHz, *dcache-dot13* has 20 processors running at 2.2 GHz. The latter additionally has over two and a half times as much memory as the first. This results in a much higher *HEP-SPEC06 (HS06)* value overall, which was chosen to be the primary metric for measuring CPU performance as defined by the *HEPiX Benchmarking Working Group*. *HEPiX* is a collection of worldwide Information Technology staff working in the context of *High Energy Physics and Nuclear Physics (HEP)*. *HS06* is based on a subset of the industry standard *SPEC® CPU2006* benchmark suite that is most relevant for *HEP*.

Tests were conducted in three different environments:

- **dot1:** Machine *dcache-dot1*, user context
- **dot13:** Machine *dcache-dot13*, user context
- **dot13cpulock:** Machine *dcache-dot13*, superuser, each container locked to a separate CPU

4.3.2 Data Structures

As was discussed in Subsection 4.2.1 (*Performance Relative to Data Structure Complexity*), it is only possible to measure *examples*, which need to be wisely selected in order to be able to generalize and draw conclusions about an entire *serialization protocol*.

Different strategies were chosen to approach being able to do so. First of all, two general sets of examples were evaluated for each protocol. The first one, which we will refer to as the *typelist* set, includes three classes, each encapsulating a list of a different type. These class objects are filled

with differently sized lists of values for serialization. The second set is referred to as *composites*. It contains six comparably more complex classes that are each composed of different data types, including all the basic types and a few composite ones from the *Java* language. Finally, one of the most commonly sent *dCache* messages was chosen to represent a context-specific, third set of test objects.

Because *JOS* serializes more than just the data fields in an object, the shown listings of serializable classes either include the declared methods and constructors explicitly or they are mentioned in comments, like the presence of *getters* and *setters* for all fields.

The Typelist Data Set includes three list encapsulating classes, one for `int` (class `IntList`), one for `double` (class `DoubleList`) and one for `String` (class `StringList`) typed values. They all have an equivalent class structure, which is shown by example of the `IntList` class in Listing 4.1.

```

1 public class IntList implements Serializable {
2
3     private List<Integer> values;
4
5     public IntList(List<Integer> ints) {
6         this.values = ints;
7     }
8
9     // argumentless constructor for kryo/protostuff-runtime
10    public IntList() {
11    }
12
13    public List<Integer> getValues() {
14        return values;
15    }
16
17    public void setValues(List<Integer> values) {
18        this.values = values;
19    }
20 }
```

Listing 4.1: Category *Typelist*: class `IntList`

The empty explicit constructor is necessary for some of the full object graph serializers such as *protostuff-runtime* and *Kryo* to reconstruct an object instance. It is therefore dangerous to set default list or map values within a parameterless constructor, because the deserialized values might merely be added to the values set by the constructor. This side effect is observable in *protostuff-runtime*. When no such types are present in an object, the empty constructor was chosen to set default values in order to minimize the overhead for *JOS*.

Because the dynamic generation of values for each benchmarked *typelist* object run would introduce potentially undesired statistical variations between results, differently sized lists of values were pre-generated for each data type and stored in *CSV* files. They contain 10, 50, 100, 1000, 10000 and 100000 values. The `int` and `double` values were generated using `Random().ints(count)` and `Random().doubles(count)`, respectively. The format of `String` values was decided to be of fixed length to reduce fluctuations and just measure the overhead for encoding this data type. These values were generated using the *Java* `UUID` class for generating unique, alphanumeric identifiers containing minus signs. The function `UUID.randomUUID()` was used, resulting in outputs that are each 36 characters long, such as `c5f1e155-4812-49e3-9d78-c0b5c04cb184`.

The *Typelist* schema definitions for *Protobuf* are shown in Listing 6.2, the ones for *Avro* in Listing 6.3.

The Composites Data Set was constructed to include a number of small artificial composite data types of increasing estimated complexity. Primitive types, as well as references, lists and maps were tested in different combinations.

```

1 public class Msg implements Serializable {
2
3     private static final long serialVersionUID = -8421133630068493665L;
4
5     public static long getSerialVersionUID() {
6         return serialVersionUID;
7     }
8 }

```

Listing 4.2: Category *Composite*: class `Msg` (*C0*)

The class `Msg` (see Listing 4.2), referred to as *C0*, only contains a static `serialVersionUID` field of type `long`, that is commonly needed by *JOS* to reliably handle class changes. It was decided that the explicitly schema-based serializers *Protobuf* and *Avro* do not encode this field, because its value is bound to the class, not a specific instance, and these protocols do not require it for schema evolution. An instance of this class is very small in every case and mainly serves to test the overhead of a class or data structure container.

```

1 public class Msg_rw01 implements Serializable {
2
3     private char _char01;
4     private byte _byte01;
5     private boolean _boolean01;
6
7     private short _short01;
8     private int _int01;
9     private long _long01;
10
11     private float _float01;
12     private double _double01;
13
14     public Msg_rw01() {
15         _char01 = (char) 4;
16         _byte01 = (byte) 6;
17         _boolean01 = true;
18
19         _short01 = 124;
20         _int01 = 4657889;
21         _long01 = 56623889;
22
23         _float01 = 2465.5f;
24         _double01 = 323698.5;
25     }
26
27     // ...getters and setters ...
28 }

```

Listing 4.3: Category *Composite*: class `Msg_rw01` (*C1*)

The class `Msg_rw01` (*C1*) is displayed in Listing 4.3. It contains a field for every basic data type in the *Java* language. Its parameterless constructor assigns default values on instantiation, the class additionally contains a simple *getter* and *setter* per field. The explicitly schema-based serialization protocols *Avro* and *Protobuf* do not directly support all of these types, but there are several alternatives of representing one instead. When creating the respective schema, the next most efficient type is attempted to be chosen. For example, they both do not inherently support a `char` or a `byte` primitive type, which could instead be represented using a `bytes` array of one

element or the shortest available integral type, in these cases 32 bit `int`. In this example, the latter was chosen for both protocols and an `int` was also selected for the `short` typed field.

```

1 public class Msg_rw02 implements Serializable {
2
3     private Msg _msg;
4     private Msg_rw01 _msg_rw01;
5
6     public Msg_rw02() {
7         _msg = new Msg();
8         _msg_rw01 = new Msg_rw01();
9     }
10
11     // ...getters and setters ...
12 }

```

Listing 4.4: Category *Composite*: class `Msg_rw02` (*C2*)

In Listing 4.4, the `Msg_rw02` (*C2*) class is displayed. It contains two fields, one of type `Msg`, one of type `Msg_rw01`. It thereby not only includes all the fields from those classes, but additionally introduces a nesting layer and two other composite types, that need to be encoded.

```

1 public class Msg_rw03 implements Serializable {
2
3     private Msg _msg;
4     private Msg_rw01 _msg_rw01_00;
5     private Msg_rw01 _msg_rw01_01;
6
7     public Msg_rw03() {
8         _msg = new Msg();
9         _msg_rw01_00 = new Msg_rw01();
10        _msg_rw01_01 = _msg_rw01_00;
11    }
12
13    // ...getters and setters ...
14 }

```

Listing 4.5: Category *Composite*: class `Msg_rw03` (*C3*)

The `Msg_rw03` (*C3*) class, as displayed in Listing 4.5, has the same structure as `Msg_rw02`, but contains a second field of type `Msg_rw01`. It is instantiated by the parameterless constructor, which assigns the same reference to both fields. Many of the analyzed serializers are unable to encode graph structures and encode references as identical copies. This is speculated to lead to increased encoding sizes and potentially (de)serialization speeds.

```

1 public class Msg_rw04 implements Serializable {
2
3     private List<String> _stringList01;
4     private Map<Integer, String> _intStrMap01;
5
6     // argumentless constructor for kryo/protostuff-runtime
7     private Msg_rw04() {
8     }
9
10    public Msg_rw04(List<String> stringList, Map<Integer, String> intStringMap) {
11        this._stringList01 = stringList;
12        this._intStrMap01 = intStringMap;
13    }
14
15    public static Msg_rw04 createTestInstance() {
16        List<String> stringList;
17        stringList = new LinkedList<String>( Arrays.asList(new String[]{"We", "
coding", "monkeys", "really", "like", "gobbledygook", "!"}) );

```

```

18
19     Map<Integer, String> intStringMap = new HashMap<>();
20     intStringMap.put(3, "hello");
21     intStringMap.put(2, "Nothing really matters.");
22     intStringMap.put(6, "some non-random, but meaningless string");
23
24     Msg_rw04 msg = new Msg_rw04( stringList, intStringMap );
25     return msg;
26 }
27
28 // ...getters and setters ...
29 }

```

Listing 4.6: Category *Composite*: class `Msg_rw04` (*C4*)

The class `Msg_rw04` (*C4*) (see Listing 4.6) contains a list of type `String` and a map with `Integer` typed keys and `String` typed values. *Kryo* and *protostuff-runtime* require the existence of an empty constructor that does not fill these objects with default values. Instead, a static method was created, that uses the second constructor to create and return a test instance of the class. The list is filled with seven `Strings` ranging from length one to twelve, the map with three keys and their associated one, three and five-word `String` values.

```

1 public class Msg_rw05 implements Serializable {
2
3     private Byte _byte01;
4     private Boolean _boolean01;
5
6     private Short _short01;
7     private Integer _int01;
8     private Long _long01;
9
10    private Float _float01;
11    private Double _double01;
12
13    private String _string01;
14
15    public Msg_rw05() {
16        _byte01 = new Byte((byte) 6);
17        _boolean01 = new Boolean(false);
18
19        _short01 = new Short((short) 124);
20        _int01 = new Integer(4657889);
21        _long01 = new Long(56623889);
22
23        _float01 = new Float(2465.5f);
24        _double01 = new Double(323698.5);
25
26        _string01 = "Lorem ipsum dolor sit amet, consetetur sadipscing elitr, " +
27        "sed diam nonumy eirmod tempor invidunt ut labore et dolore magna " +
28        "aliquyam erat, sed diam voluptua. At vero eos et accusam et " +
29        "justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea " +
30        "takimata sanctus est Lorem ipsum dolor sit amet. Lorem ipsum dolor " +
31        "sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor " +
32        "invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. " +
33        "At vero eos et accusam et justo duo dolores et ea rebum. Stet clita " +
34        "kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.";
35    }
36
37    // ...getters and setters ...
38 }

```

Listing 4.7: Category *Composite*: class `Msg_rw05` (*C5*)

The `Msg_rw05 (C5)` class is shown in Listing 4.7. It is similar to `C1`, but instead of containing fields of the basic *Java* types, it makes use of the equivalent classes for all but the purely basic `char` type. Instead, another field of type `String` was added. A test instance of this class is again constructed via the parameterless constructor. The same values as for `C1` are assigned, except for the `Boolean` typed field, which is set to `false`, and the added `String` field, which is assigned a text containing 591 characters. The serializers, which encode object trees are assumed to encode these fields as objects, whereas the explicitly schema-based protocols merely use the containers.

The *Composites* schema definitions for *Protobuf* are shown in Listing 6.4, the ones for *Avro* in Listing 6.5.

The dCache-like Data Set Whereas the previous categories are of a more general nature, the *dCache-like* set was created to test the (de)serialization of messages that are very similar to the messages within the *dCache* system. Serialization protocols might behave slightly differently on these combinations of data structure features. The frequently sent `PoolManagerPoolUpMessage` was chosen to represent a typical example. Every pool in a *dCache* system sends it to the *PoolManager* every 30 seconds as its heartbeat, communicating information about itself and consequently that it is running. The number of these messages increases with the number of pools in a system. For example, both the *CMS* and the *XFEL dCache* instances at *DESY* contain over 800 pools at the time of writing this thesis.

The class `PoolManagerPoolUpMessage` extends `PoolManagerMessage`, which in turn extends `Message`. The field included in the `Message` class are shown in Listing 4.8.

```

1 public class Message implements Serializable {
2
3     private boolean _replyRequired;
4     private boolean _isReply;
5     private int     _returnCode;
6     private Object  _errorObject;
7     private long    _id;
8     private Subject _subject;
9
10    private static final long serialVersionUID = 2056896713066252504L;
11
12    // ...methods ...
13 }
```

Listing 4.8: Category *dCache-like*: class `Message`

The `Message` class contains fields that are unspecifically needed for a message, such as its identifier, subject, whether it is a reply or requires one. It includes only primitive fields apart from the *Java* basic class `Object` and `Subject` from the package `javax.security.auth`.

```

1 public class PoolManagerPoolUpMessage extends PoolManagerMessage {
2     private final String _poolName ;
3     private final long   _serialId ;
4     private final PoolCostInfo _poolCostInfo ;
5     private final PoolV2Mode _mode;
6
7     private Map<String, String> _tagMap;
8     private Set<String> _hsmInstances;
9     private String _message = "";
10    private int _code;
11
12    private static final long serialVersionUID = -8421133630068493665L;
13 }
```

```

14 // ...methods ...
15 }

```

Listing 4.9: Category *dCache-like*: class `PoolManagerPoolUpMessage`

The class `PoolManagerMessage` only includes a static *serialVersionUID* field and two methods. Within the `PoolManagerPoolUpMessage` itself (see Listing 4.9), there are several fields, such as for the *pool* name, its serial identifier, a map of `String` tags and a list of associated *Hierarchical storage management (HSM)* instance `Strings` such as tape backends. The mode that the pool is in is of type `PoolV2Mode`, which can be seen in Listing 4.10.

```

1 public class PoolV2Mode implements Serializable {
2
3     private static final long serialVersionUID = -3620447515380724292L;
4
5     public static final int ENABLED           = 0x00 ;
6     public static final int DISABLED          = 0x01 ;
7     public static final int DISABLED_FETCH    = 0x02 ;
8     public static final int DISABLED_STORE    = 0x04 ;
9     public static final int DISABLED_STAGE    = 0x08 ;
10    // ... more static modes!
11
12    private int _mode = ENABLED ;
13
14    // ...methods ...
15 }

```

Listing 4.10: Category *dCache-like*: class `PoolV2Mode`

The `PoolV2Mode` merely stores an `int` typed value representing one of several, statically declared mode types. It additionally contains several methods for accessing, setting and printing the current mode. Its function is similar to that of an `enum`.

```

1 public class PoolCostInfo implements Serializable {
2
3     private static final long serialVersionUID = 5181562551679185500L;
4
5     private PoolQueueInfo _store;
6     private PoolQueueInfo _restore;
7     private PoolQueueInfo _p2p;
8     private PoolQueueInfo _p2pClient;
9     private final Map<String, NamedPoolQueueInfo> _extendedMoverHash;
10    private final String _defaultQueueName;
11    private PoolSpaceInfo _space;
12    private final String _poolName;
13    private double _moverCostFactor;
14
15    // ----- Own addition of default queues within class
16    public static final String DEFAULT_QUEUE = "regular";
17    public static final String P2P_QUEUE_NAME = "p2p";
18
19    // ...methods ...
20
21    public static class NamedPoolQueueInfo extends PoolQueueInfo {
22
23        private static final long serialVersionUID = -7097362707394583875L;
24        private final String _name;
25
26        // ...methods ...
27    }
28
29    public static class PoolQueueInfo implements Serializable {

```

```

30
31     private static final long serialVersionUID = 1304697767284208011L;
32
33     private int _active;
34     private final int _maxActive;
35     private int _queued;
36     private final int _readers;
37     private final int _writers;
38
39     // ...methods ...
40 }
41
42 public static class PoolSpaceInfo implements Serializable {
43
44     private static final long serialVersionUID = -8966065301943351970L;
45
46     private final long _total;
47     private long _free;
48     private final long _precious;
49     private final long _removable;
50     private final long _lru;
51     private long _gap;
52     private double _breakEven;
53
54     // ...methods ...
55 }
56 }

```

Listing 4.11: Category *dCache-like*: class `PoolCostInfo`

The class `PoolCostInfo` (see Listing 4.11) stores usage and configuration information on a *pool*. It contains three static inner classes. The one called `PoolSpaceInfo` is used for storing information on different types of space usage. The ones named `PoolQueueInfo` and `NamedPoolQueueInfo` store the current queue for different operations on the pool, such as the number of active and queued requests as well as the number of readers and writers. The `PoolCostInfo` class contains fields of type `PoolQueueInfo` for different types of requests, such as for storing, restoring from tape, pool to pool operations and pool to client operations. It has fields for instances of these classes and several methods for generating and returning them.

Because the class `PoolManagerPoolUpMessage` is more complicated than examples from the general categories, it is not as easy to create a schema for those serialization protocols which require one. Within the `Message` class, the `_errObject` field of type `Object` is frequently simply assigned a `String` and could be represented as such in an explicitly declared schema. However, there is a field of type `Subject`, which is a *Java* class that includes several fields of difficult types to explicitly represent in a schema. There are sets of credentials of type `Object` and one of type `Principal`, which is an interface with over 50 direct and many more subclassed implementations. Because of the complexity and following dependence on changes to the *Java* standard library, it was decided to exclude explicitly schema-based protocols for evaluating the `PoolManagerPoolUpMessage` (*PMPUM*) class. Consequently, these protocols are infeasible for being integrated into the *dCache* software.

4.3.3 Benchmarking Setup

Different sets of *jmh* benchmarking methods were grouped in benchmarker classes and evaluated together. These include two separate classes to evaluate each benchmarking protocol, one for the *composite* objects, one for the *typelist* ones. An additional single class encapsulates benchmarking

the (de)serialization speed of the *dCache* message `PoolManagerPoolUpMessage` (*PMPUM*) for all protocols.

All benchmarkers were run in the *jmh* mode `AverageTime`. The warmup period was specified to include four iterations of ten seconds each, the measurement ten iterations of 100 seconds. This was specified to run two times via a `Fork` value of two. Fully benchmarking a method therefore takes 2080 seconds or 34,6 minutes. The *Composites* class contains twelve methods for serializing and deserializing each of the six object types. One benchmarker run therefore takes six hours and 56 minutes. The *Listtypes* benchmarker classes contain twelve methods for each of the three list types, resulting in a runtime of 20 hours and 48 minutes. The class for benchmarking the *PMPUM* for five suitable protocols consequently includes ten methods and runs five hours, 46 minutes and 40 seconds.

Each of these benchmarkers was run five times and the results were averaged. This was done using each one of the three machine configurations to be able to compare their influence on the results. Using the *Singularity* container deployment method, several benchmarkers were run in parallel. These groups differ for the two machines because of the different number of physical cores available. On *dcache-dot1*, a group consisted of either the eight *Composites*, the eight *Listtypes* or five *PMPUM* benchmarkers. On *dcache-dot13*, the *Composites* and *Listtypes* were run together as a group of sixteen. The most controlled environment, which is consequently supposed to be the most reliable, is the *dot13cpulock*, which binds each benchmarker to a separate CPU. This can be assessed by comparing the results to those acquired without doing so. Machine *dot1* may differ in its results due to both its different hardware, the additional load generated by unrelated processes and the different numbers or types of parallel benchmarkers. It is assumed that this machine generally produces slower results, which are additionally less consistent and may fluctuate. It is possible, however, that the higher clock rate of 2,4 instead of 2,2 GHz may be of significant relevance and lead to faster results than on *dcache-dot13*.

4.3.4 Evaluating Individual Serialization Protocols

Except for the machine configuration comparison, all the shown and described results were generated on the machine environment *dot13cpulock*, which was the most controlled setting. They include all serialization protocols, both the schema-based ones and the full object graph serializers. The individual protocols are only evaluated with regard to (de)serialization speeds of the general categories *typelist* and *composite*.

Java Object Serialization Evaluating *JOS* as the protocol to be replaced is important in order to be able to compare other serializers with the current, insufficient implementation. In Figure 4.1a, the averaged serialization speed for each of the list types is shown. They are seemingly identical. Compared to the deserialization speed shown in Figure 4.1b, the speed is approximately proportional to the number of values in a list. Deserialization takes significantly longer than serialization for a small number of values but starts being similar starting from 1000 valued lists and ends up being faster for 10^5 values. `DoubleList` and `intList` are indistinguishable. Compared to the `StringList` graph, which is almost linear, fewer values take even more time to deserialize, whereas larger ones are faster.

Figure 4.2a shows the combined serialization and deserialization speed for *composite* objects. It is apparent that deserialization takes significantly longer than serialization. It is observable that objects *C2* and *C3* are almost equally fast, with a slight increase for deserialization. This indicates that *JOS* is indeed capable of encoding references and does not need to store the same object a second time for the additional field in *C3*. Since *C2* contains *C0* and *C1*, it is interesting

Figure 4.1: *JOS* (de)serialization speed for *typelist* objects. The statistical uncertainties on the averaged durations are found to be negligible.

to observe that *C2* takes roughly their combined times for serialization, but a significant amount longer for deserialization.

Kryo deserialization of *typelist* objects can be seen in 4.3a, deserialization in Figure 4.3b. Decoding is faster in all cases. The **StringList** type takes the longest, the other two are similar in both modes. Encoding is slower for small objects but approaches a linear increase with growing value count.

In Figure 4.2b, where the (de)serialization speeds for *composite* types are shown, the dominance of the serialization speed can be clearly observed. Objects *C0* to *C3* have similar encoding speeds and primarily vary with regard to an increase in decoding speed. *C4* takes significantly longer than the previous ones and *C5* takes again much longer. This indicates that lists and **string** typed values take longer to encode than primitive typed fields or even own composite type definition instances.

Fast-serialization for *typelist* objects as displayed in Figures 4.4a and 4.4b is roughly linear, but fluctuates especially for **StringList** types. They are the ones that take the longest, with **IntList** objects being the fastest slightly below **DoubleList** ones. Encoding takes longer than decoding for small objects, but ends up being equivalent for large list sizes.

Figure 4.5a visualizes the results for encoding *composite* objects with *FST*. The small difference between *C2* to *C3* as opposed to *C1* to *C2* shows that the protocol does not require a large overhead for encoding the additional reference, because it possibly does not represent it as a copy. Interestingly, *C4* and *C5* take significantly longer to (de)serialize than all the other objects, which indicates that *FST* does not efficiently encode lists/maps and especially **string** typed values.

(a) *JOS* (de)serialization speed for *composite* objects (b) *Kryo* (de)serialization speed for *composite* objects

Figure 4.2: *JOS* and *Kryo* (de)serialization speed for *composite* objects. The statistical uncertainties on the averaged durations are found to be negligible.

(a) Serialization speed

(b) Deserialization speed

Figure 4.3: *Kryo* (de)serialization speed for *typelist* objects. The statistical uncertainties on the averaged durations are found to be negligible.

(a) Serialization speed

(b) Deserialization speed

Figure 4.4: *FST* (de)serialization speed for *typelist* objects. The statistical uncertainties on the averaged durations are found to be negligible.

(a) *FST* (de)serialization speed for *composite* objects

(b) *Hessian* (de)serialization speed for *composite* objects

Figure 4.5: *FST* and *Hessian* (de)serialization speed for *composite* objects. The statistical uncertainties on the averaged durations are found to be negligible.

Hessian (de)serialization speeds are shown in Figures 4.6a and 4.6b. The types `IntList` and `DoubleList` are equivalently fast for both encoding and decoding, while type `stringList` is significantly slower. Again, serialization has an overhead for small lists but approaches a linear increase with larger values, which is the case for all sizes regarding deserialization.

Figures 4.6a and 4.6b show the (de)serialization speed for *composite* objects. Interestingly, the encoding speeds for *C1* to *C3* is almost the same and only slightly larger for *C4* and again slightly for *C5*. There is a small increase in decoding speed, however. The small difference between *C2* and *C3* indicates that storing the additional reference does not pose much of an overhead for *Hessian*. Representing lists and maps does not seem to impose much additional time cost, but large `string` values lead to relatively large decoding speeds as apparent for *C5*.

Figure 4.6: *Hessian* (de)serialization speed for *typelist* objects. The statistical uncertainties on the averaged durations are found to be negligible.

Protobuf is explicitly schema-based and as such allows making adjustments and optimizations by choosing how certain fields should be represented. In Figure 4.7a, the speed for serialization can be observed to be significantly different for each type, but is roughly linearly increasing for each one. Type `DoubleList` is fastest, `StringList` objects take the longest time. Objects of type `IntList` take slightly more time than those of type `DoubleList`, but both are very similar with regard to deserialization as can be seen in 4.7a. Both `StringList` and `DoubleList` seem to take a fixed amount of time longer for deserialization than serialization.

Figure 4.8a shows the (de)serialization speeds for the *composite* objects. As might be suspected, it seems like *C2*, which is composed of *C0* and *C1*, is the combination of both. However, from *C2* to *C3* there is a significant difference, which might result from the fact that the protocol serializes the reference as a copy. Finally, the large `string` included in *C5* in addition to the values from *C1* does not take as much time as encoding a list and a map of significantly fewer values in *C4*.

Protostuff-runtime is based on *Protobuf*, but uses reflection to dynamically construct a schema for a given object, which does not allow equal control to choose the used data types. Figures

(a) Serialization speed

(b) Deserialization speed

Figure 4.7: *Protobuf* (de)serialization speed for *typelist* objects. The statistical uncertainties on the averaged durations are found to be negligible.

(a) *Protobuf* (de)serialization speed for *composite* objects

(b) *Protostuff-runtime* (de)serialization speed for *composite* objects

Figure 4.8: *Protobuf* and *Protostuff-runtime* (de)serialization speed for *composite* objects. The statistical uncertainties on the averaged durations are found to be negligible.

4.9a and 4.9b show that (de)serialization speeds are nearly proportional to the number of values in a list. The deserialization speeds are only marginally slower. Like in the case of *Protobuf*, **StringList** objects are the slowest, while **DoubleList** objects are the fastest. **IntList** ones slightly slower.

Figure 4.8b shows the *Protostuff* (de)serialization speeds for the *composite* types. It is noticeable that in the majority of cases the encoding takes longer than decoding. The difference between *C2* and *C3* indicates *Protostuff* having to do significantly more work for encoding the additional reference or copy.

Figure 4.9: *Protostuff*-runtime (de)serialization speed for *typelist* objects. The statistical uncertainties on the averaged durations are found to be negligible.

Avro shows a large difference in serialization speed for *StringList* objects as opposed to **DoubleList** or **IntList** objects (see Figure 4.10a and 4.10b). The latter seems almost identical for both serialization and deserialization, whereas **StringList** objects take significantly longer, especially to encode. Smaller objects are generally slightly slower to (de)serialize, but with increasing value count the speed becomes proportional to them.

Figure 4.11a shows the (de)serialization speed for the *composite* objects. Even though *C2* encompasses *C0* and *C1*, its encoding speed is almost equal to that of *C1* and smaller than the combination of those of *C0* and *C1*, which is the case for the time it takes for decoding *C2*. The significant increase between *C2* and *C3* indicates that the additional reference/copy is not trivial to add. Interestingly, serializing *C4* and *C5* takes almost as much time, but decoding the list and map in *C4* takes significantly longer than the basic typed fields and the large **string** field in *C5*.

Avro-Json shows an interesting (de)serialization profile for the *typelist* objects (see Figure 4.12a). The three types take distinct times for encoding/decoding, with **DoubleList** objects being the slowest, followed by **StringList** and finally **IntList** ones. Serialization and deserialization takes roughly the same time. Smaller objects have a time overhead so that they take longer, but eventually increase linearly.

(a) Serialization speed

(b) Deserialization speed

Figure 4.10: Avro (de)serialization speed for *typelist* objects. The statistical uncertainties on the averaged durations are found to be negligible.

(a) Avro (de)serialization speed for *composite* objects

(b) Avro-Json (de)serialization speed for *composite* objects

Figure 4.11: Avro and Avro-Json (de)serialization speed for *composite* objects. The statistical uncertainties on the averaged durations are found to be negligible.

Figure 4.11b shows the (de)serialization speeds for *composite* objects. Except for *C0*, encoding and decoding takes almost the same amount of time. Possibly because of this, there is a large observable difference between *C0* and *C1*, but not between *C1* and *C2*. Interestingly, (de)serializing *C4* is faster than *C3* and even *C2*, which indicates that lists and maps are easier to encode than composite, self-defines field types.

Figure 4.12: *AvroJson* (de)serialization speed for *typelist* objects. The statistical uncertainties on the averaged durations are found to be negligible.

4.3.5 Comparison of Serialization Protocols

The evaluated serialization protocols will be compared with regard to (de)serialization speed of the *typelist* and *composite* objects as well as the *PoolManagerPoolUpMessage*. Additionally, the encoding sizes for the *typelist* objects and the *PMPUM* will be shown and contrasted.

The (de)serialization speed for *typelists* will be shown as a ratio of the speed of a serialization protocol divided by the speed of the reference protocol *JOS* (see Equation 4.3).

$$R^{\text{Prot}} = \frac{\text{Speed}_{\text{Ser}}(\text{Protocol})}{\text{Speed}_{\text{Ser}}(\text{JOS})} \quad (4.3)$$

This approach conserves the relation of serialization protocol speeds among each other, while additionally showing the relation to *JOS*, which is sought to be evaluated.

Figure 4.13a shows the relation of serialization protocol speeds on *IntList* objects. *JOS* is a vertical line at one. For the smallest list size, *AvroJson* takes almost twice as much time as *JOS* and *Hessian* equally as much. *Avro* and *Kryo* are about a quarter as fast for this size and thereafter join the others at taking roughly an eighths down to a tenth or less of *JOS*. *Protobuf* and *Avro* are the fastest. The comparison of the deserializing speeds of *IntList* objects is shown in Figure 4.13b. In this graph not even *AvroJson* at any point takes more than three quarters of

Figure 4.13: Ratio comparison of *IntList* (de)serialization speeds. The statistical uncertainties on the averaged durations are found to be negligible.

the time *JOS* needs. Most protocols take about an eighth to tenths of *JOS*, but more than they do for serialization. The fastest protocol for deserialization of *IntList* objects is *Kryo*, followed by *FST* and *Protobuf/Protostuff-runtime*.

Figure 4.14a shows the R^{Prot} comparison for serializing *DoubleList* objects. *AvroJson* takes longer than *JOS* for all list sizes, but the ratio is decreasing from taking 2.75 times as much time for the smallest list type to 1.3 times as much. *Kryo* once more starts with the same speed as *JOS* but eventually drops to a quarter. The other protocols take less than twenty percent of the *JOS* time with the ratio decreasing with an increase in list size. The fastest protocols for serializing *DoubleList* objects are *Protobuf*, *Avro* and *Protostuff-runtime*. Figure 4.14b shows the deserialization speed ratio comparison. Here, *AvroJson* is rising from taking half as much time to almost two and a half times as much as *JOS* with increasing list size. The other protocols remain stable at eight to ten percent. The fastest protocols for deserializing *DoubleList* objects are *Protobuf*, *Kryo* and *Protostuff-runtime*.

Finally, the ratio comparison of the encoding speeds for *StringList* objects is shown in Figure 4.15a. This graph shows large diversity in the speed efficiency of different encoding protocols. Both *Kryo* and *AvroJson* take longer than *JOS* for small list sizes and eventually drop down to taking half as much. *Avro* takes sixty percent and eventually forty percent as much time. *Kryo* goes from taking 50 down to 25 and back up to fifty percent of the time needed by *JOS*. The only protocol that increases the time needed relative to *JOS* is *FST*, which starts from needing 25 percent and ending up at needing 90. Finally, the fastest protocol for encoding *StringList* objects is *Protobuf*, followed by *Protostuff-runtime*, which consistently need about ten to twenty percent of time compared to *JOS*. In Figure 4.15b they are joined in the ratio range of ten to eventually 25 percent by *Kryo* and *Avro* as the fastest protocols for deserialization of *StringList* objects. *AvroJson* and *Hessian* are much slower and are additionally decreasing their ratio with increasing list element count and stagnating at about 75 percent, respectively.

The (de)serialization speed of object *C0* can be compared in Figure 4.16a. *AvroJson* and *Hessian* take the longest amount of time for serialization, *JOS* is slowest with regard to deserialization,

(a) DoubleList serialization speeds

(b) DoubleList deserialization speeds

Figure 4.14: Ratio comparison of DoubleList (de)serialization speeds. The statistical uncertainties on the averaged durations are found to be negligible.

(a) StringList serialization speeds

(b) StringList deserialization speeds

Figure 4.15: Comparison of StringList (de)serialization speeds. The statistical uncertainties on the averaged durations are found to be negligible.

Figure 4.16: Comparison of (de)serialization speeds of $C0$ and $C1$. The statistical uncertainties on the averaged durations are found to be negligible.

followed by *Hessian*. *Protobuf* is the fastest protocol, closely followed by *Protostuff-runtime*, then *FST*, which takes significantly longer for deserialization, however. *Kryo* is as fast as *Protobuf* for deserialization. These protocols are similarly the fastest for object $C1$, which can be seen in 4.16b. Here, *AvroJson* is still the slowest protocol for serialization but also deserialization, followed by *Hessian* for serialization and *JOS* for deserialization.

In Figures 4.17a and 4.17b, comparing the (de)serialization speeds of $C2$ and $C3$, the fastest protocols remain the same with *FST* catching up to *Protostuff-runtime* for $C3$. In both graphs *AvroJson* is the slowest for serialization with *Hessian* on the second place and *JOS* is the slowest for deserialization, followed by *AvroJson*. *Hessian* remains on the second lowest place for serialization speed and third slowest for deserialization and second lowest.

In Figures 4.18a and 4.18b the general deserialization speed increases further. *JOS* remains the slowest for deserialization, followed by *AvroJson*, then *Hessian*, which is the slowest for serialization, followed by *JOS* and *Hessian*. The fastest protocols for encoding are still *Protobuf*, *Protostuff-runtime* and *FST*, with *Kryo* being on thirs place for decoding except in the case of $C5$.

Finally, the (de)serialization speeds for encoding an object of the *dCache-like PoolManager-PoolUpMessage* is compared in Figure 4.19 for serialization protocols that can infer an object graph schema at runtime. These include all protocols except for *Protobuf*, *Avro* and *AvroJson*. The slowest protocol for both encoding and decoding is *JOS*, which is the one to be replaced. *Hessian* in the next slowest in both categories taking about three quarters the serialization time and a quarter of the deserialization time of *JOS*. The fastest protocol for both serialization and deserialization is *Protostuff-runtime*, followed by *FST* regarding serialization speed and *Kryo* regarding deserialization speed.

(a) C2 (de)serialization speeds

(b) C3 (de)serialization speeds

Figure 4.17: Comparison of (de)serialization speeds of C2 and C3. The statistical uncertainties on the averaged durations are found to be negligible.

(a) C4 (de)serialization speeds

(b) C5 (de)serialization speeds

Figure 4.18: Comparison of (de)serialization speeds of Composites C4 and C5. The statistical uncertainties on the averaged durations are found to be negligible.

Figure 4.19: Comparison of *PMPUM* (de)serialization speeds. The statistical uncertainties on the averaged durations are found to be negligible.

The encoding size of different protocols may explain differences in (de)serialization speeds. They are compared regarding the *typelist* `IntList` in Figure 4.20a. Except for larger variations in the size of the encoding of a ten element list, they all increase lineally along with a larger number of values. *JOS* and *AvroJson* use the largest amount of bytes, which corresponds to their last places with regard to (de)serialization speed. *Avro* and *Kryo* provide the most compact encodings. For `DoubleList` objects as shown in Figure 4.20b, the largest encoding is that by *AvroJson*, the most compact are the ones by *Avro*, *Protobuf*, *Protostuff-runtime* and *Kryo*.

Finally, Figure 4.21a shows that `StringList` objects all result in very similar encoding sizes for each protocol. When serializing the *dCache-like* `PoolManagerPoolUpMessage`, the least compact encoding size is achieved by *JOS*, followed by *Hessian* (see Figure 4.21b). The most compact encoding is generated by *Kryo*, followed by the similar *Protostuff-runtime* and *FST*.

4.3.6 Evaluating the Influence of the Computing Environment

The results so far were generated on machine *dcache-dot13* in the *dot13cpulock* configuration. The benchmarkers were also run on the same machine without locking them to a separate CPU each, potentially resulting in CPU-hopping. This is speculated to lead to slower and less stable benchmarking results. Machine *dcache-dot1* was used without CPU-locking, different hardware specifications and potentially additional load by other users. It is assumed that the results are slower than both of the others, even more unstable but when averaged over five runs on different days a predictable amount of time slower.

The compared results for the example protocol *Protostuff-runtime* on the *composite* objects for different machines is shown in Figure 4.22. The difference between configuration *dot13cpulock* (*dot13c*) and *dot13* is marginal with *dot13* being slightly slower. Machine *dot1*, however, is significantly slower for both serialization and deserialization.

(a) Encoding sizes of `IntList`

(b) Encoding sizes of `DoubleList`

Figure 4.20: Comparison of encoding sizes of `IntList` and `DoubleList` objects

(a) Encoding sizes of `StringList`

(b) Encoding sizes of `PMPUM`

Figure 4.21: Comparison of encoding sizes of `StringList` and `PMPUM` objects

Figure 4.22: Comparison of *Protostuff* (de)serialization speed on different machines. The statistical uncertainties on the averaged durations are found to be negligible.

4.4 Evaluating Schema Evolution Support and Qualitative Features

According to the dCache developer ratings from Table 3.1, the second most important criterion for choosing a serialization protocol, after speed improvements, is its ability to support schema evolution (property C). This is followed by the introduction effort and maintainability of a framework (property D), then its associated documentation and learning curve (property D). These criteria will in the following be analyzed for the chosen frameworks and protocols.

Schema evolution support is summarized in Table 4.3. It includes *JOS*, which is based on the value of a unique `serialVersionUID` field per serializable class, that is advised to be explicitly set and maintained for correct serialization. *Java* internally associates a value with a class version, which can be used to identify the structure of a serialized object and deserialize it accordingly.

The protocols included in *Protobuf* and *Avro* support adding, removing and changing fields according to certain type conversion rules. These include changes from `int` to `long`, `float` to `double` and `string` to `bytes`. They are both forward and backward compatible. *Avro* additionally requires that both schemas are present for deserialization; the one used for encoding and the current one that will be used for decoding. *Protostuff-runtime* on unchanged *Java* classes only allows appending new fields for forward and backward schema evolution, but can make use of the same functionality by means of tags on class fields. These can include manually setting a numeric identifier to be used in the schema or declaring a field as deprecated, so that it will no longer be serialized.

FST is not forward compatible. It can only decode data serialized with the same or older versions of the serializer, not newer ones. This requires manually adding version annotations to fields and incrementing them when a change occurs.

Kryo allows different levels of schema evolution in an object by means of different (de)serializers. The `VersionFieldSerializer` allows backward compatibility for adding fields. A field must be tagged with `@Since(int)` to indicate the version when it was introduced. When using the `TaggedFieldSerializer`, tagged fields are supported to be added, renamed or removed. Changing its type is not possible. These changes are then backward and forward compatible.

Criterion	Rating	Requirements	Backward Compatible	Forward Compatible
<i>JOS</i>	4	New unique <code>serialVersionUID</code> field value per Version	Most secure with explicit, private, static, final <code>serialVersionUID</code>	Like backward
<i>Protobuf</i>	5	Never reassign field IDs	Adding and removing, certain simple type conversions	Like backward
<i>Protostuff-runtime</i>	4	Class field annotations for ID and field removing	Adding and removing, certain simple type conversions, without tags: append-only	Like backward
<i>Avro</i>	4	Old and new schema present when deserializing	Adding and removing, certain simple type conversions	Like backward
<i>FST</i>	2	By adding version annotations to fields	Yes	No
<i>Kryo</i>	4	Using one of two dedicated Serializers, adding version annotations	Either append-only or adding, removing, renaming	Like backward (not append-only)
<i>Hessian</i>	1	–	No	No

Table 4.3: Schema evolution support of serialization frameworks

Finally, *Hessian* does not seem to support schema evolution at all, requiring the user to implement that functionality when required.

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.

Qualitative framework features as defined in Table 4.1 include the introduction effort and maintainability of a framework, as well as its associated documentation and learning curve. The results of the evaluation according to the *Likert*-scale are shown in Table 4.4.

Regarding documentation and learning curve (item keys DL*), the clear leaders are *Kryo* (48 points) and *Protostuff-runtime* (48 points), closely followed by *Protobuf* (47 points). They have great documentation, their underlying concepts and provided API are easily understandable. *Avro* (44 points) and *FST* (43 points) are slightly less well documented, with *Hessian* (39 points) even scoring considerably lower because of this.

Introduction effort and maintainability (item keys IM*) is also clearly lead by *Kryo* (47 points), closely followed by *Protostuff-runtime* (44 points) and *Avro* (44 points). They have good language integration, do not require code generation and require minimal effort for writing (de)serializers and are well maintained. *Protobuf* (38 points), compared to *Avro*, has the disadvantage of needing to generate code to write or parse encoded data. This is not the case for *FST* (38 points) and *Hessian* (37 points), which are not schema-based and therefore require less effort for implementing (de)serializers, but they are both not well maintained and do not have external support through an active community. They also both do not support adequate schema evolution, which would then need to be handled by the users of both frameworks, who might need to update the serialized objects in the course of maintaining their software.

Item Key	Protobuf	Protostuff -runtime	Avro	FST	Kryo	Hessian
DL-01	5	5	5	4	5	2
DL-02	5	5	3	4	5	3
DL-03	5	4	4	4	5	3
DL-04	5	4	5	4	5	3
DL-05	5	5*	5	5*	5*	5*
DL-06	5	5*	5	4	4	5
DL-07	5	5*	5	5	5	3
DL-08	4	5	4	4	4	5
DL-09	4	5*	4	5*	5*	5*
DL-10	4	5	4	4	5	5
IM-01	5	5	5	5	5	5
IM-02	3	5*	5	5*	5*	5*
IM-03	4	5*	5	5*	5*	5*
IM-04	1	5	5	5	5	5
IM-05	4	5	4	5	5	5
IM-06	2	4	2	5	5	5
IM-07	4	3	3	3	3	1
IM-08	5	5	5	2	5	1
IM-09	5	3	5	1	4	3
IM-10	5	4	5	2	5	2
DL* Sum:	47	48	44	43	48	39
IM* Sum:	38	44	44	38	47	37
Total:	85	92	88	81	95	76

Table 4.4: Qualitative evaluation of serialization frameworks (5*: item not applicable)

Overall, *Kryo* (95) has earned the most points, closely followed by *Protostuff* (92 points) on the second place. *Avro* follows with 88 points, *Protobuf* with 85 points and *FST* with 81 points. With 76 points, *Hessian* has clearly landed on the last place.

5 Discussion

The selected serialization protocols were analyzed with regard to different criteria. In order to make a recommendation regarding the protocol to be introduced into the *dCache* system as a replacement for *JOS*, these categories need to be evaluated and their results combined.

5.1 Interpretation and Evaluation of Results

Several frameworks were excluded in the course of analysis due to shortcomings in essential aspects. *Protobuf* and *Avro* along with *Avro.Json* were infeasible to use for encoding the *dCache-like* class `PoolManagerPoolUpMessage`, because it would be necessary to manually create a schema for large *Java* library class trees (`Subject` class field), which cannot be represented with significantly simpler structures based on their usage. This would both require a lot of work for translating the over 150 different messages into this new format, many of which including similarly complicated type references, and may also introduce dependencies on internal changes to the *Java* class implementations. This is unfortunate, because *Protobuf* and *Avro* were the fastest and most compact protocols with good schema evolution support in both directions. They also both have good documentation, a reasonable learning curve and a large community. *Hessian* does not seem to support schema evolution, which would need to be implemented oneself, and is therefore similarly unsuitable. Although it is platform independent, it was often significantly slower than most other protocols, does not have good documentation, support or significant community.

The remaining protocols include *FST*, *Kryo* and *Protostuff-runtime*. For both serialization and deserialization of *Typelist* objects, *Kryo* is the fastest, except for serializing `StringList` objects, where *Protostuff-runtime* takes less time. It is in turn consistently the second fastest, except in the case of serializing `StringList` objects, where it is outperformed by *FST*. Encoding the *composites* objects is fastest using *Protostuff-runtime*, except for *C3*, where *FST* is quickest. Although they are almost equal, *Protostuff-runtime* deserializes faster than *Kryo* on average in this category, especially when *Java* class types are involved. Regarding the *dCache-like* `PoolManagerPoolUpMessage`, *Protostuff-runtime* is slightly faster than the second best *FST*, and slightly faster with regard to deserializing compared to the second best protocol *Kryo*. For small lists and composite objects, *Protostuff-runtime* is on average the fastest for both encoding and decoding.

Of the three protocols, only *Protostuff-runtime* is platform-independent because it is based on and serializes to the *Protobuf* format. *Kryo* and *Protostuff-runtime* support schema evolution well, whereas *FST* only support limited backward compatibility. The documentation and learning curve of *Protostuff-runtime* is the best of all evaluated frameworks along with that of *Kryo*. *Kryo* is the best framework regarding its minimal introduction effort and easy maintainability, followed by both *Protostuff-runtime* and *Avro*. It is also the best maintained one of the remaining protocols, followed by *Protostuff-runtime*.

Concluding, the results of this evaluation indicate that *JOS*, which is currently used for message encoding in the *dCache* system, should be replaced with *Protostuff-runtime* to increase (de)serialization speeds, enable potential cross-language use cases by serializing to a language

agnostic format and allowing an easy transition to a well documented, easy to use and understand framework.

5.2 Outlook

As the next step, *Protostuff-runtime* needs to be integrated into *dCache*. In order to do so, an approach needs to be chosen that enables backward compatibility with older versions of the system. This means that a cell should be able to serialize a message to either the old or the new format, depending on its capability and what the addressee can interpret. It should be able to decode a binary message with the correct protocol that was used to encode it. One approach would entail appending a sufficiently long sequence of identifier bits for the new protocol to an accordingly serialized byte stream. When deserializing, the first bits would initially be compared to determine whether the new protocol was used. When just the message contents within a **CellMessage** envelope may be encoded with a new protocol, an additional field may indicate the used format. Another option involves opening a different **CellTunnel** for each encoding format, so that it is known when receiving a message through a specific tunnel.

Because *Protostuff-runtime* is a full object graph serializer, not much work is involved in order to be able to serialize a new object. However, in order to improve schema versioning, fields in a *Java* class may be tagged with an identifier that will be used in the internal *Protostuff* schema. If messages are going to be transitioned gradually because of the manual work involved, a strategy for identifying the supported objects needs to be devised, such as storing their class identifiers in a static set.

Finally, the improvements achieved in the *dCache* system should be measured and compared. It might additionally be advantageous to create a way of exposing the messages in a human-readable format such as *JSON*.

6 Appendix

```
1 Bootstrap: docker
2 From: centos:7
3
4 %files
5 /home/lemora/Documents/MsgEncoding.jar /Benchmark.jar
6
7 %post
8 yum -y update
9
10 ## getting java 8
11 yum install -y wget java-1.8.0-openjdk.x86_64
12
13 ## adding some potential mount points
14 mkdir /results
15
16 ## clean up
17 yum clean all
18 rm -rf /var/log/* /var/cache/yum
19
20 %environment
21 export JAVA_HOME="/usr/lib/jvm/jre-1.8.0"
22
23 %runscript
24 java -jar /Benchmark.jar --type ${SER_PROTOCOL} --out /results --name ${
    BM_NAME}
25
26 %startscript
27 java -jar /Benchmark.jar --type ${SER_PROTOCOL} --out /results --name ${
    BM_NAME}
```

Listing 6.1: Singularity recipe file for benchmarker deployment

```
1 syntax = "proto3";
2
3 package ser;
4
5 option java_package = "frameworks.protobuf.generated.protos";
6 option java_outer_classname = "ProtosContainer";
7
8 message Proto_DoubleList {
9     repeated double vals = 1;
10 }
11
12 message Proto_StringList {
13     repeated string vals = 1;
14 }
15
16 message Proto_IntList {
17     repeated int32 vals = 1;
18 }
```

Listing 6.2: *Typelist* schema descriptions for *Protobuf*

```

1 {"namespace": "frameworks.avro.generated.randobjects",
2  "type": "record",
3  "name": "Avro_DoubleList",
4  "fields": [
5    {"name": "values", "type": {"type": "array", "items": "double"}}
6  ]
7 }
8
9 {"namespace": "frameworks.avro.generated.randobjects",
10  "type": "record",
11  "name": "Avro_StringList",
12  "fields": [
13    {"name": "values", "type": {"type": "array", "items": "string"}}
14  ]
15 }
16
17 {"namespace": "frameworks.avro.generated.randobjects",
18  "type": "record",
19  "name": "Avro_IntList",
20  "fields": [
21    {"name": "values", "type": {"type": "array", "items": "int"}}
22  ]
23 }

```

Listing 6.3: *Typelist* schema descriptions for *Avro*

```

1 syntax = "proto3";
2
3 package ser;
4
5 option java_package = "frameworks.protobuf.generated.protos";
6 option java_outer_classname = "ProtosTestobj";
7
8 message Proto_Msg {
9 }
10
11 message Proto_Msg_rw01 {
12
13     int32 _char01 = 1;
14     int32 _byte01 = 2;
15     bool _boolean01 = 3;
16
17     int32 _short01 = 4;
18     int32 _int01 = 5;
19     int64 _long01 = 6;
20
21     float _float01 = 7;
22     double _double01 = 8;
23 }
24
25 message Proto_Msg_rw02 {
26
27     Proto_Msg _msg = 1;
28     Proto_Msg_rw01 _msg_rw01 = 2;
29 }
30
31 message Proto_Msg_rw03 {
32
33     Proto_Msg _msg = 1;
34     Proto_Msg_rw01 _msg_rw01_00 = 2;
35     Proto_Msg_rw01 _msg_rw01_01 = 3;
36 }
37

```

```

38 message Proto_Msg_rw04 {
39
40     repeated string _stringList01 = 1;
41     map<int32,string> _intStrMap01 = 2;
42 }
43
44 message Proto_Msg_rw05 {
45
46     int32 _byte01 = 1;
47     bool _boolean01 = 2;
48
49     int32 _short01 = 3;
50     int32 _int01 = 4;
51     int64 _long01 = 5;
52
53     float _float01 = 6;
54     double _double01 = 7;
55
56     string _string01 = 8;
57 }

```

Listing 6.4: *Composites* schema descriptions for *Protobuf*

```

1  "namespace": "frameworks.avro.generated.randobjects",
2  "type": "record",
3  "name": "Avro_Msg",
4  "fields": [
5  ]
6  }
7
8  {"namespace": "frameworks.avro.generated.randobjects",
9   "type": "record",
10  "name": "Avro_Msg_rw01",
11  "fields": [
12    {"name": "_char01", "type": "int"},
13    {"name": "_byte01", "type": "int"},
14    {"name": "_boolean01", "type": "boolean"},
15    {"name": "_short01", "type": "int"},
16    {"name": "_int01", "type": "int"},
17    {"name": "_long01", "type": "long"},
18    {"name": "_float01", "type": "float"},
19    {"name": "_double01", "type": "double"}
20  ]
21  }
22
23 {"namespace": "frameworks.avro.generated.randobjects",
24  "type": "record",
25  "name": "Avro_Msg_rw02",
26  "fields": [
27    {"name": "_msg", "type": "Avro_Msg"},
28    {"name": "_msg_rw01", "type": "Avro_Msg_rw01"}
29  ]
30  }
31
32 {"namespace": "frameworks.avro.generated.randobjects",
33  "type": "record",
34  "name": "Avro_Msg_rw03",
35  "fields": [
36    {"name": "_msg", "type": "Avro_Msg"},
37    {"name": "_msg_rw01_00", "type": "Avro_Msg_rw01"},
38    {"name": "_msg_rw01_01", "type": "Avro_Msg_rw01"}
39  ]
40  }

```

```

41
42 {"namespace": "frameworks.avro.generated.randobjects",
43  "type": "record",
44  "name": "Avro_Msg_rw04",
45  "fields": [
46    {"name": "_stringList01", "type": {"type": "array", "items": "string"}},
47    {"name": "_intStrMap01", "type": {"type": "map", "values": "string"}}
48  ]
49 }
50
51 {"namespace": "frameworks.avro.generated.randobjects",
52  "type": "record",
53  "name": "Avro_Msg_rw05",
54  "fields": [
55    {"name": "_byte01", "type": "bytes"},
56    {"name": "_boolean01", "type": "boolean"},
57    {"name": "_short01", "type": "int"},
58    {"name": "_int01", "type": "int"},
59    {"name": "_long01", "type": "long"},
60    {"name": "_float01", "type": "float"},
61    {"name": "_double01", "type": "double"},
62    {"name": "_string01", "type": "string"}
63  ]
64 }

```

Listing 6.5: *Composites* schema descriptions for *Avro*

Bibliography

- [1] I. Elaine Allen and Christopher A. Seaman. Likert scales and data analyses. *Quality Progress*, 40(7):64–65, 2007.
- [2] Sebastian Bittl, Arturo A Gonzalez, M Spaehn, and W Heidrich. Performance comparison of data serialization schemes for ETSI ITS car-to-X communication systems. *International Journal On Advances in Telecommunications*, 8:48–58, 2015.
- [3] D. E. Damasceno Costa, C. Bezemer, P. Leitner, and A. Andrzejak. What’s Wrong With My Benchmark Results? Studying Bad Practices in JMH Benchmarks. *IEEE Transactions on Software Engineering*, pages 1–1, June 2019.
- [4] Huw Evans. Why Object Serialization is Inappropriate for Providing Persistence in Java. Technical report, Department of Computing Science, University of Glasgow, 2000.
- [5] Brian Goetz. OpenJDK: Towards Better Serialization. <http://cr.openjdk.java.net/~briangoetz/amber/serialization.html>, June 2019. Accessed: 2019-07-14.
- [6] Wilhelm Hasselbring and Guido Steinacker. Microservice Architectures for Scalability, Agility and Reliability in E-Commerce. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 243–246, April 2017.
- [7] W. He and L. D. Xu. Integration of Distributed Enterprise Applications: A Survey. *IEEE Transactions on Industrial Informatics*, 10(1):35–42, February 2014.
- [8] M. Herlihy and B. Liskov. Value Transmission Method for Abstract Data Types. *ACM Transactions on Programming Languages and Systems*, 4:527–551, 1982.
- [9] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(6):1811–1841, 1994.
- [10] K. Maeda. Performance evaluation of object serialization libraries in XML, JSON and binary formats. *2012 Second International Conference on Digital Information and Communication Technology and it’s Applications (DICTAP)*, pages 177–182, May 2012.
- [11] Paul A. Millar, Olufemi Adeyemi, Gerd Behrmann, Patrick Fuhrmann, Vincent Garonne, Dmitry Litvinsev, Tigran Mkrtchyan, Albert Rossi, Marina Sahakyan, and Jurgen Starek. Storage for Advanced Scientific Use-Cases and Beyond. *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, pages 651–657, March 2018.
- [12] D. L. Parnas. On the Criteria to Be Used in Decomposing Systems into Modules. *Commun. ACM*, 15(12):1053–1058, December 1972.
- [13] Dennis Pfisterer, Marco Wegner, Horst Hellbrück, Christian Werner, and Stefan Fischer. Energy-optimized data serialization for heterogeneous WSNs using middleware synthesis. *Proceedings of the 6th Annual Mediterranean Ad Hoc Networking Work-Shop (Med-Hoc-Net)*, 2007.

- [14] Michael Philippsen and Bernhard Haumacher and Christian Nester. More efficient serialization and RMI for Java. *CONCURRENCY: PRACTICE AND EXPERIENCE*, 12:495–518, 2000.
- [15] Julien Ponge. Avoiding Benchmarking Pitfalls on the JVM. Java Magazine, issue July/August 2014, July 2014.
- [16] Roger Riggs, Jim Waldo, Ann Wollrath, and Krishna Bharat. Pickling State in the Java System. *Computing Systems*, 9:291–312, January 1996.
- [17] Jacopo Soldani, Damian Tamburri, and Willem-Jan Van Den Heuvel. The Pains and Gains of Microservices: A Systematic Grey Literature Review. *Journal of Systems and Software*, 146, September 2018.
- [18] Clarence Tauro, N. Ganesan, Saumya Mishra, and Anupama Bhagwat. Object Serialization: A Study of Techniques of Implementing Binary Serialization in C++, Java and .NET. *International Journal of Computer Applications*, 45:25–29, May 2012.
- [19] Maarten van Steen and Andrew S. Tanenbaum. A brief introduction to distributed systems. *Computing*, 98(10):967–1009, October 2016.
- [20] www.avro.apache.org. Welcome to Apache Avro. <https://avro.apache.org/>. Accessed: 2019-07-28.
- [21] www.dcache.org. dCache Website. <https://www.dcache.org>. Accessed: 2019-07-10.
- [22] [www.developers.google.com/protocol buffers](https://developers.google.com/protocol-buffers/). Protocol Buffers | Google Developers. [urlhttps://developers.google.com/protocol-buffers/](https://developers.google.com/protocol-buffers/). Accessed: 2019-07-27.
- [23] [www.en.esotericsoftware.com](http://en.esotericsoftware.com). Spine: 2d skeletal animation for games. <http://en.esotericsoftware.com/>. Accessed: 2019-08-07.
- [24] [www.github.com/EsotericSoftware/kryo](https://github.com/EsotericSoftware/kryo). Java binary serialization and cloning: fast, efficient, automatic. <https://github.com/EsotericSoftware/kryo>. Accessed: 2019-07-21.
- [25] [www.github.com/protostuff/protostuff](https://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.
- [26] [www.github.com/RuedigerMoeller/fast serialization](https://github.com/RuedigerMoeller/fast-serialization). FST: fast java serialization drop in-replacement. <https://github.com/RuedigerMoeller/fast-serialization>. Accessed: 2019-07-21.
- [27] www.grpc.io. gRPC. A high performance, open-source universal RPC framework. www.grpc.io. Accessed: 2019-07-27.
- [28] [www.hadoop.apache.org](https://hadoop.apache.org). Apache Hadoop. <https://hadoop.apache.org/>. Accessed: 2019-07-28.
- [29] [www.hessian.caucho.com](http://hessian.caucho.com). Hessian Binary Web Service Protocol. <http://hessian.caucho.com>. Accessed: 2017-07-21.
- [30] www.java.com/en. Java | Oracle. <https://www.java.com/en/>. Accessed: 2019-07-28.
- [31] [www.jbossmarshalling.jboss.org](https://jbossmarshalling.jboss.org). JBoss Marshalling. <https://jbossmarshalling.jboss.org>. Accessed: 2019/07/21.
- [32] www.jfree.org/jfreechart. JFreeChart. <http://www.jfree.org/jfreechart>. Accessed: 2019-07-31.

- [33] [www.kafka.apache.org](https://kafka.apache.org/). Apache kafka – A distributed streaming platform. <https://kafka.apache.org/>. Accessed: 2019-07-24.
- [34] [www.kotlinlang.org](https://kotlinlang.org/). Kotlin Programming Language. <https://kotlinlang.org/>. Accessed: 2019-07-28.
- [35] [www.openjdk.java.net/projects/code tools/jmh](https://openjdk.java.net/projects/code-tools/jmh). OpenJDK: jmh. <https://openjdk.java.net/projects/code-tools/jmh>. Accessed: 2019-07-31.
- [36] [www.singularity.lbl.gov](https://singularity.lbl.gov/). Singularity. <https://singularity.lbl.gov/>.
- [37] [www.spark.apache.org](https://spark.apache.org/). Apache Spark – Unified Analytics Engine for Big Data. <https://spark.apache.org/>. Accessed: 2019-08-07.
- [38] [www.wiki.openjdk.java.net/display/HotSpot/PerformanceTechniques](https://wiki.openjdk.java.net/display/HotSpot/PerformanceTechniques). PerformanceTechniques – HotSpot – OpenJDK Wiki. <https://wiki.openjdk.java.net/display/HotSpot/PerformanceTechniques>. Accessed: 2019-08-01.
- [39] [www.zookeeper.apache.org](https://zookeeper.apache.org/). Apache ZooKeeper. <https://zookeeper.apache.org/>. Accessed: 2019-07-22.

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht.

Die Arbeit wurde bisher in ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Ort, Datum

Lea Morschel