Report 1

Systems Integration
Data Representation and Serialization Formats

UNIVERSIDADE D
COIMBRA

Alexy Almeida	2019192123	adenis@student.dei.uc.pt
Edgar Duarte	2019216077	edgarduarte@student.dei.uc.pt

PL2

September 2022

Contents

1 Introduction 1
2 Data structures 1
3 Experimentation 2
3.1 Setup 2
3.2 Results 3
4 Discussion 8
5 Conclusion 9

1 Introduction

This report aims to demonstrate how text and binary formats for data serialization work, what are the differences between the most important formats and what trade-offs exist between them. Results will be provided and a comparison will be made between different data representation technologies, as well as compare their size and encoding speed. The options for data representation are XML, XML compressed with Gzip and Google Protocol Buffers.

2 Data structures

The main data structure that was used is a many-to-one relationship between two java objects, *Student* and *Professor*.

- The first one contains the following attributes: id, name, telephone, address, birth date, registration date and gender;
- The second one contains the following attributes: id, name, telephone, address, birth date and a list of *Student*.

Two other java objects are created, *ClassT* and *ClassTProto*.

- The first one contains a list of *Professor*;
- The second one contains two lists of *Student* and *Professor*

In order to serialize with protocol buffers, a .proto file *AppProto* was created, which contains three messages:

- *Student* contains the following attributes: id, name, telephone, address, birth date, registration date, gender and teacher. All of those attributes are *optional string* except teacher, which is *optional Teacher*;
- *Professor* contains the following attributes: id, name, telephone, address, birth date and students. All of those attributes are *option string* except students, which is *repeated Student*.

3 Experimentation

In order to understand the benefits and drawbacks of using XML, XML with Gzip and Protocol Buffers we need to see how those perform when serializing and deserializing java objects.

To compare each technology, the focus will be on the speed and size of each serialization procedure for every data format. But what exactly can contribute to those results? Is the number of professors more impactful than the number of students for each professor? Can the number of names of each student and professor affect the size and speed of those procedures?

3.1 Setup

When running the program, the user is asked to input a number of professors, students, number of names per student and the number of experiments. The program then proceeds to serialize & deserialize XML, serialize & deserialize XML compressed with Gzip and, finally, serialize & deserialize with Google Protocol Buffers.

To analyze the impact that the different parameters have on the performance of each text format, 4 different tests were conducted. In order to have a good sample size, each test must run 30 times. In each test a different parameter is incremented as followed:

- 1st test: the number of professors increments;
- 2nd test: the number of students increments;
- 3rd test: number of names increments;
- 4th test: every parameter increments;

Those tests were conducted on a Legion Y530-15ich, 8th generation, with 16.0 gigabytes of RAM computer, running Windows 10.

When serializing, the time for XML and GZIP starts counting right before marshaling the Java Objects and ends when the file is written, while for Google Protocol Buffers it starts and ends at the beginning and end of the file writing process. When deserializing for all the data formats, the time starts counting when the encoded file is opened and ends when the corresponding Java Object is created.

The results for each data format are presented in the next three subsections.

3.2 Results

1st test - Professors increment

This test checks the performance of the three data formats when incrementing the number of professors parameter. The number of students is fixed at 5 for each professor and every student has 3 names.

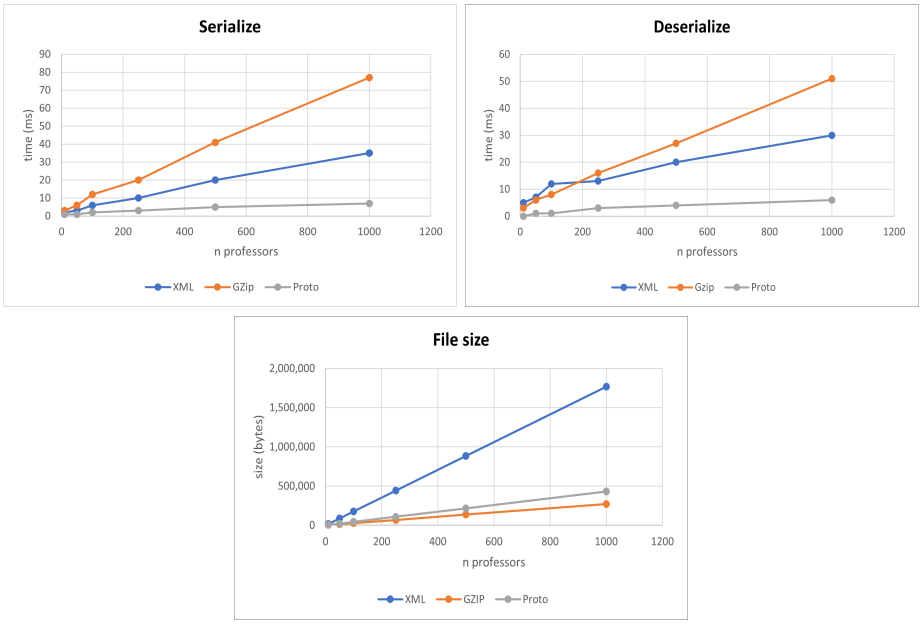

Figure 1: Performance of the data formats for different numbers of professors

The upper plots represent the time that each data format takes to serialize and deserialize, respectively, in relation to the amount of professors. The bottom plot represents the size of the file created by serializing with each of the formats.

As shown in [1], XML compression with Gzip is by far the slowest format. On the other hand, it is also the data format that creates the most compressed file. The Google Protocol Buffer format outperforms standard XML in both speed and file compression by a fair margin.

As for the impact that adding professors had on the overall performance of the data formats, it is clear that there is a linear relationship between the number of professors and time spent creating the file and size of said file.

The following table shows the standard deviation of this test for each data format in each operation:

	Serialize	Deserialize	File size
XML	12,73839341	9,224966125	669419,4909
Gzip	28,23295946	18,11905075	102797,0317
Proto	2,401388487	2,258317958	163986,9741

Table 1: Standard deviation of test 1

Through this table we can conclude that the most stable format in serializing and deserializing files is Google Protocol Buffers, while Gzip is the format with the most deviation, meaning it is more unstable. Meanwhile, XML's performance is average, comparing to the other formats.

2nd test - Students increment

This test checks the impact that incrementing the number of students parameter has on the different data formats performance. The number of professors is fixed at 100 and the number of names of every student is fixed at 3.

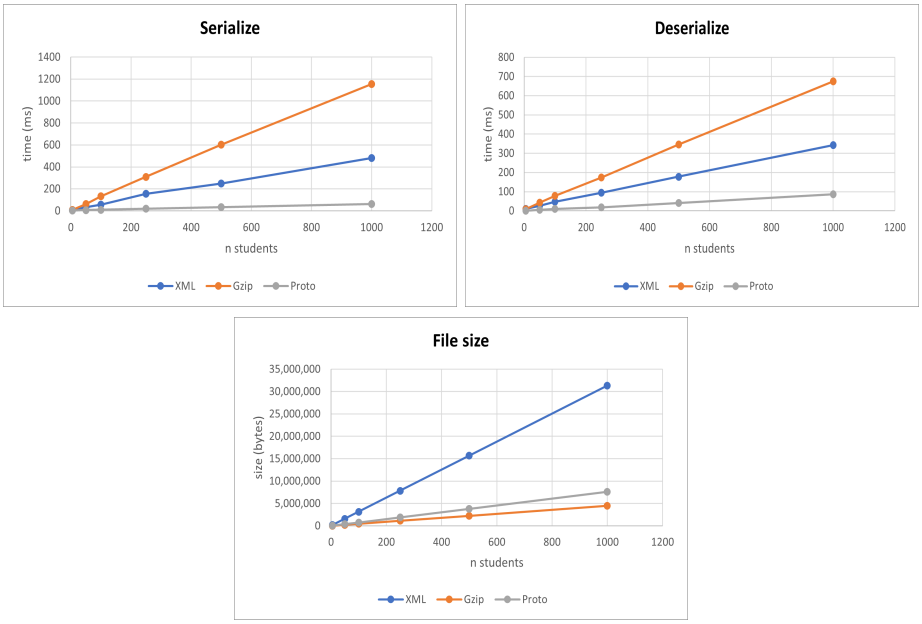

Figure 2: data formats performances with serialize and deserialize

Just as seen in [1], XML with GZIP compression is the slowest but most compressing format. On the other hand, Google Protocols Buffer format beats standard XML in both speed and compression.

The impact that the student parameter has on the performance of the algorithms is very similar to the professor parameter. There is a linear relationship between the number of students and the performance of formats, as can be

clearly seen in [2]. However, it takes around **29x more time** to serialize and **13x more time** to deserialize, as well as creating a file that's approximately **18 times bigger**.

We can then assume that **incrementing the number of students, i.e, the number of objects inside an object, will generate bigger values in both speed serializing and the resulting file size.**

Once again, the following table shows the standard deviation of this test for each data format in each operation:

	Serialize	**Deserialize**	**File size**
XML	180,0877564	126,095863	11872813,31
Gzip	436,8806473	253,4567024	1700765,013
Proto	22,35546167	32,21800739	2876316,744

Table 2: Standard deviation of test 2

Here, we can see a clear relation with table [1] where Google Protocol Bufers continues to be the most stable format in serializing and deserializing, while also making Gzip the most unstable format and XML the most average of the three of them.

3rd test - Number of student names increment

So far, the results show that adding students or professors has a linear impact on the performance of the data formats. This test checks if changing the parameters inside of the students and professors has any considerable impact. The number of professors is fixed at 100 while the number of students is fixed at 20.

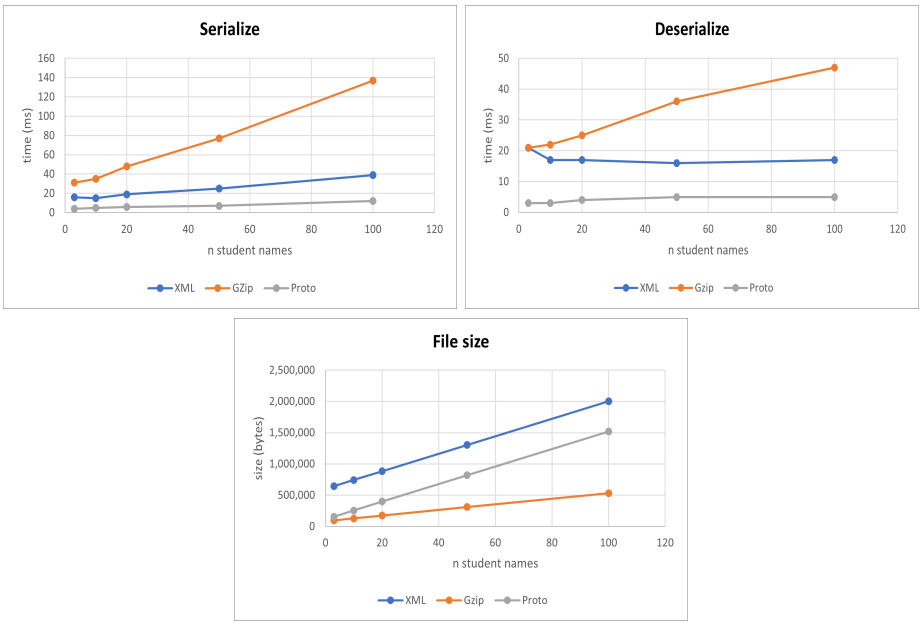

Figure 3: data formats performances with serialize and deserialize

The results from this test differ from the previous ones. In terms of speed spent serializing and deserializing, XML with Gzip shows an identical behaviour, as seen in [1] and [2]. However, the other two data formats show different behaviours towards this parameter, presenting more constant values.

On the other hand, the file size resulting from Google Protocol Buffers and XML with Gzip compressions shows different performance types compared to [1] and [2]. Even though XML with Gzip compresses the most efficiently, there's a bigger gap between its performance and the Protocol Buffers one, the latter being even more inefficient than before.

With this, we can assume that **using XML with Gzip compression is efficient when dealing with java objects containing more parameters.**

The same table is presented:

	Serialize	**Deserialize**	**File size**
XML	9,859006035	1,949358869	555872,1508
Gzip	43,79269345	11,12205017	178367,677
Proto	3,1144823	1	557531,8019

As expected, the standard deviation remains similar for every data format in terms of serialize and deserialize speeds, as seen in [1] and [2], even though it

remains pretty low for the latter. However, for XML and Protocol Buffers the value remains pretty close for both of them, making XML with Gzip stand out with a huge gap. As assumed before, XML with Gzip is the most compressing format for this type of situations.

4th test - Every parameter increment

Lastly, the final test was made by raising every parameter. From the first to the second iteration, the number of professors, students and name students was multiplied by 5. Henceforth, every parameter is multiplied by 2.

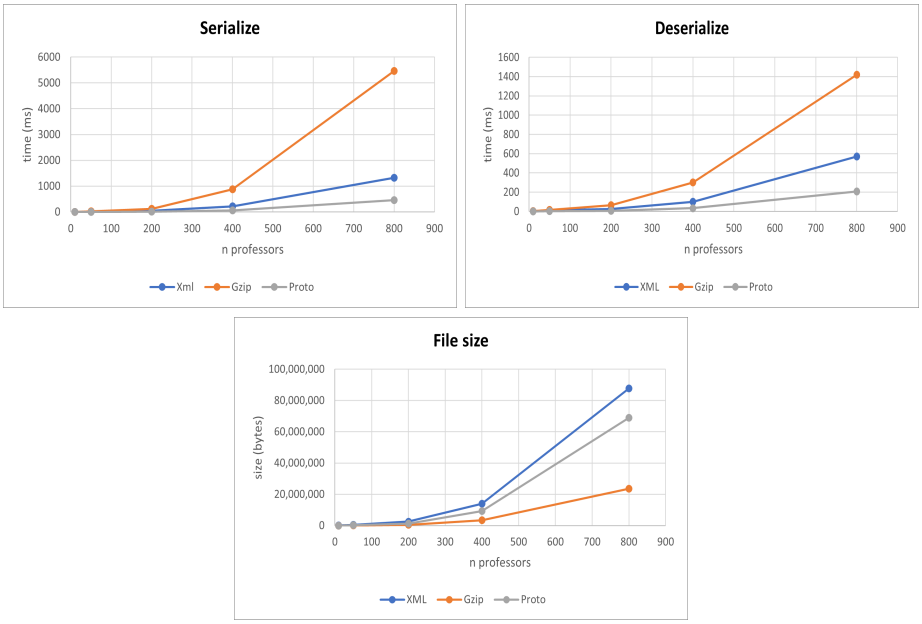

Figure 4: data formats performances with serialize and deserialize

Even though the results given are somewhat similar [1], [2] and to [3.2, both in terms of speed and file size, these last plots make the difference between each format even more clear, presenting a spike in their performance at the middle of our tests. The more parameters we increase, the more the data format performance is affected, as we could expect.

However, when dealing with a very large number of paremeters, this only emphasizes even more that XML and Protocol Buffers are preferred in terms of speed, while XML with Gzip is the most favoured for compressing and getting lower file sizes.

In [1] and [2] Protocol Buffers and XML with Gzip share a similar performance, but the truth is that **the former can become quite inefficient when we're dealing with lots of different information inside of java objects.**

Finally, the last table is presented:

	Serialize	Deserialize	File size
XML	3194,416843	964,7468062	241577323,1
Gzip	16919,73716	3724,296251	69173946,54
Proto	1148,343982	647,0689041	212190302,1

These deviation results go hand-in-hand with what has already been concluded in table [3.2].

4 Discussion

When considering all the experiments performed, the difference between each format was made clear when increasing different types of parameters inside a java object. The XML format is a pretty generic format in terms of serialize and deserialize speeds, while also being the most inefficient at compressing.

Compared to that, XML with Gzip should be highly avoided if time is of concern, being the format that takes the most time to both serialize and deserialize. Is it, however, the most reliant format for dealing with a huge variety of information.

As for the third data format, Google Protocol Buffers is an interesting data format that should be chosen very carefully, depending on the use case. If time is the biggest priority, it is the preferred format, showing the lowest values for any kind of situation. It is also an interesting choice if there's not a lot of data diversity, for example, when we want to quickly transfer little data across the network. However, for files with a bigger variety of data, if time is prioritized, this data format should be avoided.

In a network transmission context it is crucial to have small size files and to have fast serialization and deserialization operations. If you have a bad and slow network you cannot afford to pass through it big files, as there will be a lot of missing packages (that need to be resent) which leads to a very long time to transfer the entire file.

Meanwhile, if we have a fast and reliable network, bigger files can be transferred without too much of an issue, so minimizing the serialization and deserialization times could serve as profit. Due to this, we can conclude that in a setting where we do not have a very fast network system, the use of Gzip might be the preferred format, seeing that it compresses the file the most.

On the other hand, in a setting where we have a very fast network system, the use of Google Protocol Buffers might be the most indicated, as it has low serialization and deserialization times. For medium quality networks, the use of Google Protocol Buffers has, most likely, better performance than Gzip in bigger files, but we would need to do more tests to confidently affirm this.

5 Conclusion

Overall, it is considered that the goals proposed for this project were achieved. The number of tests executed were enough to show the differences between each format, however, more tests could be implemented to get an even more accurate of each data format performance. This project allowed us to understand when to use each of the three data formats presented, their drawbacks and benefits.