UNIVERSIDADE D
COIMBRA

Gui Costa, 2021186342
Dário Félix, 2018275530

Trabalho 1
Data Representation and Serialization Formats

Relatório no âmbito da cadeira de Integração de Sistemas do Mestrado em Engenharia Informática, orientado pelo Professor Filipe Araújo (PL1), do Departamento de Engenharia Informática da Faculdade de Ciências e Tecnologias da Universidade de Coimbra.

30 de setembro de 2022

Conteúdo

Lista de Figuras

1 Introdução

1.1 Sumário

Este trabalho tem como objetivo em aprender e comparar as principais diferenças dos formatos mais utilizados na serialização de dados, nomeadamente entre XML, XML comprimido com Gzip e Protocol Buffers, através de parâmetros como a complexidade de programação, velocidade de serialização e de desserialização, tamanho dos dados serializados, etc.

Assim, implementou-se um pequeno programa que gera dados aleatoriamente conforme a estrutura de dados definida, e serializa-os nos diferentes formatos. Fez-se várias experiências de modo a responder aos parâmetros referidos.

Neste documento aborda-se a implementação (as estruturas de dados e em que parte do código é medido o tempo), a metodologia utilizada e o seu ambiente (características do computador e as tecnologias e/ou bibliotecas utilizadas), os resultados obtidos e, por fim, a discussão e a análise desses resultados.

1.2 Palavras-chave

Formatos de representação de dados (texto e binário); XML; XML comprimido com Gzip; Google Protocol Buffers; Complexidade de Programação; Velocidade de serialização e de desserialização; Tamanho dos dados serializados; Tempo de transferência na rede; Java;

2 Implementação

2.1 Tecnologias e Bibliotecas Utilizadas

Nas tabelas Table 2.1 e Table 2.2 são mencionadas as bibliotecas utilizadas e as respetivas versões.

Java Development Kit (JDK)	Oracle OpenJDK version 17.0.2

Tabela 2.1: Versão do Java.

Artefacto	**Versão**
jakarta.xml.bind-api	3.0.0
jaxb-impl	3.0.0
junit	4.11
javafaker	0.12
protobuf-java	3.21.6
opencsv	4.1

Tabela 2.2: Bibliotecas utilizadas.

2.2 Estruturas de Dados

A estrutura de dados é comum a todas as experiências: tanto a classe *Professor* (Código 2.2) como a classe *Student* (Código 2.3) são subclasses da classe abstrata *Pessoa* (Código 2.1), herdando os atributos nome, identificador único, número de telefone, endereço e data de nascimento; A classe *Student* (Código 2.3) possui ainda os atributos género, o identificador único do professor e a data de registo; A classe *Professor* (Código 2.2) tem uma lista de *Students*.

Assim, existe uma relação de muitos-para-um entre os *Students* e o *Professor*.

Para efeitos de serialização é utilizado um objeto da classe *ProfList* (Código 2.4) que apenas possui um atributo: uma lista de *Professores*.

```
1 public abstract class Pessoa {
2     String uuid;
3     String name;
4     String phoneNum;
5     String addr;
6     Date bday;
7 
8     public Pessoa() {
9     }
```

```
10
11      public Pessoa(String uuid, String name, String phoneNum, String
        addr, Date bday) {
12          this.uuid = uuid;
13          this.name = name;
14          this.phoneNum = phoneNum;
15          this.addr = addr;
16          this.bday = bday;
17      }
18  }
```

Código 2.1: Classe Pessoa.

```
1  class Professor extends Pessoa {
2      List<Student> student;
3
4      public Professor() {
5      }
6
7      public Professor(String uuid, String name, String phoneNum,
       String addr, Date bday, List<Student> student) {
8          super(uuid, name, phoneNum, addr, bday);
9          this.student = student;
10     }
11 }
```

Código 2.2: Classe Professor.

```
1  public class Student extends Pessoa {
2      String profuuid;
3      Character gender;
4      Date registrationDate;
5
6      public Student() {
7      }
8
9      public Student(String uuid, String name, String phoneNum, String
       addr, Date bday, String profuuid, Character gender, Date
       registrationDate) {
10         super(uuid, name, phoneNum, addr, bday);
11         this.profuuid = profuuid;
12         this.gender = gender;
13         this.registrationDate = registrationDate;
14     }
15 }
```

Código 2.3: Classe Student.

```
1  public class ProfList {
2      List<Professor> professor;
3
4      public ProfList() {
5      }
6  }
```

Código 2.4: Classe ProfList.

O protocol buffers segue a mesma estrutura de dados descrita anteriormente (Código 2.5).

```
message ProtoStudent {
  optional string uuid = 1;
  optional string name = 2;
  optional string phoneNum = 3;
  optional string addr = 4;
  optional google.protobuf.Timestamp bday = 5;
  optional string profuuid = 6;
  optional string gender = 7;
  optional google.protobuf.Timestamp registrationDate = 8;

}

message ProtoProfessor {
  optional string uuid = 1;
  optional string name = 2;
  optional string phoneNum = 3;
  optional string addr = 4;
  optional google.protobuf.Timestamp bday = 5;
  repeated ProtoStudent student = 6;
}

message ProtoProfList {
  repeated ProtoProfessor professor = 1;
}
```

Código 2.5: Estrutura do Protocol Buffers

2.3 Medição do Tempo

Com o objetivo de criar as mesmas condições de experimentação para os diferentes formatos, a medição do tempo abrange as mesmas funções das mesmas bibliotecas, dentro daquilo que pode ser igual.

Além disso, no *Protocol Buffers*, o tempo de serialização (Código 2.11) e de desserialização (Código 2.12) foi separado do tempo da inicialização das estruturas de dados (Código 2.10 e Código 2.13) específicas ao *Protocol Buffers*.

```
long start = System.currentTimeMillis();
jakarta.xml.bind.Marshaller jaxbMarshaller = rotinaJaxbMarshaller();
File file = new File(App.utilizarPath + fileIndex + App.extensaoXML);
jaxbMarshaller.marshal(obj, file);
long end = System.currentTimeMillis();
```

Código 2.6: Serialização utilizando XML.

```
long start = System.currentTimeMillis();
Unmarshaller jaxbUnmarshaller = rotinaJaxbUnmarshaller();
InputStream inputStream = Files.newInputStream(Paths.get(App.
   utilizarPath + fileIndex + App.extensaoXML));
ProfList obj = (ProfList) jaxbUnmarshaller.unmarshal(inputStream);
inputStream.close();
```

```
long end = System.currentTimeMillis();
```

Código 2.7: Desserialização utilizando XML.

```
long start = System.currentTimeMillis();
jakarta.xml.bind.Marshaller jaxbMarshaller = rotinaJaxbMarshaller();
File file = new File(App.utilizarPath + fileIndex + App.
    extensaoXMLGzip);
GZIPOutputStream outputStream = new GZIPOutputStream(Files.
    newOutputStream(file.toPath()));
jaxbMarshaller.marshal(obj, outputStream);
outputStream.close();
long end = System.currentTimeMillis();
```

Código 2.8: Serialização utilizando XML comprimido com Gzip.

```
long start = System.currentTimeMillis();
Unmarshaller jaxbUnmarshaller = rotinaJaxbUnmarshaller();
GZIPInputStream inputStream = new GZIPInputStream(Files.
    newInputStream(Paths.get(App.utilizarPath + fileIndex + App.
    extensaoXMLGzip)));
ProfList obj = (ProfList) jaxbUnmarshaller.unmarshal(inputStream);
inputStream.close();
long end = System.currentTimeMillis();
```

Código 2.9: Desserialização utilizando XML comprimido com Gzip.

```
long start = System.currentTimeMillis();
ProtoProfList.Builder protoProfList = ProtoProfList.newBuilder();
Timestamp timestampRegistrationDate;
Timestamp timestampBday;
ProtoStudent.Builder protoStudent;
ProtoProfessor.Builder protoProfessor;

for (Professor professor: obj.getprofessor()) {
    protoProfessor = ProtoProfessor.newBuilder();
    timestampBday = new Timestamp(professor.getBday().getTime());

    protoProfessor.setUuid(professor.getUuid());
    protoProfessor.setName(professor.getName());
    protoProfessor.setPhoneNum(professor.getPhoneNum());
    protoProfessor.setAddr(professor.getAddr());
    protoProfessor.setBday(com.google.protobuf.Timestamp.newBuilder()
    .setSeconds(timestampBday.getTime()).setNanos(timestampBday.
    getNanos()).build());

    for(Student student : professor.getStudent()) {
        protoStudent = ProtoStudent.newBuilder();
        timestampBday = new Timestamp(student.getBday().getTime());
        timestampRegistrationDate = new Timestamp(student.
    getRegistrationDate().getTime());

        protoStudent.setAddr(student.getAddr());
        protoStudent.setBday(com.google.protobuf.Timestamp.newBuilder
    ().setSeconds(timestampBday.getTime()).setNanos(timestampBday.
    getNanos()).build());
        protoStudent.setGender(student.getGender().toString());
        protoStudent.setName(student.getName());
```

```
27          protoStudent.setPhoneNum(student.getPhoneNum());
28          protoStudent.setUuid(student.getUuid());
29          protoStudent.setProfuuid(student.getProfuuid());
30          protoStudent.setRegistrationDate(com.google.protobuf.
    Timestamp.newBuilder().setSeconds(timestampRegistrationDate.
    getTime()).setNanos(timestampRegistrationDate.getNanos()));
31
32          protoProfessor.addStudent(protoStudent.build());
33      }
34      protoProfList.addProfessor(protoProfessor.build());
35  }
36  long end = System.currentTimeMillis();
```

Código 2.10: Inicialização das estruturas de dados para a serialização utilizando Protocol Buffers.

```
1  start = System.currentTimeMillis();
2  File file = new File(App.utilizarPath + fileIndex + App.
       extensaoProtobuf);
3  OutputStream outputStream = Files.newOutputStream(file.toPath());
4  protoProfList.build().writeTo(outputStream);
5  outputStream.close();
6  end = System.currentTimeMillis();
```

Código 2.11: Serialização utilizando Protocol Buffers.

```
1  long start = System.currentTimeMillis();
2  File file = new File(App.utilizarPath + fileIndex + App.
       extensaoProtobuf);
3  InputStream inputStream = Files.newInputStream(file.toPath());
4  ProtoProfList protoProfList = ProtoProfList.parseFrom(inputStream);
5  inputStream.close();
6  long end = System.currentTimeMillis();
```

Código 2.12: Desserialização utilizando Protocol Buffers.

```
1  start = System.currentTimeMillis();
2  ProfList obj;
3  ArrayList <Student> students;
4  Student student;
5  Professor professor;
6  ArrayList <Professor> professores = new ArrayList<>();
7  Timestamp timestampBday, timestampRegist;
8
9  for (ProtoProfessor protoProfessor : protoProfList.getProfessorList()
       ) {
10     students = new ArrayList<>();
11     for (ProtoStudent protoStudent: protoProfessor.getStudentList())
       {
12          timestampBday = new Timestamp(protoStudent.getBday().
       getSeconds());
13          timestampBday.setNanos(protoStudent.getBday().getNanos());
14          timestampRegist = new Timestamp(protoStudent.
       getRegistrationDate().getSeconds());
15          timestampRegist.setNanos(protoStudent.getRegistrationDate().
       getNanos());
16          student = new Student(
17                  protoStudent.getUuid(),
```

```
                protoStudent.getName(),
                protoStudent.getPhoneNum(),
                protoStudent.getAddr(),
                new Date(timestampBday.getTime()),
                protoStudent.getProfuuid(),
                protoStudent.getGender().charAt(0),
                new Date(timestampRegist.getTime())
        );
        students.add(student);
    }
    timestampBday = new Timestamp(protoProfessor.getBday().getSeconds
    ());
    timestampBday.setNanos(protoProfessor.getBday().getNanos());
    professor = new Professor(
            protoProfessor.getUuid(),
            protoProfessor.getName(),
            protoProfessor.getPhoneNum(),
            protoProfessor.getAddr(),
            new Date(timestampBday.getTime()),
            students
    );
    professores.add(professor);
}
obj = new ProfList(professores);
end = System.currentTimeMillis();
```

Código 2.13: Inicialização das estruturas de dados após a desserialização utilizando Protocol Buffers.

3 Metodologias Experimentais

3.1 Características do Computador

As experiências foram executadas no mesmo equipamento, cujas as especificações estão descritas na Table 3.1.

Sistema Operativo	Microsoft Windows 11 Home, Versão 21H2, OS Build 22000.1042
Memória	8 GB DDR4 embutidos, 8GB DDR4-2666 SO-DIMM
Disco	512 GB M.2 NVMe PCIe 3.0 SSD
Processador	Intel(R) Core(TM) i7-9750H CPU @ 2.60 GHz (12M Cache, up to 4.50 GHz, 6 cores)
Placa Gráfica	NVIDIA GeForce GTX 1660Ti, 6GB GDDR6

Tabela 3.1: Especificações do computador.

3.2 Parametrização

O programa apresenta um painel de parâmetros, onde se pode alterar algumas características da experiência a executar (Código 3.1). Note-se que os parâmetros utilizados para gerar os resultados experimentais (chapter 4) são tal e qual como estão definidos no Código 3.1 e Código 3.2.

Para facilitar a análise dos dados, os dados utilizados foram divididos em 3 segmentos, o primeiro em que o numero de professores é igual ao numero de estudantes, (10,10) e (50,50) são exemplos, no segundo segmento, o número de professores é constante e igual a 50 e o numero de estudantes varia e por fim, no terceiro segmento, o número de estudantes é igual a 50 e o numero de professores varia (linhas 11, Código 3.1).

```
1  // ****** ****** PARAMETROS ****** ******
2  public static String utilizarPath = App.pathDario;
3  public static boolean testarXML = true;
4  public static boolean testarXMLGzip = true;
5  public static boolean testarProtobuf = true;
6  public static long seed = 111155588;
7  public static int totalRepeticoes = 20;
8
9  public static boolean utilizarDiferentesSeeds = true;
10
11 public static long[][] matrizGenSizesProfStudent = {
12         {10, 10},
13         {50, 50},
14         {100, 100},
15         {200, 200},
```

```
16          {300, 300},
17          {500, 500},
18          {700, 700},
19          {50, 10},
20          {50, 100},
21          {50, 200},
22          {50, 300},
23          {50, 500},
24          {50, 1000},
25          {10, 50},
26          {100, 50},
27          {200, 50},
28          {300, 50},
29          {500, 50},
30          {1000, 50}
31  };
```

Código 3.1: Parâmetros da aplicação.

3.2.1 Seeds

Existe a possibilidade de escolher entre 2 modos através da alteração da variável *utilizarDiferentesSeeds* (linhas 9, Código 3.1): se o valor for *false*, a experiência é executada (com múltiplas repetições) sempre com a mesma seed definida na variável *seed* (linhas 6, Código 3.1); se o valor for *true*, ao longo da experiência, e a cada repetição, a seed altera-se ordenadamente conforme a constante *listaSeeds* (Código 3.2).

Neste trabalho utilizou-se seeds diferentes por cada repetição.

```
1  // ****** CONSTANTES ******
2  public final static long[] listaSeeds = {12131451, 510, 6711890,
       10000, 987165, 11171, 431210, 4444144, 1213, 246180, 4516, 13579,
       7819, 5431543, 2711727, 7514385, 9996166, 190121835, 1029138,
       47116, 2111, 90110, 389146, 333133, 874911119};
```

Código 3.2: Constantes da aplicação.

3.2.2 Repetições

É possível definir o número de repetições a levar a cabo numa experiência, definido a variável *totalRepeticoes* (linhas 7, Código 3.1).

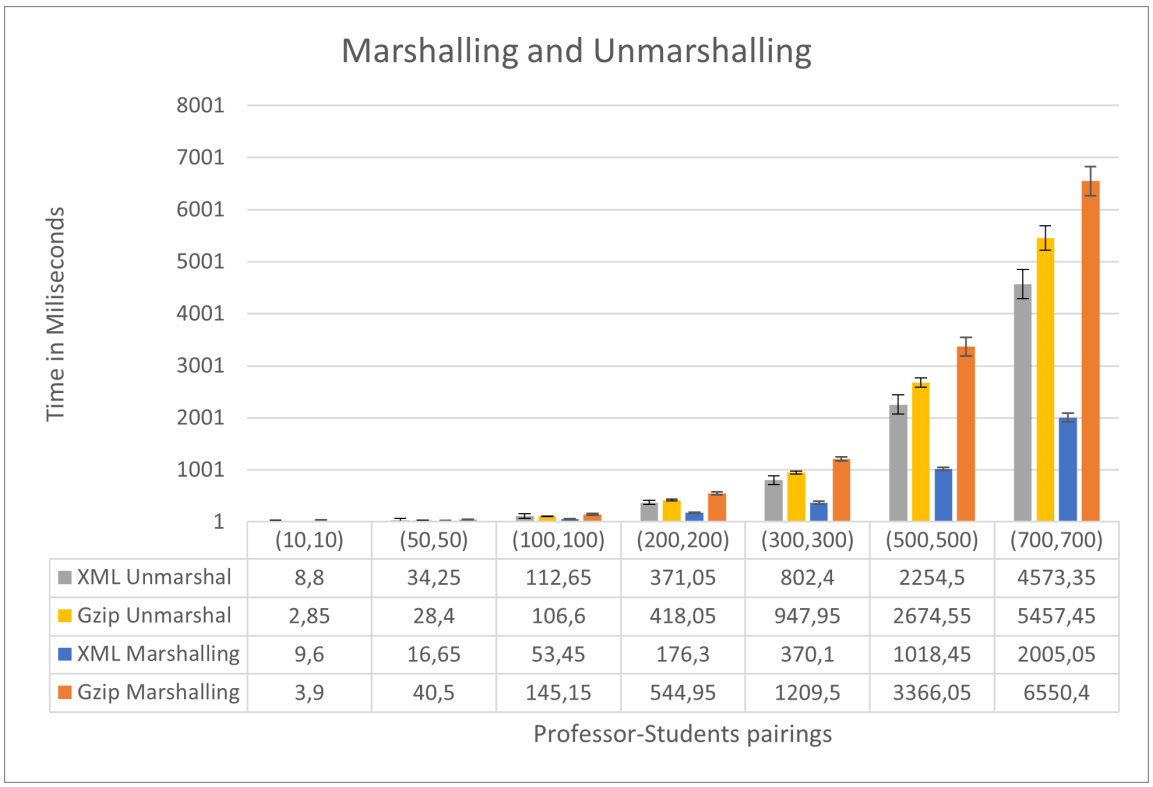
Neste trabalho fez-se 20 repetições.

4 Resultados Experimentais

4.1 Preâmbulo

Os resultados aqui apresentados refletem já a média e o desvio-padrão das 20 repetições, com diferentes seeds, como já foi referido anteriormente nas Metodologias Experimentais (chapter 3).

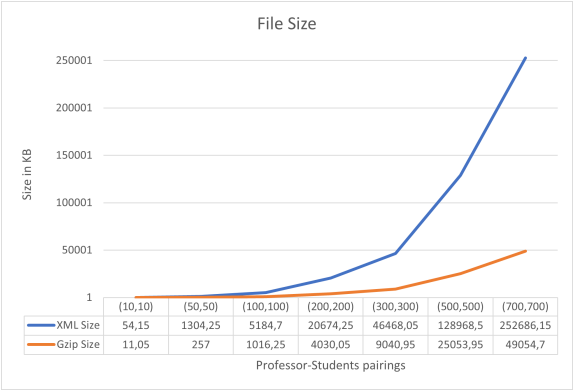
4.2 XML e XML Comprimido com Gzip



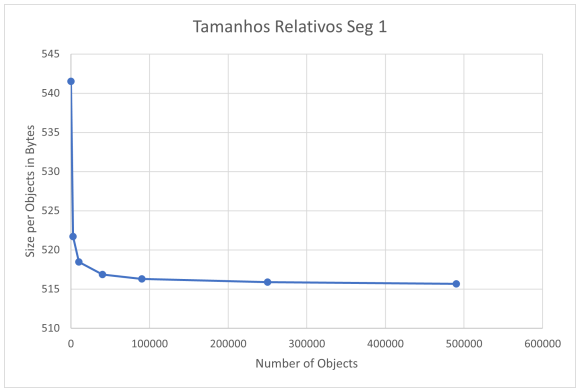
	(10,10)	(50,50)	(100,100)	(200,200)	(300,300)	(500,500)	(700,700)
XML Unmarshal	8,8	34,25	112,65	371,05	802,4	2254,5	4573,35
Gzip Unmarshal	2,85	28,4	106,6	418,05	947,95	2674,55	5457,45
XML Marshalling	9,6	16,65	53,45	176,3	370,1	1018,45	2005,05
Gzip Marshalling	3,9	40,5	145,15	544,95	1209,5	3366,05	6550,4

Figura 4.1: Tempo de marshalling e unmarshalling para XML e XML comprimido com Gzip

Na Figure 4.1 verifica-se que o tempo de Marshalling e Unmarshalling no XML Comprimido (GZip) é significativamente maior que o tempo no XML. Isto deve-se ao facto do tempo de compressão e descompressão estar incluído neste processo. Consegue-se também visualizar que o tempo de Marshalling para XML é o menor de todos.



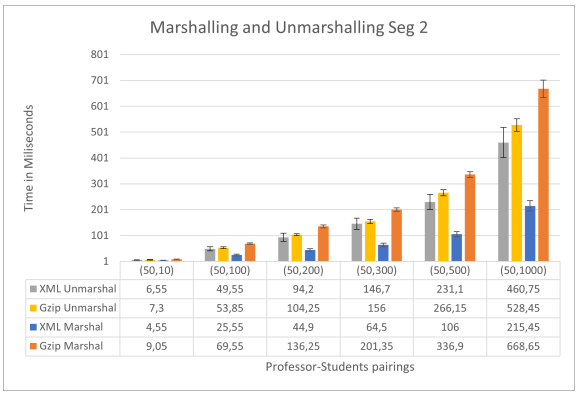
(a) Tamanho absoluto dos ficheiros XML, para cada par professor-estudante do segmento 1.



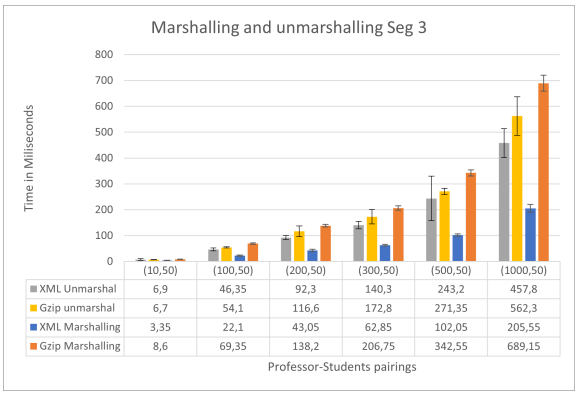
(b) Tamanho relativo dos ficheiros XML, para cada par professor-estudante, segmento 1.

Figura 4.2: Tamanhos de XML e Gzip, Seg 1

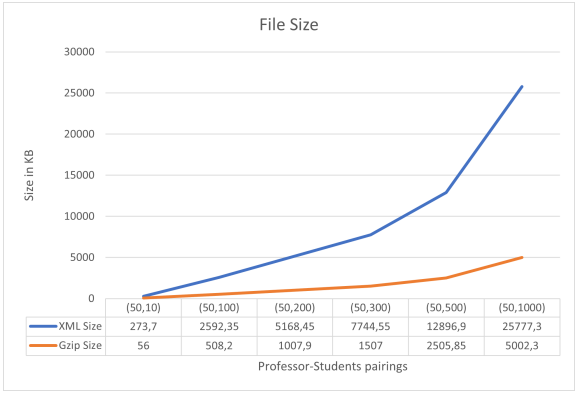
A partir da figura 4.2a observamos que o tamanho dos ficheiros para XML não comprimido cresce quase exponencialmente. O gzip tem um crescimento mais lento. Na figura 4.2b verifica-se que o numero de bytes por objeto (o numero de objetos é dado pelo professores*estudantes do par) tende a convergir para um valor de cerca de 516 bytes. O mesmo comportamento observa-se para o gzip, apesar de não ilustrado.



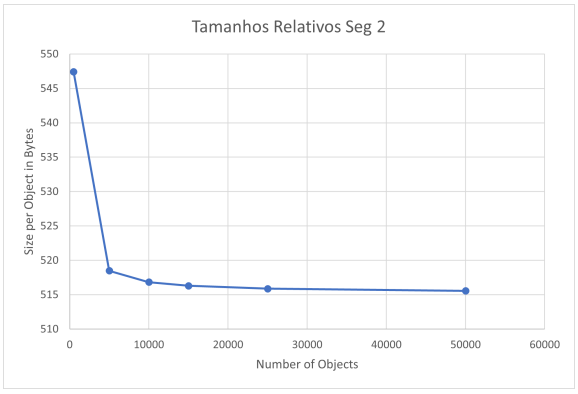
(a) Tempo de marshalling e unmarshalling para XML e XML comprimido com Gzip, segmento 2



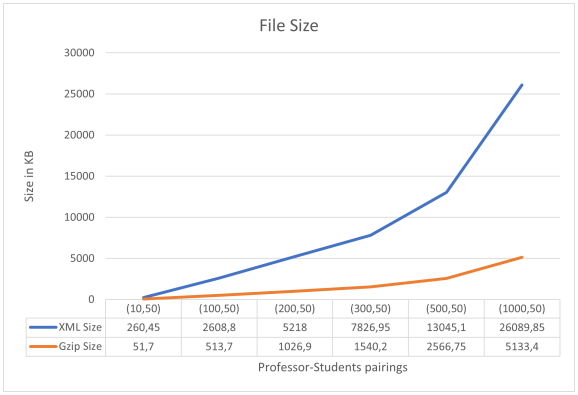
(b) Tempo de marshalling e unmarshalling para XML e XML comprimido com Gzip, segmento 3



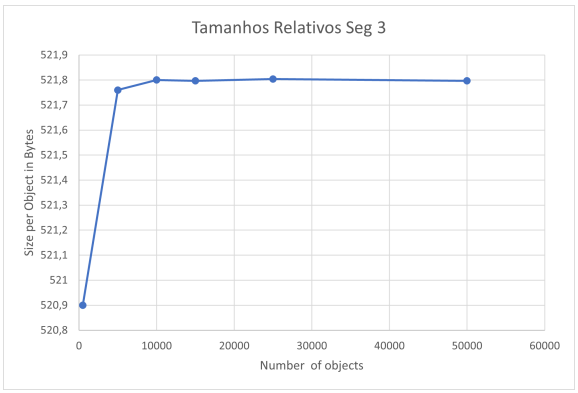
(c) Tamanho absoluto dos ficheiros XML e XML+Gzip, para cada par professor-estudante do segmento 2.



(d) Tamanho relativo (em bytes por objeto) dos ficheiros XML, para cada par professor-estudante do segmento 2.



(e) Tamanho absoluto dos ficheiros XML e XML+Gzip, para cada par professor-estudante do segmento 2.



(f) Tamanho relativo (em bytes por objeto) dos ficheiros XML, para cada par professor-estudante do segmento 3.

Figura 4.3: Tamanhos e tempos para Gzip e XML, Seg 2 e 3

Agrupamos estes resultados devido ao facto deles serem bastante semelhantes aos resultados do primeiro segmento: os tempos e os tamanhos tem o mesmo comportamento a exceção do que se observa na figura 4.3f, em que o valor varia de menor para maior (o do que o que acontece na figura 4.3d). No entanto continua a convergir para um valor, aproximadamente 521 bytes.

É possível concluir, através da analise destes segmentos que o tamanho dos ficheiros e, por consequência, o tempo despendido em marshalling e unmarshalling depende

principalmente do numero total de objetos e não de um tipo (professor ou estudante) em especifico. Pode-se entender isto, pois ambos, professor e estudante, tem como origem a classe pessoa, e estudante apenas tem mais um atributo único, sendo que o outro atributo a mais é uma referência ao uuid do objeto professor.

4.3 Protocol Buffers

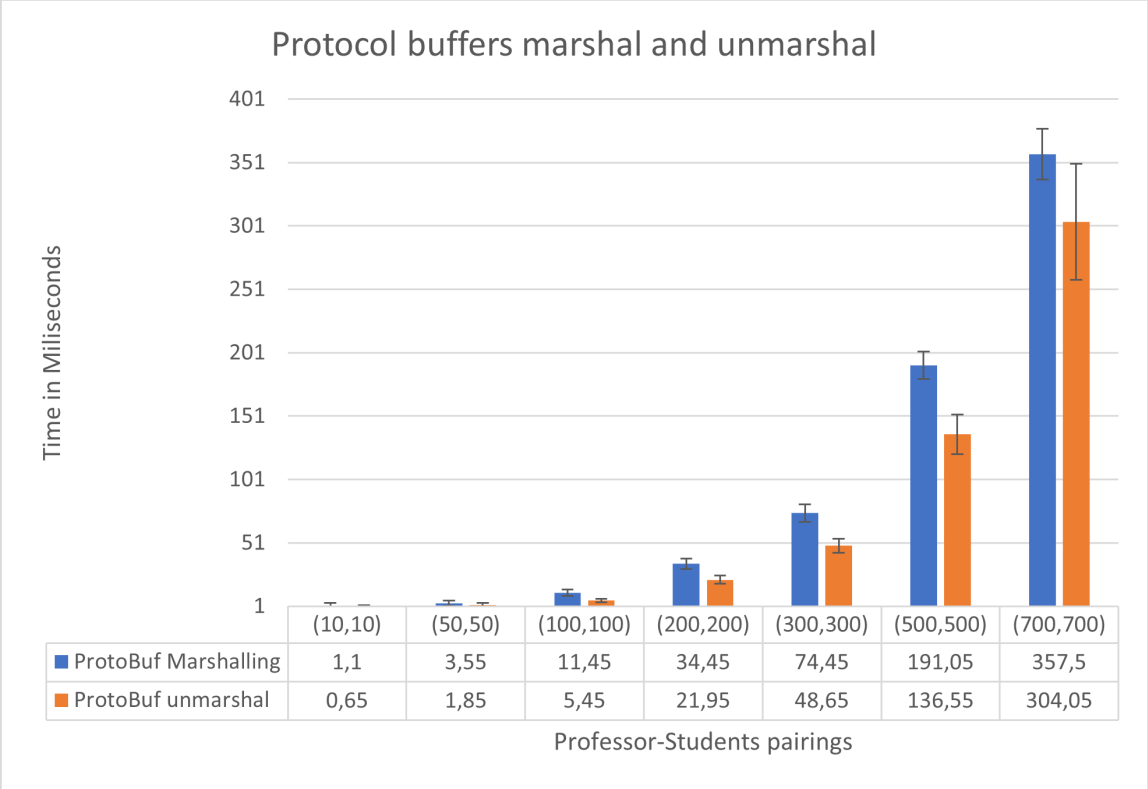


Figura 4.4: Tempo de marshalling e unmarshalling, utilizando Protocol Buffers no segmento 1.

No Protocol Buffers, nota-se que os tempos de serialização são ligeiramente superior aos tempos de desserialização. Aparenta ter uma curva exponencial.

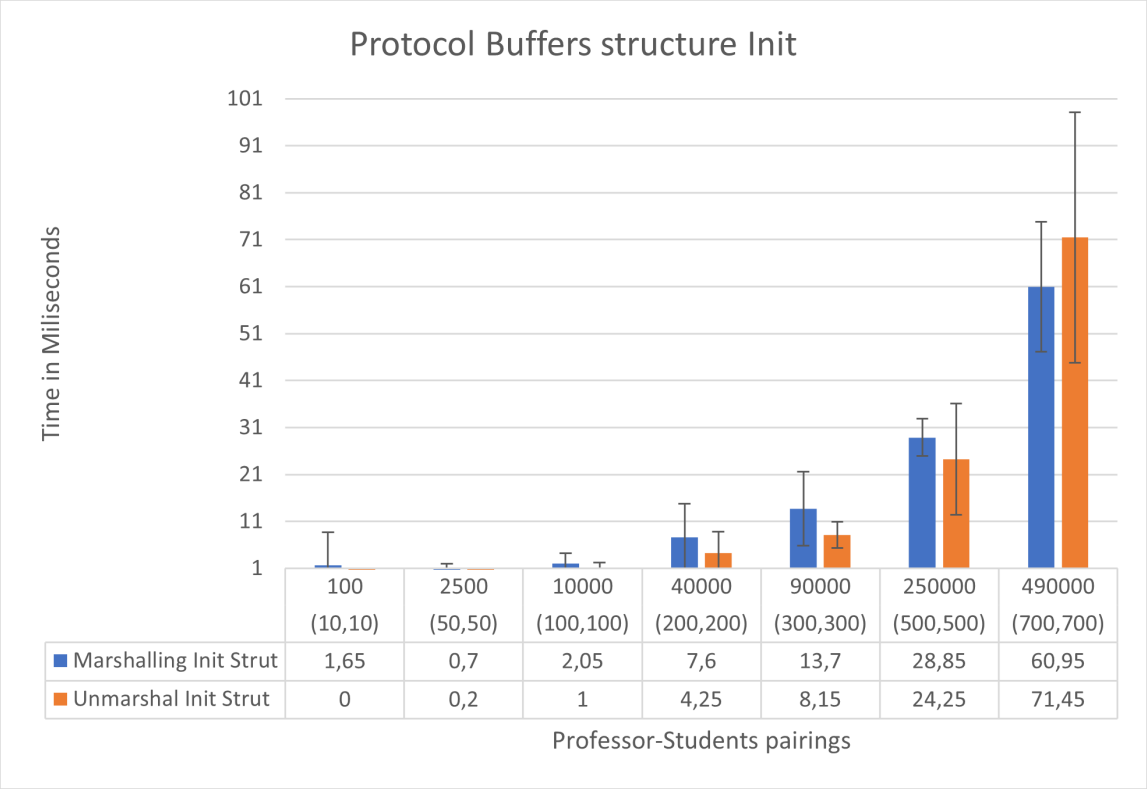
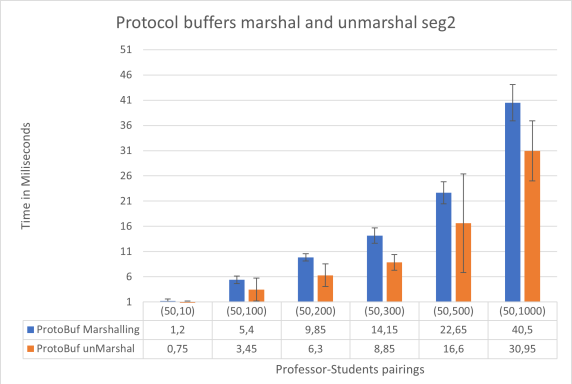


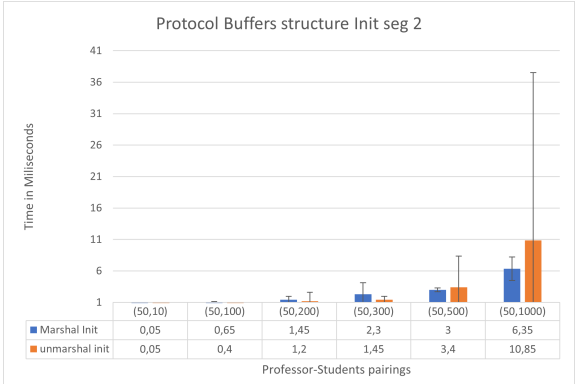
Figura 4.5: Tempo de inicialização das estruturas de dados antes do marshalling e depois do unmarshalling, utilizando Protocol Buffers no segmento 1.

Evidencia-se em todos os gráficos dos tempos de inicialização das estruturas dos dados específicas ao Protocol Buffers uma maior variação de tempos entre repetições, evidenciado pelo valor do desvio-padrão, face a todos os outros gráficos.

Será que diferentes dados (mantendo em quantidade) influência brutalmente o tempo de mapeamento e inicialização das estruturas de dados?

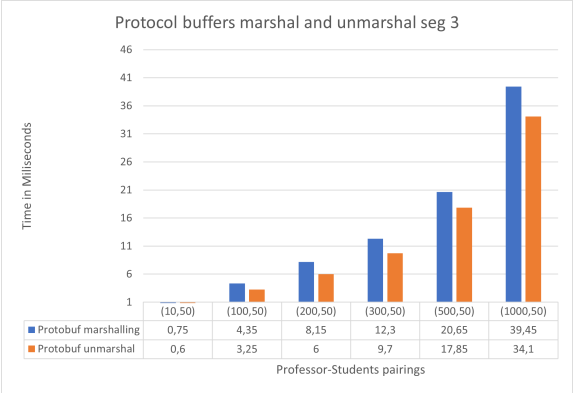


(a) Tempo de marshalling e unmarshalling, utilizando Protocol Buffers no segmento 2.

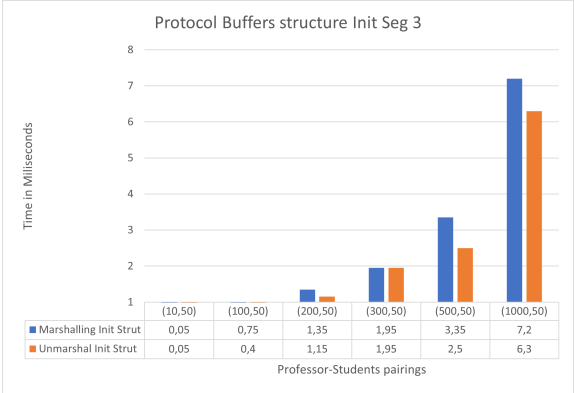


(b) Tempo de inicialização das estruturas de dados antes do marshalling e depois do unmarshalling, utilizando Protocol Buffers no segmento 2.

Figura 4.6: Tempos do protocol buffers seg 2



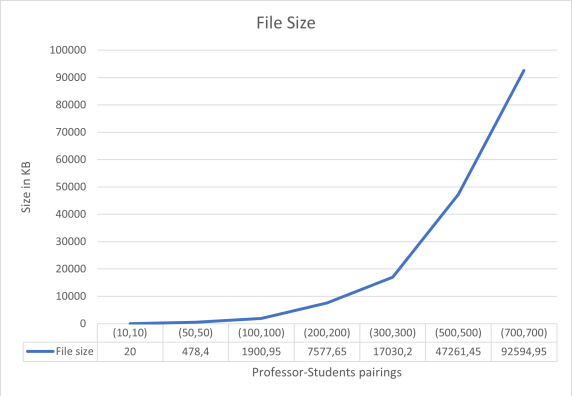
(a) Tempo de marshalling e unmarshalling, utilizando Protocol Buffers no segmento 3.



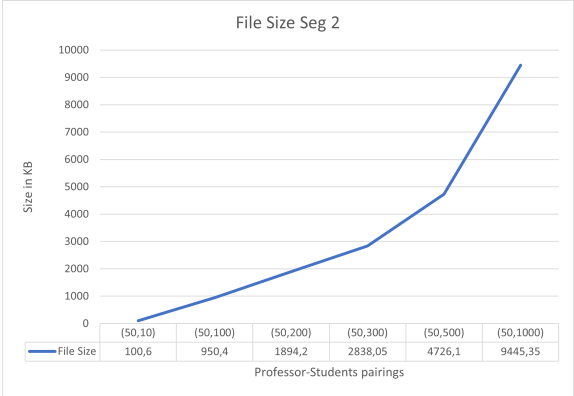
(b) Tempo de inicialização das estruturas de dados antes do marshalling e depois do unmarshalling, utilizando Protocol Buffers no segmento 3.

Figura 4.7: Tempos do protocol buffers seg 3

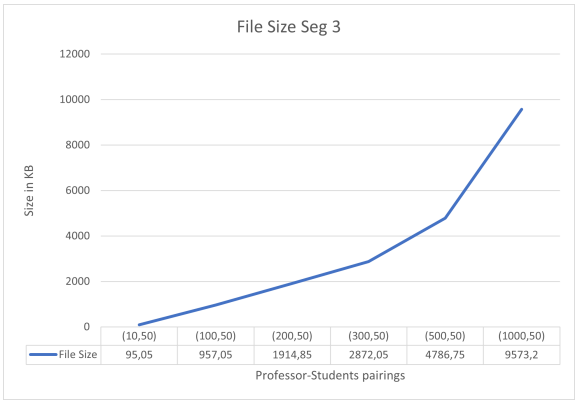
Mais uma vez, conclui-se que neste caso concreto, um número de professores substancialmente superior ao número de estudante e vice-versa, não diverge dos resultados obtidos no segmento 1.



(a) Tamanho absoluto dos ficheiros gerados pelo Protocol Buffers, para cada par professor-estudante do segmento 1.



(b) Tamanho absoluto dos ficheiros gerados pelo Protocol Buffers, para cada par professor-estudante do segmento 2.



(c) Tamanho absoluto dos ficheiros gerados pelo Protocol Buffers, para cada par professor-estudante do segmento 3.

Figura 4.8: Tamanhos dos ficheiros para protocol buffers

Independentemente do rácio professores-estudantes, todas as situações tem uma evolução semelhante.

5 Análise e Discussão

5.1 Comparação

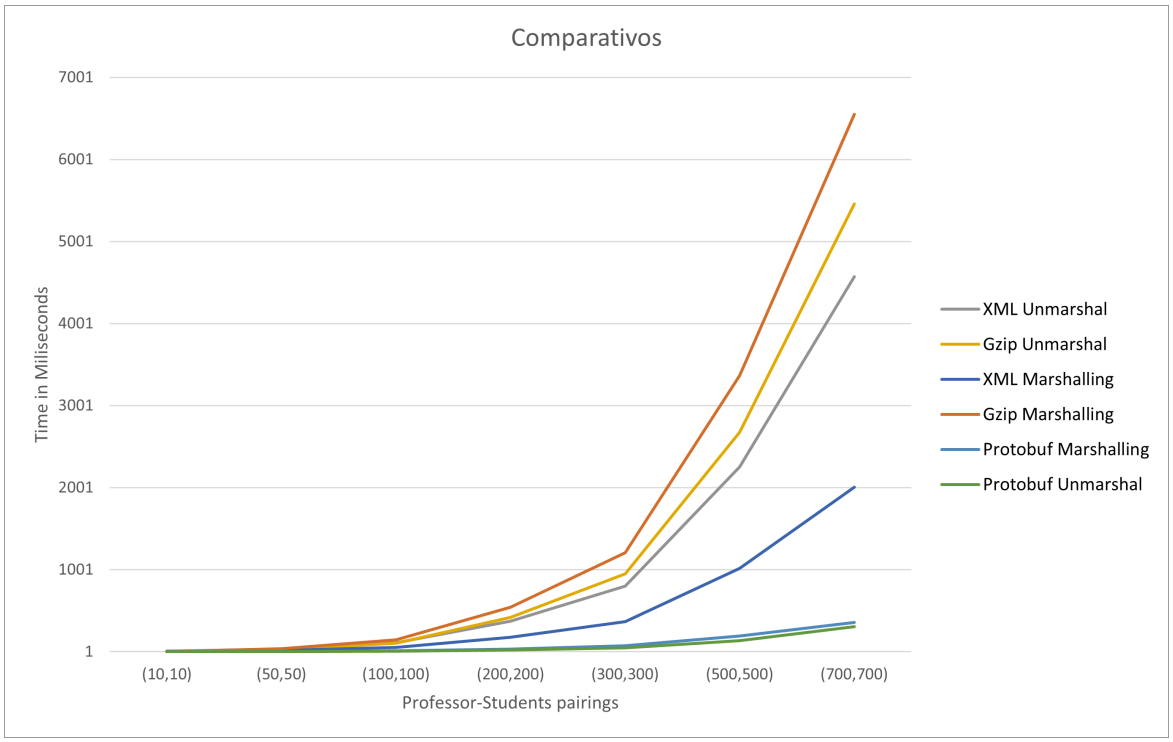
5.1.1 Complexidade de Programação

Definitivamente, o Protocol Buffers é mais complexo do que XML e XML comprimido com Gzip.

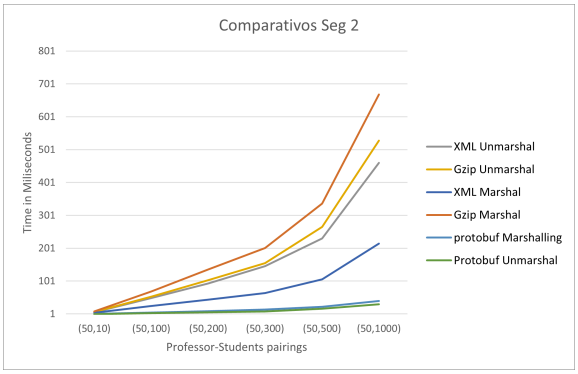
Ao contrário do XML que apenas necessita da colocação das anotações junto das classes/estruturas de dados, no Protocol Buffers é necessário vários estágios: escrever o formato das estruturas de dados no ficheiro .proto; compilar esse ficheiro de modo a gerar as classes específicas ao Protocol Buffers; mais tarde, utilizar e inicializar essas estruturas de dados especificas do Protocol Buffers, através de um mapeamento com as estruturas de dados iniciais do java.

É desprezável a diferença de complexidade de programação entre o XML e o XML comprimido com Gzip.

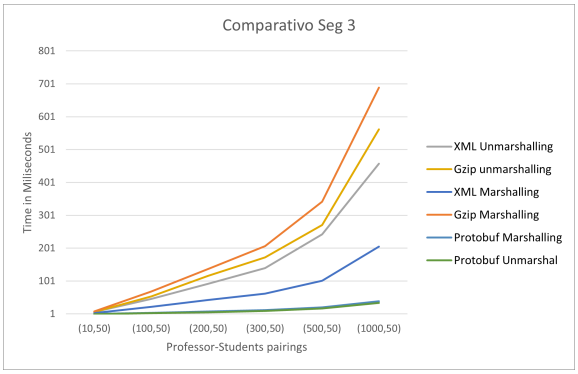
5.1.2 Serialização e Desserializiação



(a) Tempo de marshalling e unmarshalling para XML, XML comprimido com Gzip e Protocol Buffers.



(b) Tempo de marshalling e unmarshalling para XML, XML comprimido com Gzip e Protocol Buffers, no segmento 2.



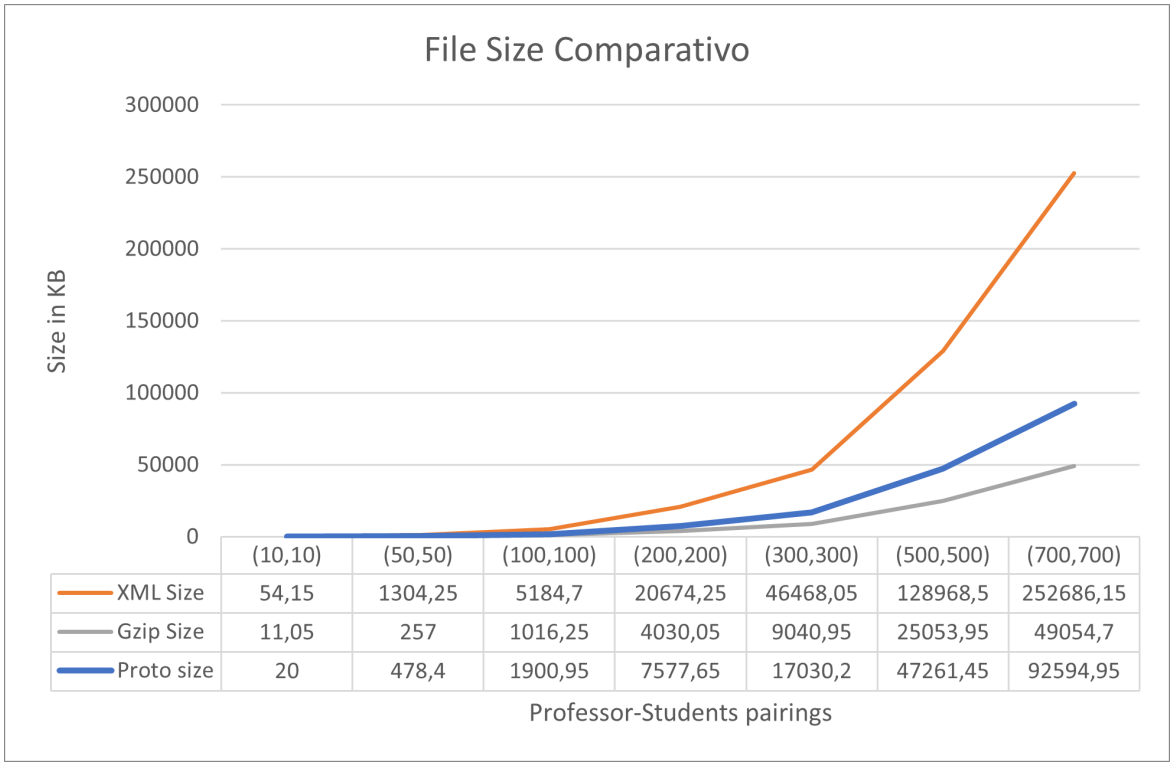
(c) Tempo de marshalling e unmarshalling para XML, XML comprimido com Gzip e Protocol Buffers, no segmento 3.

Figura 5.1: Tempos agregados para todos os métodos

Nas figuras 5.1a, 5.1b e 5.1c é fácil de verificar as diferenças de tempos no que toca ao marshalling e unmarshalling para todos os métodos. No caso do protocol buffers não foram incluídos os tempos de inicialização das estruturas, no entanto, o seu impacto não é suficiente para piorar a classificação do protocol buffers. Conseguimos ver que este, no caso em que a velocidade seja priorizada é a melhor opção, seguida de XML e por fim Gzip.

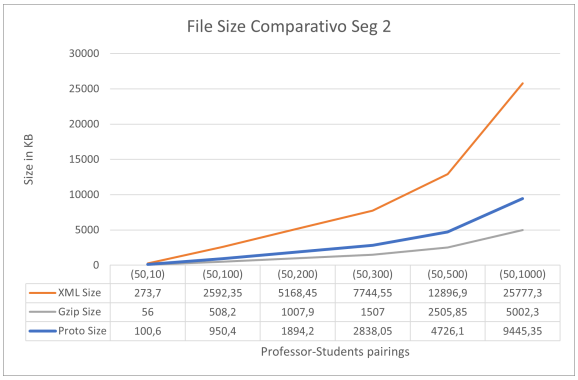
De facto, não é surpreendente que as operações de serializiação e de desserialização no XML face ao Protocol Buffers demorem em média mais tempo: enquanto que no Protocol Buffers a codificação é binária, no XML existe um passo adicional que é a codificação e descodificação em texto de atributos que podem ser dos tipos *int*, *double*, *Date*, etc., envolvendo assim *parsing* e *lookups*.

5.1.3 Tamanho



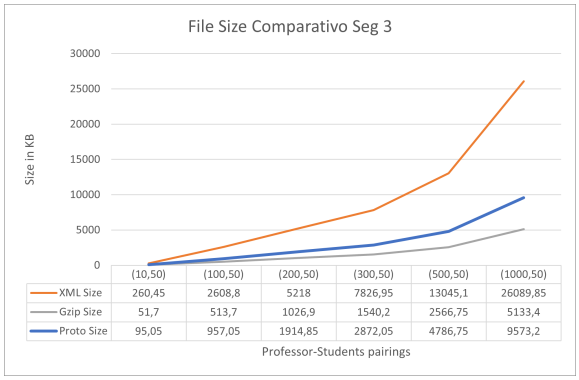
	(10,10)	(50,50)	(100,100)	(200,200)	(300,300)	(500,500)	(700,700)
XML Size	54,15	1304,25	5184,7	20674,25	46468,05	128968,5	252686,15
Gzip Size	11,05	257	1016,25	4030,05	9040,95	25053,95	49054,7
Proto size	20	478,4	1900,95	7577,65	17030,2	47261,45	92594,95

Figura 5.2: Tamanho dos ficheiros gerados pelos 3 métodos em valor absoluto, no segmento 1.



	(50,10)	(50,100)	(50,200)	(50,300)	(50,500)	(50,1000)
XML Size	273,7	2592,35	5168,45	7744,55	12896,9	25777,3
Gzip Size	56	508,2	1007,9	1507	2505,85	5002,3
Proto Size	100,6	950,4	1894,2	2838,05	4726,1	9445,35

(a) Tamanho dos ficheiros gerados pelos 3 métodos em valor absoluto, no segmento 2.



	(10,50)	(100,50)	(200,50)	(300,50)	(500,50)	(1000,50)
XML Size	260,45	2608,8	5218	7826,95	13045,1	26089,85
Gzip Size	51,7	513,7	1026,9	1540,2	2566,75	5133,4
Proto Size	95,05	957,05	1914,85	2872,05	4786,75	9573,2

(b) Tamanho dos ficheiros gerados pelos 3 métodos em valor absoluto, no segmento 3.

Figura 5.3: Tamanhos agregados para todos os métodos

Indubitavelmente, o XML não só é o pior dos três casos, como apresenta um crescimento exponencial, indicativo de não ser a melhor opção quando estamos a serializar grandes quantidades de dados. Esta característica negativa tem de ser considerada em ambientes na rede, eventualmente limitados, e ao tempo de transferência na rede.

Assim, se o tamanho é um requisito importante, o XML comprimido com Gzip é a melhor opção.

No caso do Protocol Buffer, tem um comportamento intermédio.

5.2 Vantagens e Desvantagens

Apesar de, e após a analise dos resultados, considerarmos o XML o pior dos métodos em termos de tamanho e velocidade, este não deixa de ter as suas vantagens sobre todos os outros: o XML é *text-based* e *human readable* o que significa que não são necessárias ferramentas nem operações adicionais para que algum agente possa abrir o ficheiro e confirmar o seu conteúdo, bem como efetuar alterações sobre os dados lá representados. Esta vantagem não é partilhada nem pelo protocol buffers nem pelo XML Comprimido. As desvantagens são aquelas que já foram mencionadas, nomeadamente ser mais lento no processo de marshalling e unmarshalling e o tamanho dos ficheiros ser o maior.

No caso do protocol buffers, este tem a melhor performance entre os 3 métodos testados, é o mais rápido em termos de marshalling e unmarshalling, incluindo o tempo de inicialização das estruturas, sendo que também apresenta tamanhos em disco reduzidos. Não oferece o beneficio de ser human-readable pois é *binary-based*, e devido a isso, a transmissão destes ficheiros é facilitada em termos de tempo.

No caso do XML Comprimido por gzip temos um deterioramento dos tempos de marshalling e unmarshalling mas com uma melhoria significativa ao nível do tamanho de ficheiros. Podendo então este ser bastante útil para armazenamento e transmissão.

6 Conclusão

Os objetivos foram cumpridos. Não foram sentidas quaisquer dificuldades.

Todos os formatos tem as suas vantagens conforme a relevância na sua aplicabilidade. Sucintamente, o Protocol Buffers é o mais rápido e o mais complexo em termos de programação; o XML comprimido com Gzip é o que ocupa menos espaço; o XML é interessante caso seja necessário a leitura e a escrita do mesmo por humanos.

6.1 Trabalho Futuro

Executar mais experiências que possam responder à questão levantada na exposição dos resultados (ver aqui), face ao valor alto do desvio-padrão na medição dos tempos de inicialização das estruturas dos dados inerentes ao Protocol Buffers.

7 Referências

[1] Google Developers, "Protocol Buffers: Overview". Disponível em `https://developers.google.com/protocol-buffers/docs/overview`

[2] GitHub (Google Developers), "Protocol Buffers - Google's data interchange format". Disponível em `https://github.com/protocolbuffers/protobuf`

[3] Google Developers, "Protocol Buffer Basics: Java". Disponível em `https://developers.google.com/protocol-buffers/docs/javatutorial`