



Bachelor Thesis

On data shape and its impact on relative performance of serialisation libraries

by

Floris Bastiaan

(Student Number: 2808922)

First supervisor: Daniele Bonetta

Daily supervisor: Hexiang Chang

Second reader: Alexandru Iosup

27 August 2026

*Submitted in partial fulfilment of the requirements for
the degree of Bachelor of Science in Computer Science*

On data shape and its impact on relative performance of serialisation libraries

Floris Bastiaan

Vrije Universiteit Amsterdam

Amsterdam, The Netherlands

`f.h.bastiaan@student.vu.nl`

Abstract

Data serialisation takes up a significant portion of hardware utilisation in some of the largest datacentres in the world. These datacentres account for a significant share of global energy use. Because of this, there are a lot of existing benchmarks comparing different serialisation protocols and their implementations. Most of these benchmarks are built on the assumption that a single dataset is enough to show the relative performance of the implementations. However, as we show in this thesis, the shape of the data that's being serialised has a large impact on the results of such a comparison. We design and implement a benchmarking framework to compare Cap'n Proto, FastJson2, Apache Fory, Gson, MessagePack, and Protocol Buffers. The framework takes in multiple datasets, which were sourced from real-life, large-scale projects. The results from this benchmark indicate that the shape of the data being serialised is highly important when deciding which serialisation library to use. For one dataset, MessagePack is 2.5 times as fast as Gson. But for another dataset, Gson is not only faster than MessagePack, it's 16 times faster. This means that when a developer consults a serialisation benchmark, it's important for them to consider if the shape benchmark data matches that of the data in their project. They should perform their own benchmark using data that matches their use case, which can be done using our extensible benchmarking framework Appendix A.4. Failing that, they could read the descriptions of the datasets in Section 5.3 and see which set matches their own data the best.

Keywords: Data Serialisation, Deserialisation, Performance, JSON, Benchmarking.

Contents

1	Introduction	5
1.1	Context	5
1.2	Problem statement	5
1.3	Research Questions	5
1.4	Methodology	6
1.5	Contributions	7
1.6	Societal Relevance	7
1.7	Statement of Authorship	7
1.8	Reading guide	7
2	Background	8
2.1	Serialisation protocols	8
2.2	Data shape	10
2.3	Effects of data shape on serialisation performance	11
2.4	Related Work on Serialisation Benchmarking	11
3	Design	14
3.1	Extensibility	14
3.2	Final design	15
4	Implementation	17
4.1	The class generation script	17
4.2	The Java benchmark	17
5	Experiments	21
5.1	Experimental setup	21
5.2	JVM & JMH Settings	21
5.3	Datasets	22
5.4	Findings	25

5.5	Limitations	32
6	Conclusion	33
6.1	Threats to validity	34
6.2	Future Work	35
	References	37
A	Benchmarking framework	39
A.1	Abstract	39
A.2	Artefact check-list (meta-information)	39
A.3	Description	39
A.4	Experiment workflow and customisation	39
A.5	Evaluation and expected results	40
B	Results	40

1

Introduction

1.1 Context

Serialisation benchmarking has been done many times. Yet even now, an important factor in determining performance is often overlooked.

Data serialisation and deserialisation are a significant cost in many companies. A significant portion of hardware utilisation in some of the world’s largest datacentres is spent on this task. The point of a server is to receive some data, do work, then send something back. While this work itself differs server by server, this exchange of data is a universal constant. In 2021, Google published [23] that they spend almost 10% of all their datacentres’ total CPU utilisation on data serialisation and deserialisation. ByteDance, the parent company of TikTok [3], has had instances of 40% of their total CPU usage going to (de)serialisation [2].

1.2 Problem statement

There are a lot of different serialisation protocols, and developers frequently ask which one is the fastest [4]. So it makes sense that comparing different serialisation protocols has been done many times [23, 27]. Each time someone releases a new protocol, they often have a set of benchmarks proving that their shiny new tool is the fastest [5–7]. However, these benchmarks often only use a single dataset. And even among the papers we found that do use multiple datasets, none of them had the goal of determining what data shape works well with what protocol/implementation. In this thesis, we show that data shape can have a large impact on the relative performance of serialisation protocols.

1.3 Research Questions

The goal of this work is to answer the Main Research Question (MRQ): How does data shape affect the relative performance of serialisation implementations?

1. INTRODUCTION

To answer this research question, a benchmark suite will be designed and implemented that can take in real-life datasets to measure existing libraries that implement these serialisation protocols. This process is guided by the following sub-questions.

1.3.1 Main Research Question (MRQ):

How does data shape affect the relative performance of serialisation implementations? There are many different serialisation protocols, and many even have multiple implementations per programming language. Many of these protocols and their implementations claim to be the fastest, showing some kind of benchmark result on their GitHub page. At first glance, this seems to be a contradiction. How can there be more than one “fastest”? But it is entirely possible for protocol A to be optimised for data shape A, and protocol B to be optimised for data shape B. The results of a benchmark comparing these two will then obviously depend not on the protocols themselves, but on the data shape chosen for the benchmark. This means we need to test different data shapes. This question will be answered mainly in Section 5.

1.3.2 Side Research Question 1 (RQ1):

How do the rankings of serialisation libraries vary across real-world datasets with different data shapes?

As elaborated on in Section 2.4, most existing benchmarks use only a single dataset, with the implicit assumption that these results will hold for most other datasets. The goal of this thesis is to show that this is not the case, even for real-world datasets that all represent real API traffic. Real-world data is used to prove that this is not just a theoretical issue that only holds for certain contrived datasets.

1.3.3 Side Research Question 2 (RQ2):

Which characteristics of those datasets can explain the observed differences?

Besides knowing *that* different shapes impact serialisation libraries differently, it is also important to know *what part* of the shape is responsible for the performance hit. This is difficult with real-life datasets, as there are often multiple differences between datasets.

1.4 Methodology

In order to answer all research questions, we will design, implement, and run a benchmarking framework, which is able to compare multiple different serialisation libraries using multiple datasets with different shapes.

1.5 Contributions

This thesis will show that the relative performance of different serialisation protocols differs greatly given different data shapes. These results will be relevant in any future work that aims to compare serialisation protocols, and will also be relevant for developers who wish to determine which serialisation protocol and implementation is best for their project. We will also analyse what qualities in the datasets are likely to be responsible.

1.6 Societal Relevance

A significant portion of hardware utilisation in datacentres is being spent on (de)serialisation. In 2021, Google published they spent 10% of CPU utilisation on (de)serialisation [23]. That same year, datacentres used almost 300 TWh, or around 1% of global electricity consumption [8]. Therefore, making (de)serialisation more efficient would reduce electricity usage.

1.7 Statement of Authorship

I confirm that this work is my own and not copied from any other person, LLM, website, or other source, unless otherwise indicated, and that this work has not been submitted for assessment elsewhere.

1.8 Reading guide

In Section 2 we present background information on serialisation protocols and existing benchmarks. To fully understand this thesis, it is recommended to start here. Section 3 shows a generalised design of our benchmarking framework, whereas Section 4 explains how the concrete Java implementation of this design was created. Section 5 contains the results obtained using the benchmarking framework, along with several findings based on these results. For those who wish to skip to a summary of the findings along with future work, see Section 6.

2

Background

2.1 Serialisation protocols

A serialisation protocol is a method of turning an “object” as it’s stored inside the memory of a computer program into a single stream of bytes. These bytes can then be sent over the network, stored in a file, or, if one wishes to upset a database architect, inside a database.

There are two main reasons for serialisation. The first is when there is some object stored in the memory of program A, and it has to be sent to another program B (possibly running on a different machine, and possibly written in a different programming language). The second reason is when this object has to be stored to a disk or database for later retrieval.

A very simple serialisation protocol would be to send the data as it’s stored in memory. This runs into two major issues. The first is that the same conceptual structure might be stored in two completely different ways in two different programming languages or hardware architectures. But the even larger issue is that most programming languages allow for objects which are not stored in contiguous blocks of memory. So the goal of these serialisation protocols is to design a system to turn these sparse graphs spread out throughout the program memory, stored in a different way for each programming language and hardware architecture, into a single, uniform and (usually) language-agnostic representation. There are many different ways to implement serialisation, all with their own trade-offs.

The most popular serialisation protocol in web development is JavaScript Object Notation (JSON)[9]. This protocol, as the name implies, was made to specifically work well with the JavaScript programming language, but nowadays there is a JSON library for all the major (and most minor) programming languages [10]. For most of these languages, there are even multiple, with many competing claims of being the fastest.

This study will look at the following serialisation protocols, chosen because of their popularity, difference in design, and/or support by major industry leaders:

- JavaScript Object Notation (JSON)
- Google Protocol Buffers (ProtoBuf)
- Apache Fory (Fory)
- MessagePack (MsgPack)
- Cap'n Proto (Capnp)

JSON is a text-based protocol, designed to be easily readable by humans. However, for many use cases, this readability is rarely needed, since this data is usually generated and consumed only by computers, as this is the entire point of data serialisation. This has led to binary protocols which can often be more compact and are designed to be easily read and created by computers instead of by humans.

ProtoBuf was originally developed internally at Google. It has since been open-sourced and is still extensively used in many of Google's products [11]. Since it is now open source, it is also used by other companies. But Google's usage alone justifies including it here, as Google, together with Amazon and Microsoft, operates roughly half of the world's hyperscale datacentre capacity [1]. The performance characteristics of a protocol that is frequently used within one of these three companies have a major impact on global datacentre CPU utilisation, independent of ProtoBuf's broader adoption elsewhere. Unlike JSON, ProtoBuf usually requires a predefined schema (a .proto file). Serialisation and deserialisation code specific to this schema is then generated using the ProtoBuf compiler, `protoc` [12].

Apache Fory (which used to be known as Fury [13]) uses JIT dynamic compilation [13]. This means that it dynamically generates optimised and specialised code as it's being used, using profiling data collected as it is running to do so. This approach is already used in many programming languages themselves [14]. The 1.0 version of this library was released just before work on this thesis was started [15]. Fory can be used with XLang or native mode. XLang stands for cross-language. In native mode, a message can only be deserialised in the same programming language as it was serialised in.

MessagePack's slogan is "It's like JSON. But fast and small." [16] As this slogan implies, it mainly intends to be a replacement for JSON. Just like JSON, MessagePack does not use a schema. However, whereas JSON is text-based, containing much

2. BACKGROUND

redundant information for human readability, MessagePack aims to heavily cut down on this. The data `{"shouldDisplay":true,"opacity":100}` takes up 36 bytes in JSON, one for each character in that text. In MessagePack, this would be `\x82\xADshouldDisplay\xC3\xA7opacity\x64`, only 25 bytes (here, any sequence formatted as `\xAB` is a single byte with the hexadecimal value AB).

The creator of Cap'n Proto was also the primary author of ProtoBuf version 2, although Capnp is not affiliated with Google. Among other big changes compared to ProtoBuf, Capnp is a zero-copy protocol. This means that it doesn't really have a serialisation or deserialisation step, but instead uses the same representation in program memory as it does on the wire [17]. This introduces some difficulties in benchmarking, and these are addressed in Section 3.1.3.

2.2 Data shape

Usually, when data is sent or received by an API, all those messages will conform roughly to the same “shape”. The simplest possible data shape is simply a single boolean value, which could be represented by a single bit, but could just as well be represented by the 5-byte string “false”. Also very simple to serialise would be an array of numbers stored contiguously in program memory. In certain protocols, both of these examples might be sent exactly as they are stored in memory. In many programming languages, objects might contain cycles. This is usually achieved with pointers. Sometimes explicitly, as in C, sometimes hidden, as in Java [18]. For an example of such a structure, let's say we have an API for a library. We might have an object that contains info about a certain book. This info includes the author, which in turn contains an array of books. The way this would be stored in program memory would be with two objects, each containing a pointer to the other. If we were to naively serialise this, the serialiser might get stuck following these pointers back and forth ad infinitum. There are multiple solutions to this, but they have to be implemented either by the creator or by the user of the serialisation libraries.

Besides these two examples, there are many more possible shapes. Deeply nested, sparse, huge lists of numbers, very long strings, etc. Some of these

In addition, the exact shape might not really be perfectly fixed. There are many situations in which a field might only exist sometimes, or might sometimes contain a string and sometimes an integer.

2.3 Effects of data shape on serialisation performance

In this thesis, we will show that the shape of the data will affect how easy that data is to serialise and deserialise. When the data being sent is always a positive number that fits in 32 bits, we can simply have those exact 4 bytes be the result of serialisation. Then, when deserialising, we can simply read those 4 bytes and interpret them as an integer without any validation needed, as any sequence would be a valid number. However, if the data being sent can either be such an integer or a text string, we need a byte to determine the data type, called the discriminator, followed by either 4 bytes in the case of a number or the bytes that represent the string. We also need to find some way to signify when the string ends, either using a termination byte or by prepending the string with its length. Then, when deserialising again, we need to read that first discriminator byte, then the code needs to branch. If it's an integer, the process is the same as before. However, if it's a text string, we probably want to check if the bytes we received are actually valid in the encoding that is being used. This means branching again, but for every single byte received. This checking takes time, and these branches are bad for performance, especially if they are hard to predict [21]. There are ways to optimise for certain data types and shapes. Some optimisations can be made when designing the protocol, and some can only be made when implementing it, and many of them are trade-offs, reducing performance in the cases they aren't targeting. This means that different protocols and even different implementations of these protocols will likely be optimised for differing cases, and our goal will be to test many different protocols with a diverse set of data shapes, as stated in Section 1.3.

2.4 Related Work on Serialisation Benchmarking

Much work has been done to evaluate the performance of different serialisation protocols and implementations. Vanura et al. [26] compared 49 different serialisation libraries across multiple programming languages that implement 5 different protocols. Their methodology uses a single randomly generated 12 kB JSON dataset.

Maida et al. [25] compare 11 serialisation libraries for Java, just like this thesis. However, since 2012, new libraries and protocols have come out, and many existing libraries have received updates improving performance. Their dataset is also randomly generated data based on a very small and simple struct. They do test by modifying

2. BACKGROUND

the length of the array (only) in this struct, and show how performance scales with this metric.

Jackson et al. [22] compare 12 different serialisation protocols, with “the appropriate Python client library” for each. Which libraries or versions these are is unclear, as is the Python version, since it is not stated in the paper, and the GitHub link to their paper leads to a 404 not found page. They show their results both in total latency and in throughput, normalised by the size of the data. It is unclear if this is the size as it is natively stored in the program, the serialised size in the protocol being tested, or the size in a standardised protocol. They use 8 different datasets, some of which are randomly generated, and some are real-life datasets. Their datasets are not representative of only API data, but instead a broad selection of fields. Their results also appear to show that for different datasets, relative latency between libraries can change by multiple magnitudes.

Viotti et al. [28] aim to compare serialisation protocols not through benchmarking but by comparing their histories and describing the design decisions that were made by their authors. This can provide some insight into how different data shapes can impact the relative performance of these protocols. For example, they explain that certain protocols utilise ZigZag encoding, allowing them to store negative integers that are close to zero more compactly. However, many performance characteristics will be determined by implementation-level decisions.

Viotti et al. [27] compare 13 serialisation protocols. They focus on serialised message size, and therefore don’t need to use multiple implementations per protocol. They use 400 different input JSON files, and have aimed to classify them using a standardised system according to 4 metrics:

- Size. Minified size determines the “tier”, with tier 1 being less than 100 bytes, tier 2 being between 100 and 1000 bytes, and tier 3 being 1000 bytes or more. Why these specific cut-offs were chosen or why exactly three tiers is unclear.
- Type. They determine if a JSON document is textual, numeric, or boolean based on which of those types represents the largest amount of bytes in the document.
- Redundancy. According to their specification, a JSON document is redundant if 25% or more of its content is duplicated. It’s unclear how this deals with boolean values,

as they only have 2 states and therefore will increase the amount of duplicated content.

- Nestedness. A JSON document is either nested or flat, according to a formula that is better read in their paper.

This categorisation is better than nothing, but many of the thresholds seem arbitrary. Viotti et al. [29] introduce a new protocol and implementation, and benchmark it for serialised message size, just as they did in [27]. They use that same categorisation system as in that paper, but for this benchmark they only use 27 different JSON documents.

Langdale et al. [24] introduce `simdjson`, an implementation for JSON which uses SIMD instructions. They also benchmark this new implementation together with two other JSON libraries. This benchmark is done using 14 different JSON datasets. As their benchmark only focuses on JSON parsing, they characterise these datasets by how many bytes in the dataset belong to a certain subset (numerical, structural, null, etc). The results on these different datasets are then averaged for each library. Although doing so is useful to get a single number to compare these implementations, it erases the fact that for some datasets, `simdjson` was almost 5 times as fast as `RapidJSON`, for others, this factor was less than 2. Both results are valid, and readers should consider their own data to determine what speedup they should expect if they switch from `RapidJSON` to `simdjson`.

`Simdjson` was not included in our benchmark as it is not frequently used in Java [19].

3

Design

3.1 Extensibility

The benchmark needs to support many different serialisation protocols and implementations. It also has to be able to take in many different real-life datasets. Finally, we must be able to benchmark these different serialisation protocols according to industry standards on proper benchmarking.

This benchmark will need to be very flexible. This is because we want to add many different datasets and allow future researchers to do the same, so we want to minimise the amount of new code that needs to be manually written per added dataset. The same is true for adding a new serialisation protocol. These two issues are solved in the following ways:

3.1.1 New datasets

We want to first deserialise the datasets as they are formatted on the disk to an object native to the programming language in which the benchmark runs. For compiled languages, this requires a class or struct whose structure matches that of the entries in the dataset. Writing these manually would be tedious and error-prone. Therefore, we need a script that needs to be run once for each newly added dataset. This script will read the dataset and generate an appropriate class or struct. It would also be possible to require each dataset to be accompanied by a schema file in some standard format, but this would make sourcing datasets a lot harder.

3.1.2 New serialisation protocols

For each new serialisation protocol we want to write only a minimal amount of code to connect an existing library which implements said protocol to our existing benchmarking infrastructure. We also want the benchmarks to be highly similar to ensure the test is fair. This all points to making a single shared interface for benchmarking.

3.1.3 Zero-copy protocols

Some of the serialisation protocols that are being benchmarked here are zero-copy protocols. This means they use the same binary representation in memory as they do on the wire. This means they have no real serialisation or deserialisation step. Instead, the libraries that implement them expose methods to read or write these fields directly in this format. This means measuring them as we do the other serialisation protocols would result in a perfect score, as seen in the (not so serious) Figure 1.

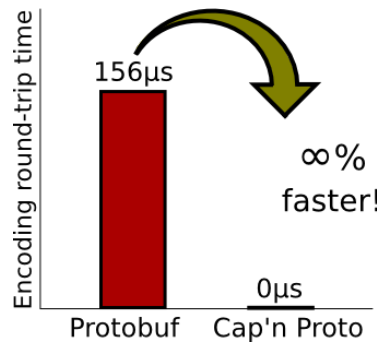


Figure 1: joke graph from <https://capnproto.org>.

However, this reading and writing with a non-native representation is slower than doing the same in a native format. So, we will measure this reading and writing in place of serialising and deserialising. In a real application, only a subset of these fields will be read. Therefore, we must make it clear in our results that these measurements must be multiplied by the fraction of fields that is expected to be used.

3.2 Final design

The benchmarking framework will work with the following steps:

1. The generator script will automatically generate class definitions native to the programming language in which the rest of the framework is written. This script will generate a class for each dataset, and, recursively, for each non-primitive field in that class. For many items in the datasets, certain fields are set to `null`. This means this generator script will have to iterate through each entry in the dataset until either all field types have been determined or all entries have been checked.
2. The following steps will be repeated for each serialisation library that is to be tested:
 1. The following steps will be repeated for each dataset, times 15:
 1. Load the dataset into memory represented as the classes generated in step 1 and store these in array A

3. DESIGN

2. Serialise each item and store the results into array B
3. Start measurement A.
4. Serialise each item in array A. Discard the result in a way which prevents the benchmark from being optimised away.
5. Stop measurement A.
6. Start measurement B.
7. Deserialise each item in array B. Discard the results in the same way.
8. Stop measurement B.

The “times 15” here is to collect multiple data points.

Another possible approach is to serialise each item only once, and use that to both measure and to store the result for the deserialisation test. However, doing so risks measuring the time it takes to store the result.

Each entry in the dataset is serialised and later deserialised one by one, instead of serialising the entire dataset in one go. This is because these datasets are taken from real-life APIs, where the items would also be serialised, sent, and deserialised one at a time as a web request or response.

The total time of serialising an entire dataset is measured instead of the time for each item. (although the items are serialised individually). These two approaches would result in the same arithmetic mean result (after dividing by the size of the dataset), but the standard deviation would be different. By measuring all at once, the standard deviation reflects how consistent the measurements are over multiple runs. When measuring one by one, it instead reflects how different the items in the dataset are when it comes to serialisation times. Both numbers can be useful, but for this thesis it seemed more important to check if our results are valid.

4

Implementation

4.1 The class generation script

For every dataset, we need a matching plain Java class. The dataset can then be loaded into memory as this class during the setup step, after which we can measure the serialisation and deserialisation performance. To create these classes, we will create a python script which will read through the datasets and detect the structure of the data. Python was picked for this as it's often used for data transformation, and because of the author's familiarity with the language. This script can be found at <https://github.com/FBastiaan04/bsctest>.

4.2 The Java benchmark

This benchmark is implemented in Java, as this language is used all over the industry, there exists a widely used benchmarking suite for it (JMH), and many serialisation protocols have official and well-document Java implementations. However, there is no reason this benchmark couldn't be ported over to other programming languages, especially using LLMs. We will first create an abstract class which takes a generic type parameter to indicate what the library serialises to (usually `string` or `byte[]`). This abstract class has a `benchSerialise` and a `benchDeserialise` method, which are marked with JMH's `@Benchmark` annotation, and take in a JMH `Blackhole`. These methods iterate through all items in the current dataset, and call the `serialiseItem`/`deserialiseItem` abstract method and pass the result on to `Blackhole.consume`, which prevents the JVM from realising that the code isn't doing anything useful and can just be optimised away. This abstract class can then be inherited for each serialisation library. For each of these concrete subclasses, only the `serialiseItem` and `deserialiseItem` and optionally a `setup` method need to be implemented.

Here it also becomes relevant to talk about the libraries that implement these protocols. These were the libraries used for each protocol:

4. IMPLEMENTATION

Table 1: Tested serialisation libraries.

Protocol	Library (Maven group id)	Version
JSON	com.google.code.gson	2.13.2
JSON	com.alibaba.fastjson2	2.0.62
Cap’n Proto	org.capnproto.runtime	0.1.16
Fory	org.apache.fory	1.1.0
MessagePack	org.msgpack	0.9.12
ProtoBuf	io.protostuff	1.7.4

JSON and ProtoBuf are the only two protocols where we did not use the official implementation. For JSON, this is because the official implementation (JSON-Java) is not actually the one the industry uses. The most commonly used JSON libraries are Jackson and GSON. [20] Besides these popular libraries, there are also a great number of less-used libraries which claim to be multiple times faster. Some of them are hobby projects that achieve these reported speedups by simply not implementing certain features. Others are made by large companies that decided they wanted more speed than either of the industry-standard libraries offers. One such library is also tested here, namely Alibaba’s FASTJSON 2, which, according to their own benchmarks, is more than 3 times faster than either GSON or Jackson. [5]

For ProtoBuf, the official implementation was incompatible with the benchmarking framework. Most serialisation libraries expose a serialisation function which can be passed a Java Object and returns either a string or a byte array, together with a deserialisation function which does the opposite. ProtoBuf instead requires the user to create a `.proto` file which specifies the shape of the data in a language-independent way. Based on this file, a `Builder` Java class is then generated. This means our Python script will have to be modified to also output these `.proto` files. But that’s the easy part. There is still no simple `ProtoBuf.serialize()` method. This means we must either not use the `abstractBenchmark` class and create a new benchmark from scratch, or we use Java’s reflection to create a function which can transform our existing Java objects into a `ProtoBuf Builder`.

The first option would be a lot of work and has a risk of introducing unfairness. The more the code diverges per benchmark, the less fair our benchmark will be. The second option would be even more work, and would likely severely disadvantage ProtoBuf, as this extra reflection step would be slow and not accurate to real-life usage. Instead of either of these two options, we decided to use a library with an API that is more similar to the other serialisation libraries, namely ProtoStuff. We will be using Java Flight Recorder, which we will use, among other things, to analyse how much overhead this library contains.

At first, Fory's official (and only) implementation library was causing crashes in the framework. The error message indicated a memory access violation. This protocol requires any fields which can be `null` to be marked with the `@Nullable` annotation, and any primitive fields to use Java's boxed variants. These boxed variants are wrapper classes around the primitives, which allow them to be set to `null`. This probably should have been done from the start, as many datasets contain `null` for primitive fields, and they were then incorrectly loaded into Java objects. Instead of being `null`, they would be initialised to their type's default value.

The official library for Cap'n Proto works very similar to the official ProtoBuf library. Sadly, there is no ProtoStuff equivalent for Cap'n Proto. Such a library also wouldn't make a lot of sense, as Cap'n Proto is a zero-copy protocol. The intended use of this protocol is that the user utilises builder functions provided by the implementation library to build the object in the exact representation in which the message will be sent, meaning that there is no real serialisation step. As stated in Section 3.1.3, we will be measuring this building step, as it is the equivalent of the serialisation step in a regular protocol. Thus, we will be using the second approach we proposed for ProtoBuf. The Python script was modified to output `.capnp` files next to the `.java` files, and the `CapnpBenchmark.java` file was modified to use Java reflection during the `setup` function and store references to each getter and setter method on the builders and readers generated from the `.capnp` files. The reason this would not be a good approach for ProtoBuf is that there we would be measuring both this getting and setting in addition to the serialisation and deserialisation. For a zero-copy protocol like Cap'n Proto, this isn't the case, as getting and setting are in essence the same as the serialisation and deserialisation steps.

4. IMPLEMENTATION

After the Python script has finished, there is a plain Java class and a matching Cap'n Proto builder and reader class for each dataset. During the setup step, we create a list of `FieldAccessors`. Each `FieldAccessor` holds a `MethodHandle` for both the getter and the setter on the builder and reader. It also contains a `VarHandle` to the field of the plain Java class. This means that during the serialisation benchmark, we can iterate through this list of `FieldAccessors` and get the value of the plain Java object using the `VarHandle`, then set it on the Cap'n Proto builder using the setter `MethodHandle`. And vice versa for the deserialisation benchmark. Creating these `MethodHandles` and `VarHandles` is by far the slowest step, so doing this inside the setup step should greatly reduce the overhead during the benchmark itself. It is also possible to modify the Python script so it generates conversion methods for each plain Java class which convert plain Java objects into completed Cap'n Proto builders of the matching type, and vice versa for readers. But that Python script has already grown quickly in scope, and as we are using Java Flight Recorder, we are able to measure this overhead. These numbers can be found in Section 5.

The total implementation can be downloaded or read at <https://github.com/FBastiaan04/bsctest>. The datasets are not included on GitHub, as the files are too large for the Git protocol. They can be downloaded separately at <https://drive.google.com/drive/folders/1jP1VB7-ScjNPwucivnh7JlBzwWRYyczN>.

5

Experiments

5.1 Experimental setup

The experiments were run on a node in the AtLarge research cluster. This machine has 256 GiB of RAM and an Intel Xeon Silver 4416+ CPU. This CPU has 20 cores which run at 800-2000 MHz. None of the tested libraries are GPU-accelerated. During the experiment, we did our best to ensure no other processes on this node were using significant resources. This experiment could be replicated on any other machine, but having at least 32 GiB total memory is recommended, and hardware utilisation from other processes should be minimised.

The benchmark was compiled using Java Development Kit 24.0.2 for Windows, and executed using Java Development Kit 24.0.2 for Linux x64 on the machine discussed above.

5.2 JVM & JMH Settings

We use `jvmArgsAppend("-Xms16g", "-Xmx16g", "-XX:+UseG1GC")`. The first two arguments specify the initial and maximum heap memory size, setting both to 16 GiB. The last argument specifies which garbage collector to use. This is mainly for reproducibility and was picked arbitrarily.

Besides this we also use

```
.mode(Mode.AverageTime)
.timeUnit(TimeUnit.MICROSECONDS)
.warmupIterations(5)
.measurementIterations(5)
.forks(3)
.addProfiler(JavaFlightRecorderProfiler.class)
```

Everything will be serialised on a single thread, so measuring throughput or latency is essentially the same, just the inverse of the other. Some warmup iterations are

5. EXPERIMENTS

needed, and from manual testing it appears that performance stabilises after three iterations for all the libraries that were tested here. 5 iterations and 3 forks means 15 measurements per library per dataset. This results in a couple of hours of total benchmark time.

5.3 Datasets

This thesis focuses on the typical data sent or received by an API. As such, we will use datasets that are large arrays of objects with similar shapes. Each of these entries was sent one by one by an API. The framework doesn't serialise the entire array, but instead serialises each entry one by one.

The datasets used here were chosen to be roughly the same total size when measured in bytes encoded as JSON. Each is around 1 GB, with the smallest being 0.84 GB and the largest 1.21 GB. This is not strictly needed, but the datasets should be big enough to reduce variance and to allow JFR to do proper profiling.

Table 2: Dataset characteristics.

	Bestbuy	Google	Twitter	Walmart	Wiki
Total size (Bytes)	1,044,158,893	1,136,060,129	842,634,884	994,805,150	1,098,712,599
Entries	230,089	4,439	155,489	272,499	121,805
Size/entry (Bytes)	4,538	255,927	5,419	3,650	9,020

Table 3: Dataset descriptions.

Dataset	Description
Bestbuy	Shallow, wide
Google	Narrow, deep, very large entries
Twitter	Complicated, deep, wide
Walmart	Shallow, lot of null values
Wiki	Recursive type, very deep

5.3.1 Bestbuy

Entries in the Best Buy dataset (Bestbuy) are very shallow and wide. The root struct has 284 fields, 130 of them strings, 75 booleans, 31 integers and 19 floats. The other 29 are lists, 10 of which are lists of strings, the other 19 are lists of structs, all having only 1-3 string or integer fields.

Entries in the Best Buy dataset (Bestbuy) have the following shape:

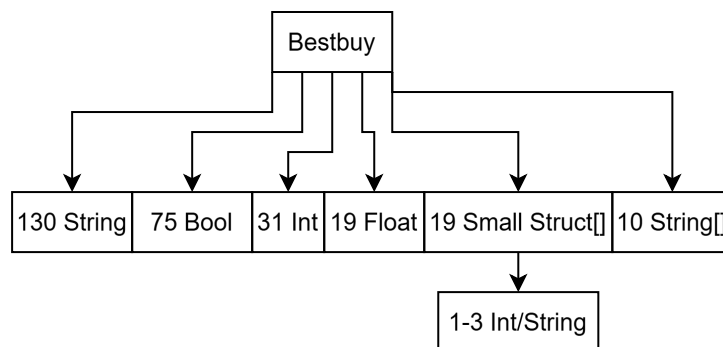


Figure 2: Bestbuy data shape.

In this image, the numbers signify the number of fields with that type. A type ending in [] means that it is an array. The 9 different small structs were combined as their shapes are very similar. This is a very wide and very shallow shape.

5.3.2 Google

Entries in the Google dataset have the following shape:

5. EXPERIMENTS

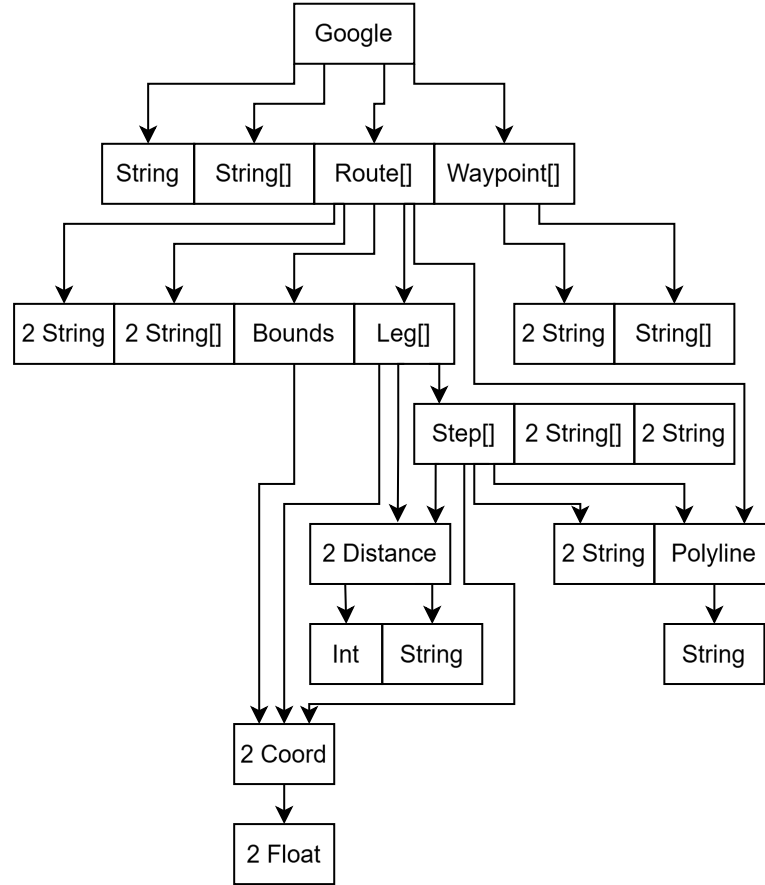


Figure 3: Bestbuy data shape.

This is a much deeper and far less wide structure. It is also no longer fully a tree, but it is still acyclic.

5.3.3 Twitter

The shape of the entries in the Twitter dataset is too convoluted to fit on a page in a readable way. To summarise it, it is even deeper than Google, and also wider. The root contains 34 fields, one of which is `RetweetedStatus`, which itself contains almost the same fields as the root, except the `RetweetedStatus` field is replaced with a `QuotedStatus`, which again looks almost the same, but this time without a self-similar field. Note that despite these similarities, there is no cycle, so depth is still bounded.

5.3.4 Walmart

Entries in the Walmart dataset have the following shape:

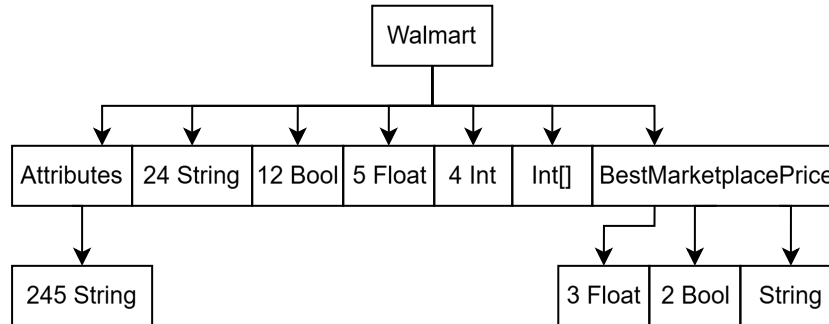


Figure 4: Bestbuy data shape.

This is a relatively simple tree, but the `Attributes` struct has 245 String fields! However, looking at the actual data, almost all of these are always set to null. 233,374 entries have no attributes, 31,967 have 1, 6,434 have 2, and the other 724 have 3. This means that any protocol or implementation that deals poorly with null values (especially strings) is going to have a hard time on this dataset

5.3.5 Wiki

The Wiki dataset entries are easily the most complicated. If it were to fit in a similar graph as the other datasets, this graph would contain cycles, meaning that there is no upper bound on depth. The root itself only has 8 fields, but some of those fields have hundreds of inner fields. One has 998 fields, a little over half of them strings, the rest arrays of other structs.

5.4 Findings

These are the results from JFR. All the following entries are formatted as “arithmetic mean $\pm$ standard deviation”, which were calculated based on four runs of the complete benchmark. For the Fory entries, the N means native mode (XLang disabled) and the X means XLang mode (XLang enabled). In these results, XLang is always better than native, so it should only be enabled for projects that require the cross-language capabilities.

For all the following figures, “Serialise” means that this was measured during serialisation, whereas “Deserialise” means they were measured during deserialisation. The error bars represent standard deviation, **not from item to item but from run to run, as explained in Section 3.2**. The only exception to this is Figure 11, where the standard deviation from run to run would always be zero, as serialising the

5. EXPERIMENTS

same object should always result in the same serialised representation (with perhaps some reordering).

The following two results were obtained using a Python script which extracts the results from the Java Flight Recorder log files.

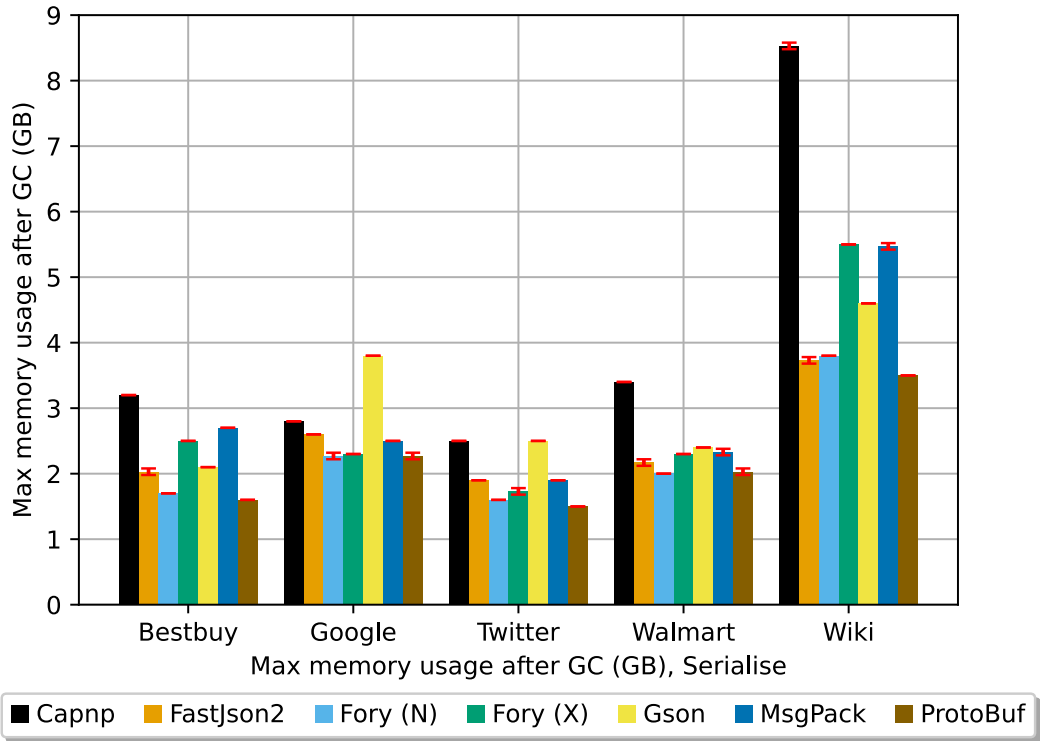


Figure 5: Max memory usage after GC (GB), Serialise (error bars are std dev).

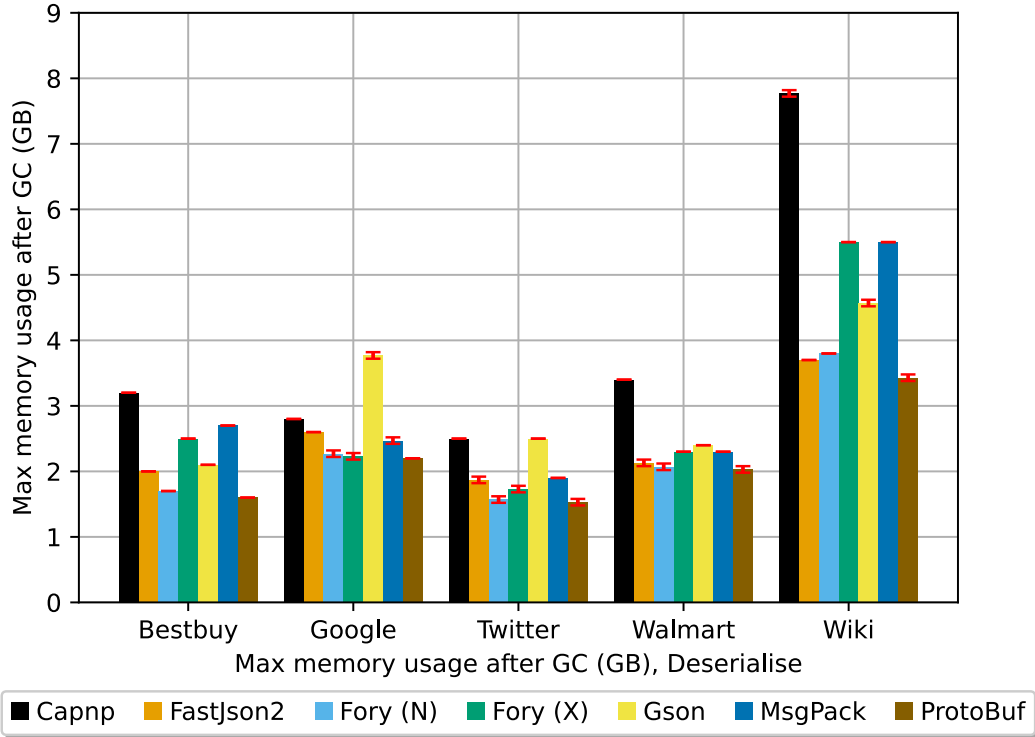


Figure 6: Max memory usage after GC (GB), Deserialise (error bars are std dev).

Figure 5 and Figure 6 show memory usage right after garbage collection, as this represents the memory that is actually being used. In garbage-collected programming languages like Java, allocations are never freed, but instead accumulate over time. At certain points in the process, the garbage collector steps in to free those objects that it finds are no longer in use.

Memory usage appears to be quite symmetric for these libraries, as the serialise and deserialise graphs appear almost identical.

The large entries in the Google dataset appear to have an especially large impact on memory usage in Gson. The recursive structure and many different struct definitions of the Wiki dataset appear to have a large impact on Capnp. For this dataset, Fory also uses a lot less memory in Native mode than XLang mode, going from 5.50 GB to 3.80 GB.

5. EXPERIMENTS

The following results were obtained from JMH. Note that for Cap'n Proto (Capnp), these results assume that every field is actually read during deserialisation. In a real application, this might not be the case. If an application frequently reads only part of the received object, the performance of Cap'n Proto will almost certainly be better than shown here.

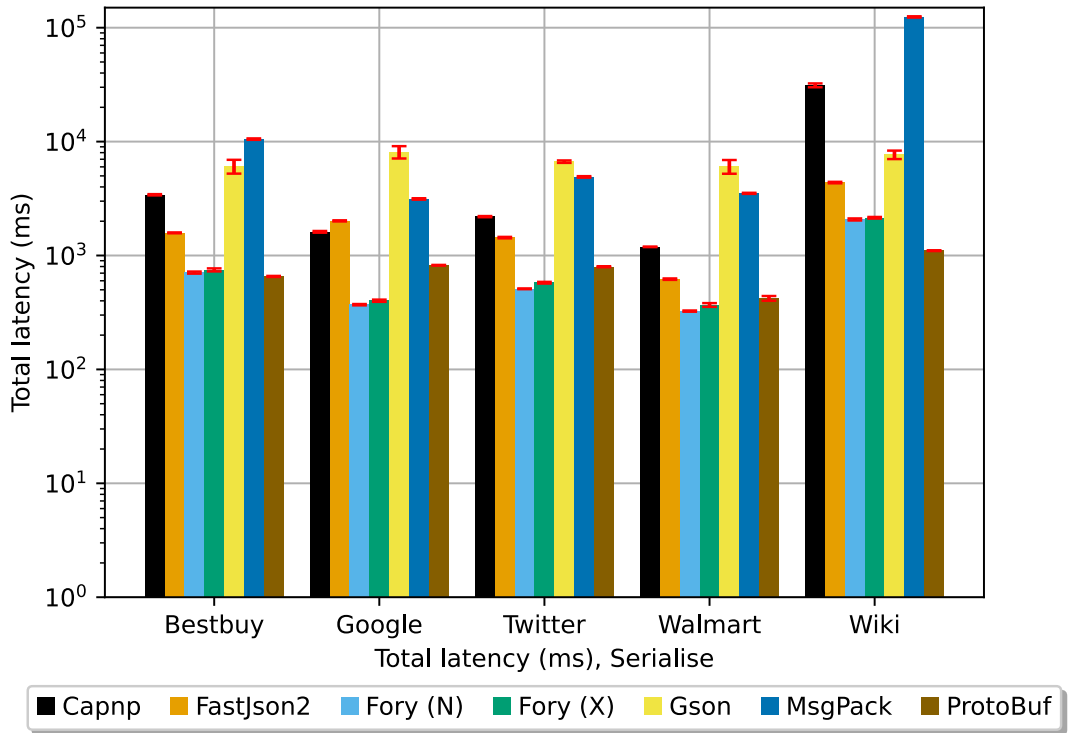


Figure 7: Total latency (ms), Serialise (error bars are std dev).

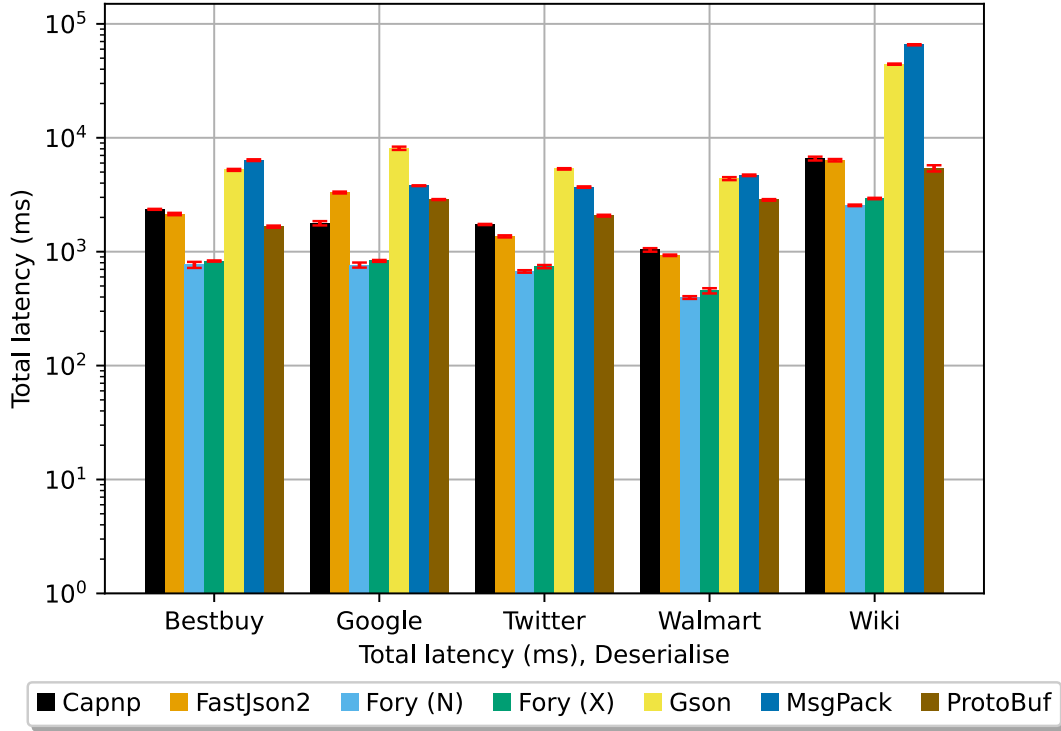


Figure 8: Total latency (ms), Deserialise (error bars are std dev).

Figure 7 and Figure 8 show that latency is much less symmetric, with there being significant differences between serialising and deserialising.

For each of the tested datasets, Fory’s Native mode was faster than XLang mode. The largest relative difference between the two modes is 13% for the Walmart dataset. The smallest is 3% for the Wiki dataset. This means that there is frequently some performance to be gained by enabling Native mode. However, native mode is only available if both serialisation and deserialisation is done in the same programming language, and a full rewrite of one side of a system is probably not worth even the 13% (de)serialisation performance gain.

Fory did very well in general, coming in first place even in XLang mode for all datasets when deserialising. However, when serialising entries in the Bestbuy and Wiki datasets, ProtoBuf had the lowest latency. For the Wiki dataset, ProtoBuf was 88% faster than Fory. For the Bestbuy dataset, this difference was only 8%. The most likely culprit

5. EXPERIMENTS

for this difference is the recursive nature of the Wiki dataset, as this is the main difference between the Wiki and Twitter datasets.

Comparing these results to those found in related work, Jackson et al. [22] concluded that in their Python benchmark, MessagePack and JSON had very similar latency in all cases. However, in their results, it is shown that ProtoBuf is sometimes a little faster than JSON, and sometimes it was 10 times slower. As stated in Section 2.4, it is unclear which Python libraries they used, but in our results ProtoStuff was consistently faster than either Gson or FastJson2.

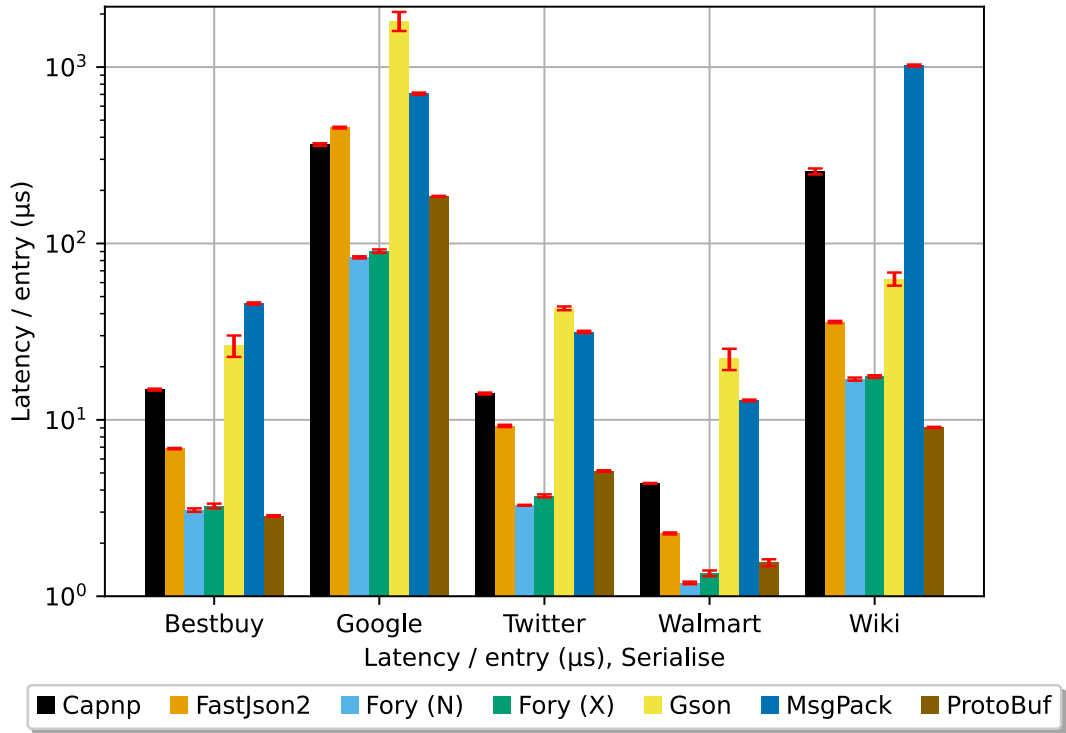


Figure 9: Latency/entry (μ s), Serialise (error bars are std dev).

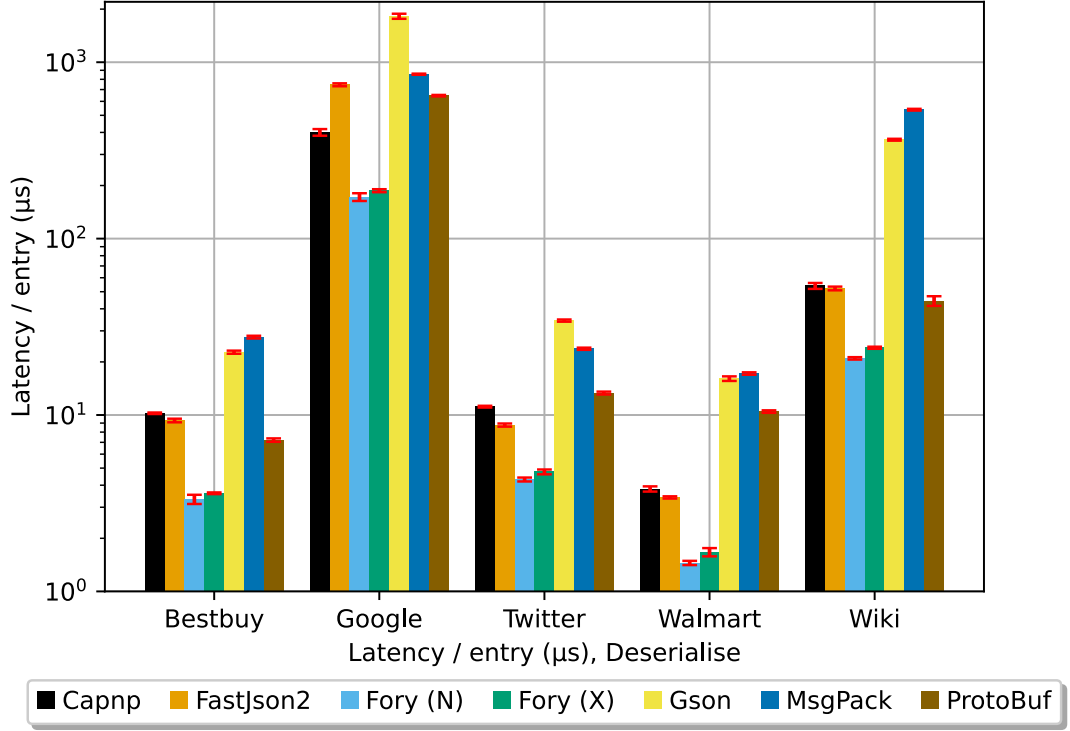


Figure 10: Latency/entry (μs), Deserialise (error bars are std dev).

Figure 9 and Figure 10 were generated by dividing the total latency for each dataset by the number of entries in that dataset, which can be found in Table 2. As such, they have the same relative results as Figure 7 and Figure 8. These graphs are included so developers can get a rough estimate of their potential performance by seeing which dataset aligns most with their own data.

5. EXPERIMENTS

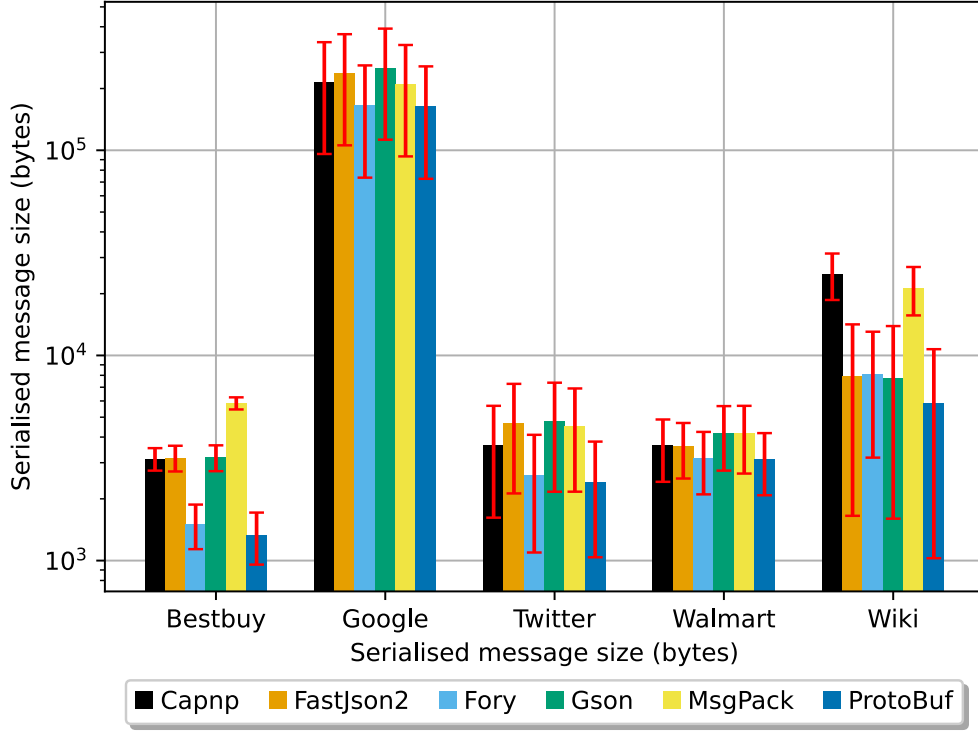


Figure 11: Serialised message size (bytes) (error bars are std dev).

Figure 11 is the only graph where the standard deviation is not from run to run but from dataset entry to dataset entry. These deviations are large as the serialised size depends heavily on the size of the object being serialised.

Comparing these results to those found in related work, Viotti et al. [27] found MessagePack was on average 22.8% more compact. Our findings paint a different picture. It is possible the Java implementation of MessagePack creates less efficient messages than the library used in their research, or it is possible that their datasets were all sufficiently different from the ones used in this thesis

5.5 Limitations

This benchmark was done on only real-life datasets. As these datasets are very different, it is difficult to determine which attributes of the data shape are responsible for the changes in relative performance. As discussed in Section 6.2, research with synthetic datasets might give deeper insight into the causes, and would allow for stronger conclusions to be drawn.

6

Conclusion

From the data in Section 5, it appears that the relative performance of the implementations of different serialisation protocols does indeed vary quite significantly with the data shape. And these results are not based on hypothetical, highly unrealistic, extreme shapes, but on real-world datasets. This means that before taking any benchmark as gospel and picking a protocol purely based on that benchmark, one should always consider if the shape of the data from the benchmark matches the shape of the data they will be (de)serialising. Apache Fory can be twice as fast as Google's Protocol Buffers, or it could be twice as slow. All with the same implementation, environment, and configuration. This also means that it is quite easy to cherry-pick benchmark data which shows a very biased result, while looking entirely fair. It is even entirely possible for this exact situation to happen accidentally. Let's say a company determines that the serialisation protocol they are using isn't fast enough, and they decide to develop their own. These developers would likely look at their own datasets to determine how to optimise their code. They would run a benchmark, tweak their code, then run the benchmark again. This is, of course, not a bad thing. In the end, this company will have saved a lot of hardware utilisation. But it becomes a problem when they publish this benchmark claiming an 800% speedup, based on the exact data shape that the protocol was optimised for.

To find what protocol and implementation should be used for a specific project, the best approach is for the developer to run their own benchmark using data that matches their use case. This can be done using our extensible benchmarking framework. Failing that, they could read the descriptions of the datasets in Section 5.3 and see which set matches the project's data the best.

For these specific datasets and serialisation libraries, we can draw the following conclusions:

6. CONCLUSION

It appears that Apache Fory has very low latency in most cases, and the difference between XLang and native mode is usually quite small. However, in both the Bestbuy and the Wiki dataset, it was outperformed by Protocol Buffers. The difference was the most significant for the Wiki dataset, where ProtoBuf was more than twice as fast. This was the only dataset with a recursive data type. It appears that ProtoBuf was the only one to handle this dataset well. This suggests that the ProtoStuff library is a good choice for recursive datasets. The fact that ProtoBuf also does well on the Bestbuy dataset suggests that it is also a good pick for serialising small and simple structures. Note that this is only the case for serialisation. On the deserialisation side, Fory was the fastest for all 5 datasets. This does not mean that it is the fastest for any possible data shape, but it is a good sign. However, deserialisation latency may be much less important in some projects.

Gson seems to scale very poorly with the size of the entry, both serialising and deserialising, based on its score for the Google dataset.

6.1 Threats to validity

Please note that this thesis is not claiming that these results will hold for all implementations of the same protocols. As shown, there is a large difference between the performance of GSON and FASTJSON2, which both implement the JSON protocol. A similar experiment in another programming language than Java would likely have different results, as that experiment would use other implementations. Even our conclusions about which data shapes have performance impacts on the implementation might not hold for other implementations of the same protocol. Some things are inherent to the protocol, like JSON representing booleans with `true` or `false`. This protocol-level design decision means that the density of booleans in the data will likely have a performance impact on any JSON implementations when compared to other protocols. However, many other decisions that determine what data shape will work best for an implementation are at the implementation level. The conclusions in this thesis regarding which serialisation library is best for certain data shapes are only valid for those specific libraries and those specific versions, which are specified in Table 1. However, the conclusion that data shape has a highly significant impact on relative serialisation performance is likely generalisable to other implementations in other

programming languages, although this is something that would need to be tested empirically, as stated in Section 6.2.

As explained in Section 3.1.3, for Cap'n Proto we are not really measuring the serialisation or deserialisation steps, as those don't exist in zero-copy protocols like Cap'n Proto. This means we are fundamentally measuring different things, as in a real production environment it is very common to first write to an object and then serialise this object. For a non-zero-copy protocol, these are two separate steps, where our measurements only reflect the second part. For a zero-copy protocol, this happens in a single step, resulting in better performance. Another benefit of zero-copy, which was already highlighted in Section 3.1.3, is that during deserialisation, many real-life applications do not read every field, but only those fields that are needed. With non-zero-copy protocols, everything is deserialised regardless of what is actually required. Zero-copy protocols, on the other hand, are able to only read those items that the developer requests.

Lastly, in Section 5, the results are shown in total latency and latency/entry. The reason there is no throughput graph is that throughput would have to be measured in either entry/time or bytes/time. Entry/time would just be the inverse of the latency/entry graph. Bytes/time would not be appropriate. How many bytes is an entry? If we use the size in the original JSON-encoded dataset, a boolean is four or five bytes, strings are two extra bytes because of the quotes surrounding them, etc. If we use the size as encoded in the protocol being tested, then protocols that are space-inefficient would have an advantage in throughput.

6.2 Future Work

While this thesis provides insight into how data shape affects the relative performance of serialisation protocols, there is still more to be researched.

First, the benchmark could be extended to cover a broader range of serialisation protocols and implementations, especially for programming languages other than just Java. The present study focused on a limited selection, and including additional protocols, particularly those with fundamentally different design philosophies, would help determine whether similar results are observed for other protocols and implementations. Testing for other languages is especially important, as this thesis can only give advice for projects implemented in Java.

6. CONCLUSION

Second, future work could use custom dataset generators that include more fine-grained and controlled differences in data shape. A more fine-grained analysis of how performance fluctuates when specific properties of the data change could give a more detailed guide on which serialisation library to use for a project.

Whereas this initial thesis deliberately used real-world data to demonstrate that data shape has a measurable impact on serialisation performance in practical, real-life use cases, such data does not allow individual structural properties to be isolated. Synthetic datasets that vary systematically along a single dimension at a time, for example nesting depth, field count, array/string size, or value type, would make it possible to pinpoint more precisely which specific characteristics of a data shape cause decreased performance in certain protocols or libraries.

References

- [1] 2021. Retrieved from <https://www.datacenterdynamics.com/en/news/microsoft-amazon-and-google-operate-half-the-worlds-600-hyperscale-data-centers/>
- [2] 2025. Retrieved from <https://github.com/bytedance/sonic/blob/main/docs/INTRODUCTION.md>
- [3] 2026. Retrieved from <https://www.bytedance.com/en/>
- [4] 2026. Retrieved from <https://stackoverflow.com/search?q=fastest+serialization>
- [5] 2026. Retrieved from https://github.com/alibaba/fastjson2/wiki/fastjson_benchmark
- [6] 2026. Retrieved from <https://flatbuffers.dev/benchmarks/>
- [7] 2026. Retrieved from <https://github.com/ndsev/zserio-protobuf-benchmarks>
- [8] 2026. Retrieved from <https://www.iea.org/energy-system/buildings/data-centres-and-data-transmission-networks>
- [9] 2026. Retrieved from <https://beeceptor.com/docs/concepts/data-exchange-formats/>
- [10] 2026. Retrieved from <https://www.json.org/json-en.html>
- [11] 2026. Retrieved from <https://protobuf.dev/history/>
- [12] 2026. Retrieved from <https://protobuf.dev/overview/>
- [13] 2026. Retrieved from https://fory.apache.org/blog/fury_blazing_fast_multiple_language_serialization_framework/
- [14] 2026. Retrieved from <https://www.ibm.com/docs/en/sdk-java-technology/8?topic=reference-jit-compiler>
- [15] 2026. Retrieved from <https://mvnrepository.com/artifact/org.apache.fory/fory-core>

6.2. REFERENCES

- [16] 2026. Retrieved from <https://msgpack.org/>
- [17] 2026. Retrieved from <https://capnproto.org/>
- [18] 2026. Retrieved from <https://math.hws.edu/eck/cs124/javanotes3/java2cpp/s2.html>
- [19] 2026. Retrieved from <https://mvnrepository.com/artifact/org.simdjson/simdjson-java>
- [20] 2026. Retrieved from <https://mvnrepository.com/open-source/json-libraries>
- [21] S. Eyerman, J.E. Smith, and L. Eeckhout. 2006. Characterizing the branch misprediction penalty. In *2006 IEEE International Symposium on Performance Analysis of Systems and Software*, 2006. 48–58. <https://doi.org/10.1109/ISPASS.2006.1620789>
- [22] Samuel Jackson, Nathan Cummings, and Saiful Khan. 2024. Streaming Technologies and Serialization Protocols: Empirical Performance Analysis. Retrieved from <https://arxiv.org/abs/2407.13494>
- [23] Sagar Karandikar, Chris Leary, Chris Kennelly, Jerry Zhao, Dinesh Parimi, Borivoje Nikolic, Krste Asanovic, and Parthasarathy Ranganathan. 2021. A Hardware Accelerator for Protocol Buffers. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21)*, 2021. Association for Computing Machinery, Virtual Event, Greece, 462–478. <https://doi.org/10.1145/3466752.3480051>
- [24] Geoff Langdale and Daniel Lemire. 2019. Parsing gigabytes of JSON per second. *The VLDB Journal* 28, 6 (October 2019), 941–960. <https://doi.org/10.1007/s00778-019-00578-5>
- [25] Kazuaki Maeda. 2012. 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)* (2012), 177–182. Retrieved from <https://api.semanticscholar.org/CorpusID:6729040>
- [26] Jan Vanura and Pavel Kriz. 2018. Performance Evaluation of Java, JavaScript and PHP Serialization Libraries for XML, JSON and Binary Formats. In *Services Computing – SCC 2018*, 2018. Springer International Publishing, Cham, 166–175.

- [27] Juan Cruz Viotti and Mital Kinderkhedia. 2022. A Benchmark of JSON-compatible Binary Serialization Specifications. Retrieved from <https://arxiv.org/abs/2201.03051>
- [28] Juan Cruz Viotti and Mital Kinderkhedia. 2022. A Survey of JSON-compatible Binary Serialization Specifications. Retrieved from <https://arxiv.org/abs/2201.02089>
- [29] Juan Cruz Viotti and Mital Kinderkhedia. 2022. Benchmarking JSON BinPack. Retrieved from <https://arxiv.org/abs/2211.12799>

Appendix A: Benchmarking framework

A.1 Abstract

The artefact contains the benchmarking framework described in Section 4 and used in Section 5. This framework contains the script for generating Java classes from the datasets, but this script has already been run for the provided datasets. A link to the datasets used in this thesis is also given here.

A.2 Artefact check-list (meta-information)

- Program: benchmark (<https://github.com/FBastiaan04/bsctest>) & dataset (<https://drive.google.com/drive/folders/1jP1VB7-ScjNPwucivnh7JlBzwWRYyczN>)
- How much time is needed to prepare workflow (approximately)?: 15 minutes
- How much time is needed to complete experiments (approximately)?: 5 hours
- Publicly available?: The benchmarking framework is publicly available via the GitHub repository
- Code licenses (if publicly available)?: MIT License

A.3 Description

A.3.1 Hardware dependencies

The hardware is expected to have at least 16 GiB of RAM.

A.3.2 Software dependencies

The benchmark is designed to be run on any platform that supports Java. It requires JDK 21 and Apache Maven to compile and run.

A.4 Experiment workflow and customisation

After downloading it from GitHub, the framework can be compiled using maven. The `Datasets.zip` file must be downloaded from Google Drive and put in the root

directory of the git project. It should **not** be unzipped. The benchmark can then be run using java.

New datasets can be added to the zip file. If they are in the same format (entries in JSON format with one entry per line in the file). If they have a different format, code must be added to read that format in the `/src/main/java/eu/hippix/bsctest/FileReader.java` and `/j2j.py` files. The basic requirement is that all entries in the new dataset share the same data shape.

New serialisation libraries can be added as follows:

1. `/src/main/java/eu/hippix/bsctest/Benchmarks/EmptyBenchmark.java`.
2. Change the `Object` in `AbstractBenchmark<Object>` to the Java type the library serialises to (usually `byte[]` or `String`).
3. Fill in the `serializeItem`, `deserializeItem` and `serialisedSize` methods.
4. Add the new class to the `benchmarks` array in `/src/main/java/eu/hippix/bsctest/App.java` on line 15.
5. Recompile the framework.

A.5 Evaluation and expected results

The relative results of running this framework should be similar to the ones in Section 5. The absolute numbers will certainly depend on the hardware used.

Appendix B: Results

Table 4: Max memory usage after GC (GB), Serialise.

	Bestbuy	Google	Twitter	Walmart	Wiki
Capnp	3.20 $\pm$ 0.00	2.80 $\pm$ 0.00	2.50 $\pm$ 0.00	3.40 $\pm$ 0.00	8.53 $\pm$ 0.05
FastJson2	2.03 $\pm$ 0.05	2.60 $\pm$ 0.00	1.90 $\pm$ 0.00	2.17 $\pm$ 0.05	3.73 $\pm$ 0.05
Fory (N)	1.70 $\pm$ 0.00	2.27 $\pm$ 0.05	1.60 $\pm$ 0.00	2.00 $\pm$ 0.00	3.80 $\pm$ 0.00
Fory (X)	2.50 $\pm$ 0.00	2.30 $\pm$ 0.00	1.73 $\pm$ 0.05	2.30 $\pm$ 0.00	5.50 $\pm$ 0.00
Gson	2.10 $\pm$ 0.00	3.80 $\pm$ 0.00	2.50 $\pm$ 0.00	2.40 $\pm$ 0.00	4.60 $\pm$ 0.00
MsgPack	2.70 $\pm$ 0.00	2.50 $\pm$ 0.00	1.90 $\pm$ 0.00	2.33 $\pm$ 0.05	5.47 $\pm$ 0.05
ProtoBuf	1.60 $\pm$ 0.00	2.27 $\pm$ 0.05	1.50 $\pm$ 0.00	2.03 $\pm$ 0.05	3.50 $\pm$ 0.00

Table 5: Max memory usage after GC (GB), Deserialise.

	Bestbuy	Google	Twitter	Walmart	Wiki
Capnp	3.20 $\pm$ 0.00	2.80 $\pm$ 0.00	2.50 $\pm$ 0.00	3.40 $\pm$ 0.00	7.77 $\pm$ 0.05
FastJson2	2.00 $\pm$ 0.00	2.60 $\pm$ 0.00	1.87 $\pm$ 0.05	2.13 $\pm$ 0.05	3.70 $\pm$ 0.00
Fory (N)	1.70 $\pm$ 0.00	2.27 $\pm$ 0.05	1.57 $\pm$ 0.05	2.07 $\pm$ 0.05	3.80 $\pm$ 0.00
Fory (X)	2.50 $\pm$ 0.00	2.23 $\pm$ 0.05	1.73 $\pm$ 0.05	2.30 $\pm$ 0.00	5.50 $\pm$ 0.00
Gson	2.10 $\pm$ 0.00	3.77 $\pm$ 0.05	2.50 $\pm$ 0.00	2.40 $\pm$ 0.00	4.57 $\pm$ 0.05
MsgPack	2.70 $\pm$ 0.00	2.47 $\pm$ 0.05	1.90 $\pm$ 0.00	2.30 $\pm$ 0.00	5.50 $\pm$ 0.00
ProtoBuf	1.60 $\pm$ 0.00	2.20 $\pm$ 0.00	1.53 $\pm$ 0.05	2.03 $\pm$ 0.05	3.43 $\pm$ 0.05

The following results are from JMH (not JFR).

Table 6: Total latency (ms), Serialise.

	Bestbuy	Google	Twitter	Walmart	Wiki
Capnp	3414 $\pm$ 44	1614 $\pm$ 28	2194 $\pm$ 26	1191 $\pm$ 6	31211 $\pm$ 1218
FastJson2	1581 $\pm$ 14	2014 $\pm$ 23	1436 $\pm$ 21	620 $\pm$ 9	4369 $\pm$ 65
Fory (N)	708 $\pm$ 15	370 $\pm$ 5	510 $\pm$ 4	325 $\pm$ 4	2073 $\pm$ 43
Fory (X)	748 $\pm$ 23	401 $\pm$ 9	577 $\pm$ 10	368 $\pm$ 14	2145 $\pm$ 35
Gson	6077 $\pm$ 843	8114 $\pm$ 1006	6669 $\pm$ 165	6057 $\pm$ 836	7679 $\pm$ 656
MsgPack	10516 $\pm$ 153	3137 $\pm$ 45	4900 $\pm$ 73	3513 $\pm$ 39	124354 $\pm$ 1579
ProtoBuf	655 $\pm$ 7	821 $\pm$ 6	797 $\pm$ 8	421 $\pm$ 20	1103 $\pm$ 9

Table 7: Total latency (ms), Deserialise.

	Bestbuy	Google	Twitter	Walmart	Wiki
Capnp	2354 $\pm$ 23	1778 $\pm$ 77	1733 $\pm$ 20	1037 $\pm$ 37	6572 $\pm$ 255
FastJson2	2142 $\pm$ 51	3307 $\pm$ 63	1363 $\pm$ 29	930 $\pm$ 14	6355 $\pm$ 154
Fory (N)	767 $\pm$ 47	764 $\pm$ 39	671 $\pm$ 17	396 $\pm$ 11	2552 $\pm$ 40
Fory (X)	828 $\pm$ 10	830 $\pm$ 16	740 $\pm$ 24	454 $\pm$ 24	2930 $\pm$ 41
Gson	5226 $\pm$ 105	8092 $\pm$ 268	5333 $\pm$ 77	4382 $\pm$ 132	44289 $\pm$ 543
MsgPack	6362 $\pm$ 112	3796 $\pm$ 29	3692 $\pm$ 52	4684 $\pm$ 73	65543 $\pm$ 749
ProtoBuf	1659 $\pm$ 36	2869 $\pm$ 28	2071 $\pm$ 39	2849 $\pm$ 46	5402 $\pm$ 340

2. RESULTS

Table 8: Latency / entry (μs), Serialise.

	Bestbuy	Google	Twitter	Walmart	Wiki
Capnp	14.84 ± 0.19	363.70 ± 6.22	14.11 ± 0.17	4.37 ± 0.02	256.24 ± 10.00
FastJson2	6.87 ± 0.06	453.73 ± 5.29	9.24 ± 0.14	2.27 ± 0.03	35.87 ± 0.53
Fory (N)	3.08 ± 0.07	83.42 ± 1.17	3.28 ± 0.02	1.19 ± 0.02	17.02 ± 0.35
Fory (X)	3.25 ± 0.10	90.44 ± 2.07	3.71 ± 0.07	1.35 ± 0.05	17.61 ± 0.29
Gson	26.41 ± 3.66	1827.83 ± 226.58	42.89 ± 1.06	22.23 ± 3.07	63.04 ± 5.39
MsgPack	45.70 ± 0.66	706.72 ± 10.10	31.52 ± 0.47	12.89 ± 0.14	1020.92 ± 12.97
ProtoBuf	2.85 ± 0.03	184.96 ± 1.24	5.13 ± 0.05	1.55 ± 0.07	9.06 ± 0.07

Table 9: Latency / entry (μs), Deserialise.

	Bestbuy	Google	Twitter	Walmart	Wiki
Capnp	10.23 ± 0.10	400.47 ± 17.36	11.15 ± 0.13	3.81 ± 0.13	53.96 ± 2.09
FastJson2	9.31 ± 0.22	744.99 ± 14.30	8.77 ± 0.18	3.41 ± 0.05	52.18 ± 1.26
Fory (N)	3.33 ± 0.20	172.13 ± 8.71	4.31 ± 0.11	1.45 ± 0.04	20.95 ± 0.33
Fory (X)	3.60 ± 0.04	186.95 ± 3.64	4.76 ± 0.15	1.67 ± 0.09	24.06 ± 0.33
Gson	22.71 ± 0.46	1822.97 ± 60.36	34.30 ± 0.49	16.08 ± 0.48	363.61 ± 4.46
MsgPack	27.65 ± 0.49	855.06 ± 6.46	23.75 ± 0.34	17.19 ± 0.27	538.10 ± 6.15
ProtoBuf	7.21 ± 0.16	646.34 ± 6.25	13.32 ± 0.25	10.46 ± 0.17	44.35 ± 2.79

Table 10: Message size (bytes).

	Bestbuy	Google	Twitter	Walmart	Wiki
FastJson	3174	236870	4697	3599	7912
MsgPack	5851	209702	4535	4168	21328
ProtoBuf	1335	164529	2420	3132	5877
Fory (X)	5171	166805	3780	4043	21774
Fory (N)	1506	166399	2599	3171	8113
Capnp	3139	216197	3650	3647	25010
GSON	3187	252190	4765	4204	7753