Mestrado em Engenharia Informática

Integração de Sistemas - Projecto 1

Leandro Pais - 2017251509
João Branco - 2021179862

8 de outubro de 2021

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE D
COIMBRA

Índice

1 Introdução

Com este trabalho, pretende-se comparar tecnologias de representação de dados. De acordo com o enunciado foi dada aos alunos a possibilidade de escolherem as tecnologias a comparar (XML ou JSON *vs. Protocol Buffers* ou *Message Pack*.

Foi decidido que o estudo comparativo recairia entre o *XML* (eXtensible Markup Language) e o *Protocol Buffers*. Para tal, foi desenvolvida, em *Java*, uma estrutura de dados comum, com relação *many-to-one*, que representa a relação *pet-owner*.

Posteriormente, foi desenvolvido um *script* em *python* capaz de fazer a geração de dados para popular as listas de *pets* e *owners*. Seguiu-se um trabalho de preparação do código para realizar os testes, esta fase inclui o cálculo do *marshalling* e *unmarshalling* das estruturas e o cálculo do espaço ocupado em memória.

A etapa seguinte, consistiu em realizar uma bateria exaustiva de testes com o objetivo de minimizar o erro na obtenção das métricas. Dos testes realizados foram obtidos os resultados evidenciados na secção 4. Por fim, após uma cuidada análise destes resultados serão apresentadas neste relatório as conclusões a retirar, sobre qual das duas tecnologias é, objetivamente, melhor.

2 Estruturas de Dados

As estruturas de dados utilizadas foram as sugeridas no enunciado, ou seja, foram utilizadas duas classes, *Pet* e *Owner* com uma relação *Many-to-One* respectivamente. A classe *Pet*, possui *Id*, *Name*, entre outras informações relevantes sobre o *pet*, tendo para cada um dos atributos os respetivos *getters* e *setters* e o construtor da classe. A classe *Owner* também contém *Id*, *Name*, entre outras informações sobre o *owner* com os *getters*, *setters* e construtores da classe. A totalidade dos atributos pode ser vista na Figura 1.

Para definir a relação entre ambas as classes existem algumas possibilidades, como por exemplo, manter uma chave estrangeira na classe *Pet* com o *Id* do respetivo *owner*. No entanto, para simplificar o grupo optou para manter na classe *Owner* uma lista com os *pets* que lhe pertencem (como se pode ver na Figura 1).

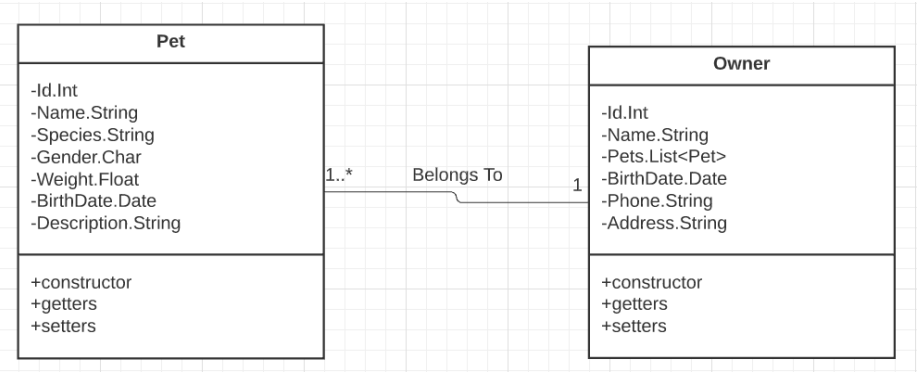

Figura 1: Estrutura de dados

3 Ferramentas utilizadas

Neste capítulo, é feita uma breve descrição de todas as tecnologias essenciais para a realização do projeto.

3.1 Maven

O Apache Maven foi utilizado neste projecto para ser a ferramenta responsável pela gestão e compilação do projecto em questão. Esta gestão é feita essencialmente realizada através de um ficheiro, *pom.xml*, onde, por exemplo, estão descritas as dependências e respectivas versões necessárias para a execução do projecto, o nome e versão do projecto, entre outros.

3.2 JAXB

Esta biblioteca, de nome *Java Architecture for XML Binding*, foi utilizada para a serialização e desserialização dos objectos Java para XML.

3.3 Eclipse

O Eclipse foi o IDE (*Integrated Development Environment*, Ambiente de Desenvolvimento) escolhido por ambos os membros do grupo para se poder desenvolver o projecto.

3.4 GitLab

Foi utilizado um gestor de repositório *online* para gerir o código dos dois membros do grupo, facilitando o trabalho dos elementos do grupo para um desenvolvimento mais eficiente.

3.5 XML

XML, *eXtensible Markup Language*, foi criado para servir como transporte e armazenamento de dados caracterizado pela fácil interpretação e leitura por parte de humanos.

3.6 Protocol Buffers

Tecnologia desenvolvida pela Google que tem como objectivo a serialização de estruturas de dados. É útil para transferência de dados pela rede ou para armazenamento de dados. Mais rápido, mais leve e simples que o XML. Para que seja possível a serialização de dados deve-se utilizador o compilador *protoc*.

4 Contexto Experimental

Como o projecto tem por base uma comparação entre diferentes tecnologias, estando a essência do mesmo relacionada com processamento de dados importa definir cuidadosamente as condições em que a experimentação é realizada e que métodos são utilizados.

4.1 Condições Experimentais

Assim sendo, são apresentadas as informações sobre a máquina onde os testes foram executados.

- *Hardware*
  - Processador: i7-7700HQ @ 2.80Ghz (8 CPUs)
  - Disco: SSD Micron_1100_MTFDDAV256TBN 256GB
  - RAM: 16GB DDR4
- *Software*
  - OS: Windows 10 Home 64-bit (Build 190439)
  - Java: java 17 2021-09-14 LTS
  - Maven: Apache Maven 3.8.2
  - JAXB: Versão 3.0.0
  - Google Protocol Buffer: Versão 3.18.0
  - IDE: Eclipse 2021-09

4.2 Método Experimental

Primeiramente, foi utilizado um *script*, desenvolvido para o efeito utilizando a linguagem *python*, no qual podemos especificar o número de *pets* e *owners* que se pretenda que os ficheiros tenham, e assim, são gerados os dados necessários para popular os objetos. Este *script* pode ser visualizado em anexo ([1]). De referir que os dados utilizados, são fictícios.

Posteriormente, para carregar estes dados para memória foi desenvolvida uma classe *Seed* que tem um método *initializeData* que é responsável por ir às diretorias onde os ficheiros de *pets* e *owners* se encontram. Depois o ficheiro *pets.in* é lido linha a linha, o objeto *pet* é criado e adicionado a uma lista de pets. Após todos os pets terem sido carregados para a memória é a vez dos *Owners* que estão sujeitos a um processo em tudo igual. De seguida, os *pets* em lista, são distribuídos, de forma aleatória, por todos os *owners* garantindo assim a relação *Many-to-One*. Por fim, a lista de *owners* já com os pets é associada uma classe *Root* que contém apenas a lista final. Esta classe final com a lista é a que vai ser objeto de serialização recorrendo às tecnologias mencionadas.

Para a realização dos testes em si, para agilizar o processo optou-se por ter um ciclo *for* a percorrer os ficheiros de *pets*, um outro encadeado para percorrer

os ficheiros de *owners* e um terceiro, para repetir a ação um número de vezes para garantir a fiabilidade dos dados obtidos. A ação é semelhante para ambas as tecnologias e consiste em utilizar o método *System.currentTimeMillis()*, e guardando o valor retornado, imediatamente antes de ser chamado o método que serializa os dados e novamente assim que método terminar e depois é feito o cálculo do tempo que este demorou e algo semelhante é feito para a desserialização. Na Figura 2, temos o exemplo para a serialização/desserialização do *XML*.

```
Marshaller marshaller = Marshal.doMarshal();
try {
    marshaller.marshal(root, new FileOutputStream("./root.xml"));
}
catch(Exception e) {
    e.printStackTrace();
}
long end = System.currentTimeMillis();

writerTimeMarshalling.println((end-start));

start = System.currentTimeMillis();

Unmarshaller unmarshaller = Marshal.unMarshal();
try {
    InputStream inStream = new FileInputStream( "root.xml" );
    Root rootUnmarshalled = (Root) unmarshaller.unmarshal( inStream );
} catch (JAXBException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}

end = System.currentTimeMillis();
writerTimeUnMarshalling.println((end-start));
```

Figura 2: Cálculo do tamanho dos dados serializados

No que toca ao tamanho dos ficheiros, foi apenas utilizado um método do Java (*Files.size()*) para obter o tamanho (em kB) dos ficheiros após serialização (como se pode ver na Figura 3). Estes dados são também exportados para fazer a respetiva análise.

```
Path path = Paths.get("root.xml");
long bytes = Files.size(path);
```

Figura 3: Cálculo do tamanho dos dados serializados

O número de testes que realizámos para cada conjunto de ficheiro foi dez, porque após alguns testes iniciais reparámos que para valores maiores a média não varia muito pois durante os testes os *outliers* foram praticamente inexistentes. Estes valores são depois exportados para um ficheiro onde depois foram feitos os cálculos dos valores médios de serialização/desserialização.

4.3 Resultados Experimentais

Neste capítulo serão apresentados os resultados das experiências realizadas entre as duas tecnologias escolhidas, XML e *Protocol Buffers*. Em cada sub-tópico deste capítulo serão apresentados os dados finais, relativamente a tempos de serialização, desserialização e tamanho dos ficheiros de *output*, entre as duas tecnologias para cada teste efectuado. É de notar que foram efetuados muitos mais testes que não foram incluídos por uma questão de espaço mas que estão acessíveis em anexo (ver [3]).

Para simplificar a leitura, foi introduzida o caractere 'k' na expressão dos nº de owners utilizados, sendo o seu equivalente, o milhar. Ou seja, 1k representa 1000.

4.3.1 Teste 1 - 1k *owners*

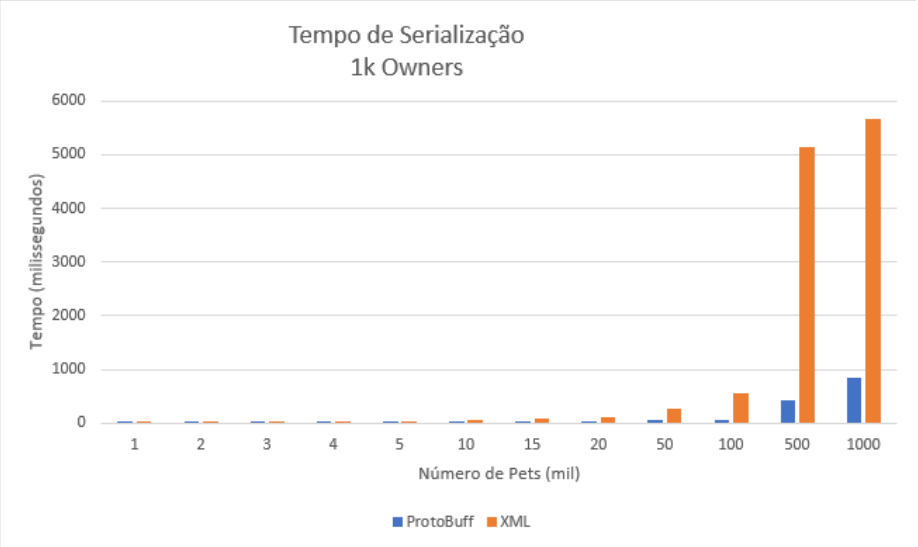

Figura 4: Tempo de serialização - 1k *owners*

A partir do gráfico na Figura 4 podemos ter uma noção da disparidade dos tempos de serialização quando testamos mil *owners* com os vários valores de pets. No gráfico, devido à escala não é percetível para os valores mais pequenos mas podemos ver na Tabela 1 que o *ProtoBuff* é sempre mais rápido que o *XML* no que toca à serialização. No entanto, é de observar que ambas as tecnologias seguem uma tendência de crescimento aparentemente $\mathcal{O}(n \log n)$.

1000 *owners*						
Nº de *pets*	1k *pets*	3k *pets*	5k *pets*	15k *pets*	100k *pets*	1000k *pets*
XML	10 ms	21 ms	31 ms	84 ms	551 ms	5652 ms
Protocol Buffer	1 ms	3 ms	6 ms	13 ms	70 ms	859 ms

Tabela 1: Teste 1 - Serialização

No que toca aos tempos de desserialização podemos ver pela Tabela 2 que o *XML* é ligeiramente mais lento neste processo do que no anterior e é ainda consideravelmente mais lento que o *ProtoBuff*. Este último, contrariamente ao *XML*, apresenta tempos de desserialização inferiores aos tempos de serialização, ou seja, é mais rápido a desserializar do que a serializar.

1000 *owners*						
Nº de *pets*	1k *pets*	3k *pets*	5k *pets*	15k *pets*	100k *pets*	1000k *pets*
XML	15 ms	27 ms	40 ms	110 ms	724 ms	7916 ms
Protocol Buffer	1 ms	2 ms	3 ms	7 ms	48 ms	511 ms

Tabela 2: Teste 1 - Desserialização

Por fim, resta analisar os tamanhos dos ficheiros resultantes da serialização que constam na Tabela 3. E daqui retiramos que os ficheiros gerados pelo *XML* são duas a três vezes maiores que os ficheiros produzidos pelo *ProtoBuff*. Esta diferença de tamanho pode-se justificar através da estrutura necessária que um ficheiro .xml proporciona (*tags*, *namespaces*, entre outros) ao contrário do ficheiro binário, que correspondente a *bytes*.

1000 *owners*						
Nº de *pets*	1k *pets*	3k *pets*	5k *pets*	15k *pets*	100k *pets*	1000k *pets*
Ficheiro XML (kB)	514 kB	1129 kB	1744 kB	4838 kB	31300 kB	315009 kB
Ficheiro *Protocol Buffer* (kB)	185 kB	398 kB	611 kB	1695 kB	11156 kB	114868 kB

Tabela 3: Teste 1 - Tamanho do ficheiros de *output*

4.3.2 Teste 2 - 5k *owners*

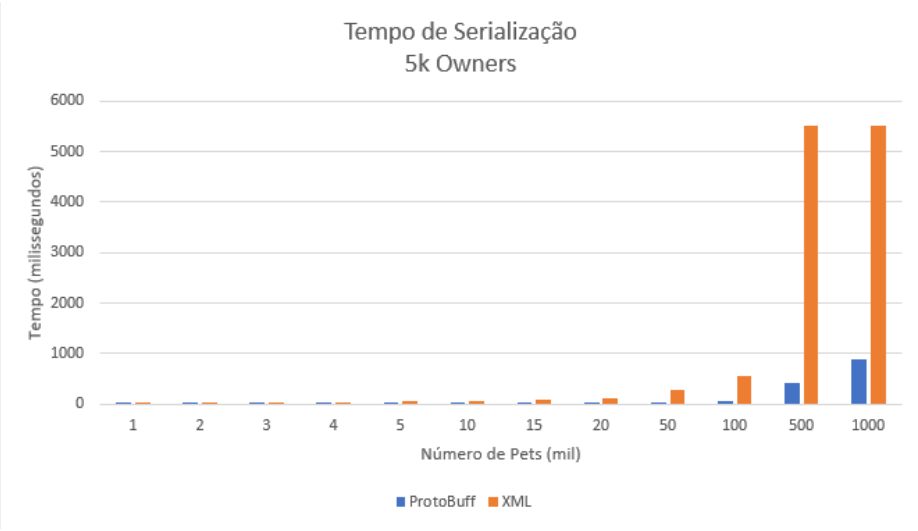

Figura 5: Tempo de Serialização - 5k *owners*

5000 *owners*						
Nº de *pets*	1k *pets*	3k *pets*	5k *pets*	15k *pets*	100k *pets*	1000k *pets*
XML	25 ms	36 ms	53 ms	101 ms	587 ms	5502 ms
Protocol Buffer	5 ms	7 ms	7 ms	14 ms	74 ms	879 ms

Tabela 4: Teste 2 - Serialização

No 2º teste realizado, o resultado que é mais interessante é o facto de apesar aumentar a quantidade de *pets* em 2000, a velocidade de serialização do *Protocol Buffers* manteve-se nos 7ms. A tendência manteve-se sempre, ou seja, a tecnologia da Google foi sempre a mais "responsiva"nesta operação face ao XML.

5000 *owners*						
Nº de *pets*	1k *pets*	3k *pets*	5k *pets*	15k *pets*	100k *pets*	1000k *pets*
XML	18 ms	32 ms	45 ms	114 ms	731 ms	7373 ms
Protocol Buffer	1 ms	2 ms	4 ms	10 ms	53 ms	562 ms

Tabela 5: Teste 2 - Desserialização

5000 *owners*						
Nº de *pets*	1k *pets*	3k *pets*	5k *pets*	15k *pets*	100k *pets*	1000k *pets*
XML (kB)	1369 kB	1984 kB	2599 kB	5693 kB	32155 kB	315855 kB
Protocol Buffer (kB)	524 kB	739 kB	953 kB	2038 kB	11499 kB	115215 kB

Tabela 6: Teste 2 - Tamanho dos ficheiro *output*

4.3.3 Teste 3 - 20k *owners*

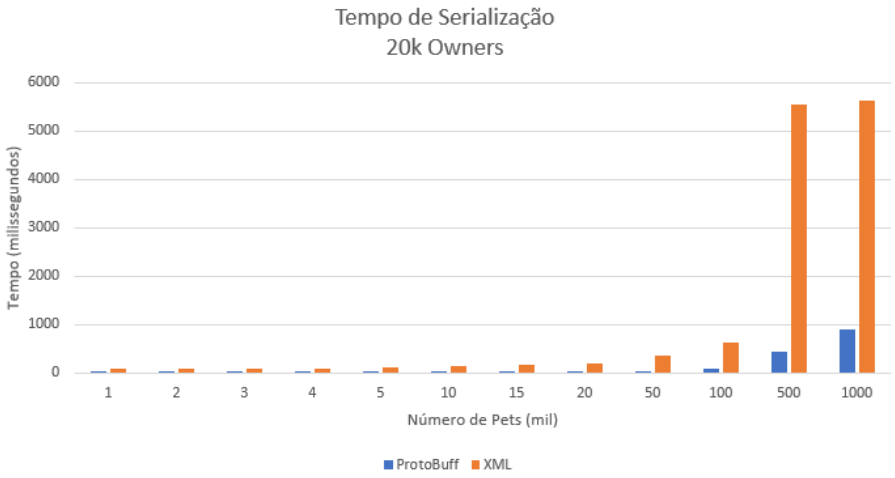

Figura 6: Teste 1 - 20k Owners

20000 *owners*						
Nº de *pets*	1k *pets*	3k *pets*	5k *pets*	15k *pets*	100k *pets*	1000k *pets*
XML	83 ms	100 ms	106 ms	157 ms	636 ms	5641 ms
Protocol Buffers	14 ms	16 ms	17 ms	21 ms	97 ms	893 ms

Tabela 7: Teste 3 - Serialização

20000 *owners*						
Nº de *pets*	1k *pets*	3k *pets*	5k *pets*	15k *pets*	100k *pets*	1000k *pets*
XML	111 ms	127 ms	141 ms	211 ms	840 ms	7454 ms
Protocol Buffers	7 ms	8 ms	10 ms	18 ms	65 ms	456 ms

Tabela 8: Teste 3 - Desserialização

200000 *owners*						
Nº de *pets*	1k *pets*	3k *pets*	5k *pets*	15k *pets*	100k *pets*	1000k *pets*
XML (kB)	4595 kB	5210 kB	5825 kB	8919 kB	35381 kB	319085 kB
Protocol Buffers (kB)	1814 kB	2029 kB	2243 kB	3337 kB	12804 kB	116514 kB

Tabela 9: Teste 3 - Tamanho dos ficheiros de *output*

4.3.4 Teste 4 - 1000k *owners*

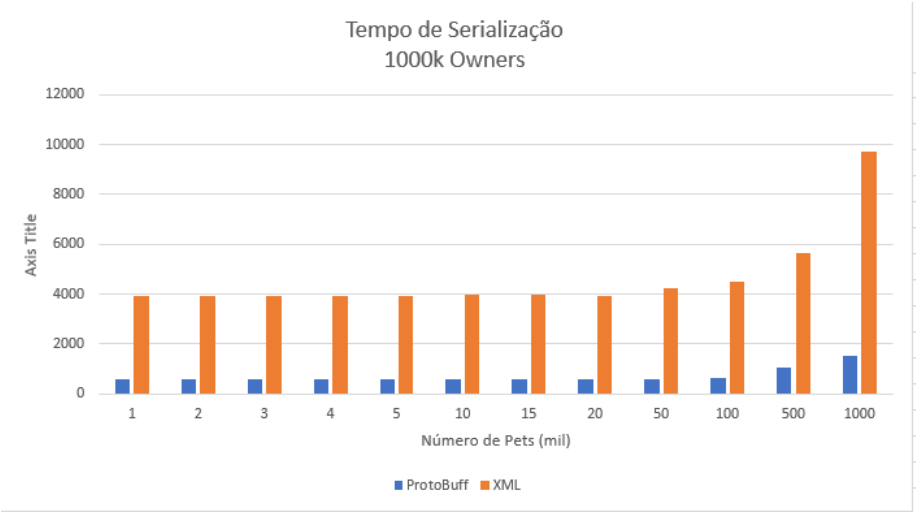

Figura 7: Teste 4 - 1000k Owners

1000000 *owners*						
Nº de *pets*	1k *pets*	3k *pets*	5k *pets*	15k *pets*	100k *pets*	1000k *pets*
XML	3908 ms	3938 ms	3938 ms	3972 ms	4493 ms	9696 ms
Protocol Buffers	595 ms	573 ms	583 ms	578 ms	653 ms	1531 ms

Tabela 10: Teste 4 - Serialização

Com base nos registos da tabela 10, verificamos que para uma quantidade enorme de *owners* o XML desde logo demorou aproximadamente 4 segundos, contrastando com uns aproximadamente 600 milisegundos por parte do *Protocol Buffers*. Ou seja o XML teve mais dificuldade para serializar uma maior quantidade de *owners*, sendo que esta estrutura de dados é relativamente simples. Portanto o *Protocol Buffers*, que apesar de ter que gerir uma quantidade enorme de *owners*, e sendo esta estrutura de dados mais simples que a dos *pets*, conseguiu claramente melhores resultados. Isto acontece também pela optimização desta tecnologia para serialização de estruturas mais simples.

1000000 *owners*						
Nº de *pets*	1k *pets*	3k *pets*	5k *pets*	15k *pets*	100k *pets*	1000k *pets*
XML	5691 ms	5530 ms	5592 ms	5756 ms	6660 ms	14371 ms
Protocol Buffers	498 ms	479 ms	505 ms	495 ms	539 ms	1231 ms

Tabela 11: Teste 4 - Desserialização

Relativamente à desserialização, nota-se um dado interessante relativamente ao *Protocol Buffers*, visto que é, uma vez mais, mais rápido a desserializar os dados do que a serializá-los.

100000 *owners*						
Nº de *pets*	1k *pets*	3k *pets*	5k *pets*	15k *pets*	100k *pets*	1000k *pets*
XML (kB)	217746 kB	218362 kB	218976 kB	222069 kB	248531 kB	532237 kB
Protocol Buffers (kB)	88488 kB	88704 kB	88919 kB	90012 kB	99556 kB	204147 kB

Tabela 12: Teste 4 - Tamanho dos ficheiros de *output*

5 Conclusão

Após a análise dos dados, no que toca à serialização em todos os testes efetuados o *Protocol Buffers* tem um desempenho superior sendo em muitos casos cinco ou seis vezes mais rápido que o *XML*.

Relativamente ao tempo de desserialização, para uma quantidade menor de dados obtivemos tempos menores utilizando o *XML* mas a partir dos três mil *owners* isto deixa de ser verdade e o *Protocol Buffers* volta a ganhar destaque.

No que toca ao espaço ocupado pelos ficheiros dos dados serializados, em todos os testes, obtivemos ficheiros duas a três vezes menores para o *Protocol Buffers* do que para o XML. Esta informação é especialmente importante quando consideramos um cenário de comunicação em que estes dados terão que ser enviados para um servidor/cliente.

Assim, olhando apenas para os dados quantitativos facilmente se chega a conclusão de que o *Protocol Buffers* é uma tecnologia superior ao *XML* por ser mais rápida na serialização/desserialização e para além disto, gera também ficheiros muito mais pequenos. No entanto, importa ainda olhar para alguns dados não quantitativos.

Para a implementação no caso do *XML*, foi apenas necessário utilizar uma biblioteca existente, JAXB (*Java Architecture for XML Binding*) para se realizar a serialização/desserialização dos dados, automatizando a tarefa de implementação desta operação. De referir que os dados recolhidos face aos tempos de serialização/desserialização podem depender das bibliotecas usadas, mais concretamente da optimização existente das mesmas.

Enquanto que, no caso do *Protocol Buffers* foi necessário proceder à instalação de um compilador (*Protoc*), criar a estrutura de dados equivalente à que foi utilizada no estudo do XML, através de um ficheiro .proto, e por fim, gerar os ficheiros *.java* necessários a partir do compilador instalado. Após isso, foi gerado um ficheiro *.java* que permite gerir todas as informações relativas à nossa nova estrutura de dados (por exemplo, *getters* e *setters*).

Para além disto, o *XML* contém no ficheiro toda a informação necessária para a reconstrução do objecto no destino, o que simplesmente não acontece para o *Protocol Buffers* onde fazer a reconstrução dos dados no destino pode apresentar um desafio se não for conhecido o esquema inicial.

Outro aspecto a ter em conta em relação ao *Protocol Buffers*, é a complexidade das estruturas de dados que pretendemos serializar, visto que esta tecnologia está optimizada para modelos de dados mais simples. Um exemplo disto é, por exemplo, o tipo de dados disponíveis, sendo elas maioritariamente valores inteiros, *float*, *strings*, *bytes* e *booleanos*.

Posto isto, concluímos que a tecnologia desenvolvida pela Google, *Protocol Buffers*, é a melhor tecnologia, pelo menos no contexto deste projecto, sendo a mais rápida e eficiente para serialização e desserialização dos dados utilizados neste projecto. No entanto, por todas as razões apresentadas, isto pode não se verificar noutros casos e cabe ao programador decidir que tecnologia utilizar consoante as condições do problema que pretende resolver.

Referências

[1] *Script para gerar testes*
https://tinyurl.com/2xyj8hsp

[2] *Código fonte*
https://tinyurl.com/3sannps

[3] *Testes realizados*
https://tinyurl.com/xry58f6t