UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

RAFAEL KOCH PERES

Avaliação de Formatos de Serialização sobre HTTP na Cloud

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Engenharia da Computação

Orientador: Prof. Dr. Claudio Fernando Resin Geyer

Porto Alegre
2020

AGRADECIMENTOS

Agradeço a Deus por tudo, à família pelo amor e suporte, aos amigos pela compreensão e à Universidade pela oportunidade.

RESUMO

Com o uso cada vez mais frequente da computação na nuvem e, dentro desta, microsserviços, se faz necessário a definição das interfaces expostas por estes. Parte desta definição é o formato de dados utilizados, que costuma seguir um formato padrão com suporte a múltiplas linguagens de programação, mantendo a interoperabilidade necessária para o cenário. A escolha deste formato implica diretamente no tempo de serialização e desserialização dos dados, e também no tamanho dos dados serializados. Assim, neste cenário, a comunicação entre quaisquer dois destes microsserviços está sujeita ao canal de comunicação fornecido pela nuvem e à performance do formato utilizado, que impacta também no tempo de comunicação, de acordo com o tamanho dos dados transmitidos. Dos formatos já existentes, espera-se encontrar o melhor para este cenário. Este é objetivo deste trabalho, que avalia a comunicação entre dois serviços na nuvem utilizando diferentes formatos de dados.

Palavras-chave: Serialização. formato de dados. computação em nuvem.

Evaluation of Serialization Formats over HTTP in the Cloud

ABSTRACT

As the usage of cloud computing increases and, within it, the usage of microservices, the definition of the interfaces exposed by these is necessary. Part of such definition is the data format, that usually follows a standard format definition that supports multiple languages, keeping the interoperability required in the scenario. The choice of this format implies directly over the serialization a deserialization time, and in the size of the serialized data. Therefore, in this scenario, the communication between any two of those microservices is subject to the communication channel provided by the cloud and to the performance of the used format, which also impacts in the communication time, according to the size of the transmitted data. Given the already existing formats, one should prove the best for this scenario. This is the goal of this work, which evaluates the communication between two services in the cloud using several data formats.

Keywords: Serialization, data format, cloud computing.

LISTA DE ABREVIATURAS E SIGLAS

HTTP	Hypertext Transfer Protocol
RMI	Remote Method Invocation
JSON	JavaScript Object Notation
XML	Extensible Markup Language
SDK	Software Development Kit

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

Serviços encontrados na nuvem (*cloud*) se comunicam através de mensagens. Assim como o protocolo de comunicação, existem formatos padrão que predominam nesta comunicação. Estes formatos possuem diversas análises quanto a sua eficiência, porém sem trazer esta comparação para um contexto de nuvem, propósito deste trabalho.

Ao longo deste capítulo, cobre-se a motivação, objetivo e estrutura do texto, nas seções 1.1, 1.2 e 1.3, respectivamente.

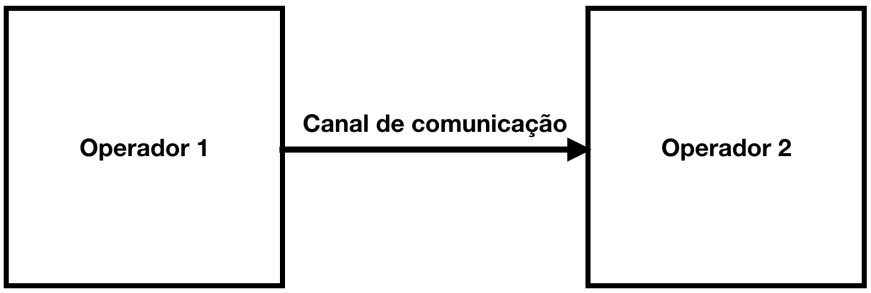
1.1 Motivação

A Arquitetura Orientada a Serviços, do inglês *Software Oriented Architecture* (SOA), é uma arquitetura de baixo-acoplamento projetada para atender às necessidades de negócio de uma organização, construída a partir de serviços que se comunicam sobre um determinado protocolo, mas sem definir nenhuma tecnologia específica (MICROSOFT, 2016; ARSANJANI, 2004). Entre as características da SOA está a interoperabilidade: a habilidade de sistemas que usam diferentes plataformas e linguagens de se comunicar (BIEBER; CARPENTER, 2001; COMMITTEE et al., 1990). Por este motivo, é esperado que uma solução em SOA acabe com implementação em múltiplas linguagens de programação para proveito de vantagens específicas que cada uma oferece. Para alcançar a interoperabilidade, os serviços que estão a se comunicar precisam estar alinhados em um protocolo e um formato de dados. Enquanto cada serviço pode se desenvolver de forma independente, é crucial que sua interface se mantenha estável (BROWN, 2008).

Baseando-se em SOA, a arquitetura de microsserviços é uma proposta mais recente, onde todos seus membros são microsserviços, ou seja, processos coesos e independentes que interagem através de mensagens. O baixo acoplamento entre estes microsserviços facilita a manutenção de cada um e também a escalabilidade da solução. Microsserviços tem seu foco na computação em nuvem, onde aplicações são implantadas sobre contêineres, que tem um custo baixo para criação e remoção, contribuindo para a escalabilidade (DRAGONI et al., 2017).

A computação em nuvem está crescendo rápido e constantemente (COSTELLO, 2019), e, com esta, o uso de microsserviços. Por isso, deseja-se saber o impacto na eficiência de uma solução quando escolhido um formato para as mensagens comunicadas entre os serviços.

Figura 1.1: Exemplo de grafo



Fonte: O Autor

Durante o desenvolvimento de sua atividade profissional, o autor desenvolve uma aplicação de definição de grafos, onde cada vértice é um operador e as arestas são os canais de comunicação entre estes operadores, como visto na figura 1.1. Operadores são microsserviços já implementados com uma funcionalidade específica, como, por exemplo, a leitura de arquivos de um serviço da nuvem. Estes são implantados em contêineres, onde a aplicação é responsável por criar o canal de comunicação entre eles. Para esta aplicação, diferentes formatos para as mensagens foram considerados, mas uma análise profunda se mostra necessária, sendo também o propósito deste trabalho.

1.2 Objetivo

O objetivo deste trabalho é analisar a eficiência de diferentes formatos de serialização utilizados na comunicação entre serviços na nuvem. A avaliação deve permitir distinguir entre diferentes cenários presentes na nuvem, contando com o serviço utilizado, sua disposição geográfica, e o volume e tipo de dados comunicados, visando apontar o melhor formato para cada um destes. Ainda, a avaliação deve apontar o impacto da escolha de um formato para estes cenários, sendo que a escolha de um formato poderá ser não relevante frente a outros fatores.

1.3 Estrutura do texto

Este trabalho está dividido em seis capítulos, sendo este capítulo o primeiro. O capítulo 2 apresenta conceitos básicos sobre formatos de dados e sua serialização, e sobre

a computação em nuvem, relevantes para o entendimento deste trabalho. O capítulo 3 expõe trabalhos relacionados que servem de base e complemento para o desenvolvido neste, comparando os resultados apresentados. O capítulo 4 apresenta a metodologia para os experimentos realizados. O capítulo 5 contém os resultados dos restes e sua análise. Por fim, o capítulo 6 apresenta as conclusões gerais sobre os resultados e o trabalho desenvolvido, bem como possíveis extensões para .

2 CONCEITOS BÁSICOS

Este capítulo apresenta conceitos básicos necessários para a contextualização e entendimento deste trabalho. Na seção 2.1, trata-se do conceito de serialização, alvo de análise deste trabalho. Na seção 2.2, aborda-se a computação em nuvem, ambiente onde esta análise será realizada.

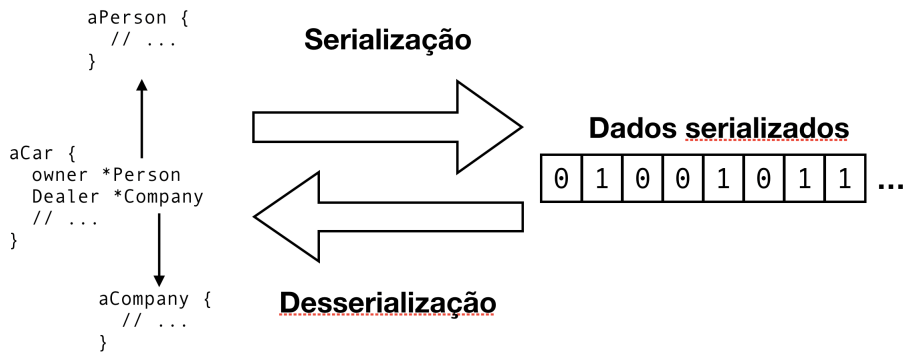
2.1 Serialização

Sob o ponto de vista da análise deste trabalho, vale notar que se adota o termo "serialização" de forma a abranger o processo de serializar e desserializar os dados, assim como o formato dos mesmos quando serializados.

Serializar é o processo de transformar objetos, ou estruturas de dados, em uma sequência de bytes, para que possam ser salvos em disco ou transportados pela rede (MICROSOFT, 2018). O processo inverso, resultando novamente em um objeto, é conhecido como desserialização. Outras nominações para este processo também são *marshalling* e *flattening* – referindo-se a dados complexos e estruturados que se tornam planos. Como visto na figura 2.1, à esquerda, temos um objeto em memória que é serializado em uma sequência contígua de bytes, encontrada à direita. O processo na direção inversa também é possível, constituindo a desserialização.

O formato de serialização define o estado dos dados serializados. O processo de serializar e desserializar pode ou não ser definido pelo formato. Mesmo quando não definido, os requisitos do formato afetam diretamente a sua implementação.

Figura 2.1: Serialização



Fonte: O Autor

A escolha de um formato impacta tanto na eficiência espacial – tamanho resultante dos dados serializados – quanto na temporal – tempo levado no processo de serialização e desserialização. Eficiência não é o único fator, características qualitativas também devem ser levadas em consideração de acordo com o contexto, a exemplo do requisito de interoperabilidade em um determinado cenário.

Usualmente, uma linguagem suporta a serialização de seus objetos de forma nativa, como é o caso de Java (ORACLE, 2017), C# (MICROSOFT, 2018) e várias outras. Porém a serialização binária nativa é um obstáculo para a interoperabilidade, ou seja, ao se serializar em uma linguagem, é difícil ou impossível desserializar o resultado em outra. Um contraexemplo é JavaScript, onde o resultado da serialização nativa é em JSON (MOZILLA, 2017), formato intercambiável e um dos analisados neste trabalho, explicado em detalhes na seção 4.3.1.

Ademais, um formato de dados pode ser textual, assim sendo legível para humanos, porém mais restrito quanto a possíveis representações, ou em bytes (não textuais), tendo total liberdade para definir sua representação dos dados em bytes. Há também diferentes níveis de expressividade, e a necessidade ou não de um *schema*, que define formalmente a estrutura dos dados serializados em tempo de compilação ou execução.

Compilando os fatores apresentados, serialização engloba:

- Formato dos dados serializados;
- Processo de serializar os dados;
- Processo de desserializar os dados.

O formato de serialização conta com dois fatores de eficiência:

- Eficiência temporal;
- Eficiência espacial.

Além de contar com outras características qualitativas, das quais se destacaram:

- Interoperabilidade (número de linguages suportadas);
- Binário ou textual;
- Necessidade de um *schema*.

2.2 Computação em nuvem

A definição de computação em nuvem, ou *cloud computing*, varia entre autores e é razão de confusão. Para este trabalho, adota-se a definição dada pelo NIST (MELL; GRANCE et al., 2011), em tradução livre: "um modelo que permite acesso ubíquo, conveniente, sob demanda para um conjunto comum de recursos computacionais configuráveis (e.g., redes, servidores, armazenamento de dados, aplicações e serviços) que podem ser rapidamente provisionados e dispensados com um mínimo esforço de gerenciamento ou interação do provedor do serviço." Esta definição ainda trás um conjunto de 5 características, 3 modelos de serviço e 4 modelos de implantação.

Este trabalho considera apenas a nuvem pública, um dos 4 modelos de implantação, que é de acesso geral e controlada pelos provedores de Internet. As 5 características são expostas na seção 2.2.1, e os 3 modelos de serviço na seção 2.2.2.

2.2.1 Características

Segundo Mell, Grance et al. (2011), caracterizam a computação em nuvem:

- Autosserviço sob demanda (*on-demand self-service*): o consumidor provisiona capacidades computacionais unilateralmente, como tempo de servidor e capacidade de armazenamento, sem necessitar de ação humana pelo provedor de serviço;
- Acesso amplo à rede (*broad network access*): capacidades do serviço são disponibilizadas pela rede e acessíveis por mecanismos padrão e heterogêneos, como telefones celular, tablets e laptops;
- *Pooling* de recursos (*resource pooling*): recursos computacionais físicos e virtuais do provedor são alocados e realocados sob-demanda, a partir de uma *pool* de recursos, segundo um modelo *multi-tenant*. O consumidor não tem controle ou conhecimento sobre a localização exata dos recursos alocados, opcionalmente trabalhando com um conceito mais alto nível, como país ou região;
- Elasticidade rápida (*rapid elasticity*): recursos são alocados e liberados elasticamente, podendo até ocorrer automaticamente, para que escalem proporcionalmente à demanda recebida. Dão ao consumidor uma impressão de recursos ilimitados, sem limitação de quantia ou de hora de acesso;
- Serviço mensurado (*measured service*): os sistemas cloud controlam e otimizam o

uso de recursos ao proporcionarem alguma capacidade de medida de acordo com o tipo de serviço fornecido. O uso de recursos pode ser monitorado, controlado e reportado, gerando transparência para o provedor e para o consumidor.

Como exemplo dessas características, tem-se o Google Cloud Storage[1] e o Amazon S3[2], ambos serviços de armazenamento na nuvem. Estes permitem a qualquer momento, de forma automática, alocar um espaço para armazenamento de arquivos na nuvem. Esta alocação é realizável através de diversas plataformas, tanto por interação humana através de um navegador, quanto por interfaces programáticas. O consumidor é cobrado apenas pelo tráfego de dados e espaço utilizado, podendo liberar ou alocar espaço adicional a qualquer momento. Os gastos gerados são visíveis através de relatórios e gráficos em uma aplicação de cobrança, que pode ser desabilitada a qualquer momento, cancelando os serviços fornecidos.

2.2.2 Modelos de serviço

Serviços de computação disponibilizados na nuvem se dividem em três modelos, baseados no nível de customização da infraestrutura fornecido ao consumidor, servindo propósitos diferentes, como especificado por Mell, Grance et al. (2011):

- *Software as a Service* (SaaS): é provido ao consumidor através de uma infraestrutura na nuvem, sendo acessível através de clientes, como um navegador, ou através de interfaces programáveis. O consumidor não tem controle da infraestrutura, rede, sistema operacional ou armazenamento subjacente, nem de capacidades individuais da aplicação, com exceção das configurações específicas para usuário expostas;
- *Platform as a Service* (PaaS): o consumidor é provido da capacidade de implantar na nuvem suas aplicações, desde que utilizem as linguagens e outras ferramentas suportadas pelo provedor. O consumidor não controla a infraestrutura subjacente, mas tem controle sobre a aplicação e possivelmente de configurações para o seu ambiente;
- *Infrastructure as a Service* (IaaS): provê ao consumidor recursos computacionais na nuvem, como infraestrutura de processamento, rede e/ou armazenamento, onde se pode implantar softwares arbitrários. O consumidor não controla a infraestru-

[1] https://cloud.google.com/storage
[2] https://aws.amazon.com/s3

tura subjacente, mas tem controle sobre seu sistema operacional, armazenamento e aplicações, podendo também ter algum controle limitado sobre componentes da rede.

Neste trabalho, utiliza-se o Google App Engine[3], serviço de PaaS que provisiona recursos para serviços Web (GOOGLE, 2019a). O serviço, junto de outros disponibilizados pelo Google Cloud Platform, possui um período gratuito de testes de 12 meses, com uma cota de uso de US$ 300,00, que é descontada de acordo com o uso dos serviços (GOOGLE, 2019b). Percebe-se, com isso, a característica de um serviço mensurado. A alocação de recursos deste ainda não necessita de qualquer ação humana do provedor, ocorrendo automática e instantaneamente, sendo configurada através do navegador de um laptop, mas também é acessível através de outros dispositivos, caraterizando um autosserviço sob demanda com um amplo acesso à rede. Durante os testes, diversas instâncias dos serviços desenvolvidos neste trabalho foram implantadas e desativadas, indicando uma elasticidade rápida do provedor. Ainda, a localização do serviço implantando foi apenas especificada em alto nível, sem dar a noção de uma localização física específica, completando a característica de *pooling* de recursos. Detalhes técnicos da utilização do serviço são expostos na seção 4.4.

[3]https://cloud.google.com/appengine/

3 TRABALHOS RELACIONADOS

Neste capítulo são apresentados trabalhos que avaliam formatos de serialização diversos na seção 3.1, como por exemplo XML, JSON e Protobuf. No total, três diferentes trabalhos são abordados neste capítulo.

3.1 Análise local de formatos de serialização

Partindo do protocolo Java RMI, que utiliza a serialização nativa e exclusiva à linguagem, Maeda (2012) desconfiou de sua performance. Em seu trabalho, trouxe uma comparação entre a serialização padrão Java para com outros formatos, apontando também a necessidade de um formato que suporte diversas linguagens de programação num ambiente heterogêneo como a Internet. O autor analisou o tamanho resultante dos dados serializados e o tempo de execução da serialização e desserialização através um objeto em Java de tamanho variável.

Os testes foram realizados num sistema operacional Mac OS X, sobre um computador iMac (by Apple Inc.) com Intel Core2 Duo 2.66 GHz e 2 GB de memória, utilizando a linguagem Java. Foram avaliados os formatos textuais JSON, Numbered JSON e XML, e os formatos binários Java nativo, Thrift, Avro e Protocol Buffers, incluindo múltiplas bibliotecas de serialização para cada formato, quando disponíveis. Para os testes realizados, usou-se um objeto Pedido (*Order*) com uma série de atributos de tipos primitivos, além de um atributo como uma lista de Opções (*Option*), uma segunda classe com uma série de atributos de tipos primitivos. Entre os testes realizados, foi serializado o objeto Pedido, variando-se o tamanho da lista de 0 até 900 itens, avançando-se de 100 em 100 (i.e. 0, 100, 200, ..., 900).

Na tabela 3.1 são apresentados os resultados em relação ao espaço. Avro apresenta o melhor resultado na média geral, e o segundo melhor quando o número de opções é zero. Protobuf é o segundo melhor na média, mas o mais eficiente para zero opções. Também fica claro o impacto na performance trazido pelos dois formatos textuais, XML e JSON, quando comparados aos outros formatos binários. Ainda assim, JSON utilizou menos do que o dobro do espaço do XML, na média. Ainda, é importante destacar o alto valor resultante para a serialização nativa de Java com zero opções, que foi menos eficiente do que todos os outros formatos em consequência dos metadados serializados junto dos dados. Com relação ao tempo de serialização e desserialização, visto nas tabelas

3.2 e 3.3, e considerando apenas a biblioteca mais eficiente para cada formato, XML ainda apresentou os piores resultados com grande margem. Protobuf foi o mais eficiente, seguido da biblioteca Jackson para JSON em ambos os casos (Maeda, 2012).

Tabela 3.1: Tamanho serializado em bytes

	Média (*B*)	Sem opções (*B*)
XStream in XML	36973.6	160
JSON libraries	15307.9	106
Numberd JSON	12124.2	66
Thrift in binary	11328.0	78
Object Serialization in Java	10746.6	348
Protobuf	7585.4	47
Avro in binary	5844.1	49

Fonte: Maeda (2012)

Tabela 3.2: Tempo em de serialização em *ms*

	Média (*ms*)
XStream in JSON	9.7858
JsonLib	2.9422
XStream in XML	2.8698
FlexJson	0.9740
ThriftJson	0.7422
Gson	0.4756
JsonMarshaller	0.4094
Jsonic	0.3044
Object serialization in Java	0.2606
JsonSmart	0.2328
AvroJson	0.2278
Thrift in binary	0.1654
Avro in binary	0.1672
ProtoStuff with static schema	0.1750
ProtoStuff with dymaic schema	0.1532
Jackson	0.1488
Protobuf	0.0476

Fonte: Maeda (2012)

No trabalho do autor Sumaray e Makki (2012), avaliaram-se diferentes formatos de serialização em plataformas *mobile*, onde foram abordados dois formatos textuais, XML e JSON, e dois formatos binários, Thrift e Protocol Buffers. A análise se deu sobre a serialização de dois diferentes objetos: *Book* e *Video*. A primeira classe teve um foco em atributos textuais, enquanto a segunda teve um maior número de atributos numéricos.

Tabela 3.3: Tempo em de desserialização em *ms*

	Média (*ms*)
XStream in JSON	22.112
JsonLib	5.1280
XStream in XML	4.2166
FlexJson	1.4978
ThriftJson	0.9874
Gson	0.9038
JsonMarshaller	0.4036
Jsonic	0.3638
Object serialization in Java	0.2800
JsonSmart	0.2098
AvroJson	0.1922
Thrift in binary	0.1710
Avro in binary	0.1626
ProtoStuff with static schema	0.1316
ProtoStuff with dymaic schema	0.1058
Jackson	0.0996
Protobuf	0.0020

Fonte: Maeda (2012)

Os testes foram desenvolvidos no ambiente de programação do Android SDK.

Os resultados são apresentados nas tabelas 3.4, 3.5 e 3.6. Eles mostraram uma grande diferença, favorecendo os dois formatos binários, Thrift e Protocol Buffers, sobre os dois formatos textuais, JSON e XML. De forma geral, Protocol Buffers se mostrou o mais eficiente, mas Thrift manteve resultados próximos, sendo o mais eficiente apenas no tempo de serialização. Entre os formatos textuais, JSON foi o mais eficiente em todos os aspectos, destacando-se pela diferença no tempo de serialização e desserialização (SUMARAY; MAKKI, 2012).

Tabela 3.4: Tamanho serializado em bytes

	Book	Video
XML	873	231
JSON	781	139
Protobuf	687	59
Thrift	720	92

Fonte: Sumaray e Makki (2012)

Num contexto de Internet das Coisas, focando em padrões já utilizados em *Smart Grids* e suas respectivas limitações, Petersen et al. (2017) avaliou formatos de serialização

Tabela 3.5: Tempo médio de serialização em *ms*

	Book	Video
XML	22.842	17.884
JSON	4.177	4.097
Protobuf	2.339	1.800
Thrift	2.315	1.747

Fonte: Sumaray e Makki (2012)

Tabela 3.6: Tempo médio de desserialização em *ms*

	Book	Video
XML	7.908	6.742
JSON	1.199	0.755
Protobuf	0.298	0.197
Thrift	0.732	0.310

Fonte: Sumaray e Makki (2012)

e respectivas bibliotecas Java. Os testes foram executados num computador com processador Intel Duel Core 2.1GHz, com 8GB de memória e sistema operacional Windows 10, com mensagens de teste seguindo o modelo de dados IEC 61850, permitindo aos testes se aproximarem de mensagens reais de um Smart Grid. Os resultados foram obtidos de uma média de 1000 medidas para cada formato. Além do tamanho resultante dos dados serializados e o tempo de serialização e desserialização, o trabalho também examinou o impacto da compressão dos dados após sua serialização. Nos resultados apresentados nas tabelas 3.7 e 3.8, foram apresentados apenas os formatos de interesse dentre os apresentados por Petersen et al. (2017). Nestes, Protocol Buffers apresentou o melhor tempo de serialização e desserialização. MessagePack e Avro demonstraram tamanho resultante menor, com Protocol Buffers como o terceiro menor, todos com valores próximos. Os formatos textuais exibiram a pior performance, ainda com JSON superando XML em todos os aspectos.

Tabela 3.7: Tamanho serializado em bytes

XML	12387
JSON	5918
Protobuf	2870
Avro	2446
MsgPack	2217

Fonte: Petersen et al. (2017)

Tabela 3.8: Tempo de processamento das mensagens (μs)

	Serialiazação	Desserialização
XML	72	110
JSON	51	55
Protobuf	18	16
Avro	179	155
MsgPack	107	99

Fonte: Petersen et al. (2017)

4 METODOLOGIA

Neste capítulo, a seção 4.1 introduz a metodologia dos experimentos, esclarecendo seu objetivo e expondo os fatores analisados. A seção 4.2 enumera os diferentes tipos de dados a serem transmitidos nas mensagens dos experimentos. A seção 4.3 apresenta os formatos analisados, apontando otimizações disponíveis de cada formato que poderão afetar o resultado dos experimentos. A seção 4.4 esclarece como os experimentos serão realizados, a fim de cobrir todos os formatos e tipos propostos, apresentando a disposição na qual serão executados, como serão extraídas as medidas e com qual amostragem.

4.1 Visão geral

No âmbito de dois serviços na nuvem se comunicando sobre o protocolo HTTP, seguindo as premissas de SOA, um serviço não tem conhecimento da implementação do outro, mas apenas da sua interface disponibilizada. Para que dados sejam comunicados através desta interface, um formato interoperável é adotado, como JSON ou XML (BRAY, 2017; W3C, 2016).

Neste cenário, há interesse em saber o impacto dos diferentes formatos na performance da comunicação entre estes dois serviços. Para a escolha deste formato num cenário real, também se leva em conta suas limitações, como tipos representáveis e a necessidade ou não de uma definição da estrutura dos dados comunicados em tempo de compilação.

Avalia-se apenas as características quantitativas de performance dos formatos selecionados quando se comunicando como serviços na nuvem, a fim de os resultados servirem parâmetro na escolha do formato em casos reais.

Similar aos trabalhos já apresentados no capítulo 3, são tomados como fatores o tamanho dos dados serializados e o custo da serialização e desserialização. Enquanto o tamanho não sofre influências externas, dependendo apenas dos dados a serem serializados, o custo de serialização e desserialização sofre com os recursos de hardware disponíveis e a biblioteca utilizada para sua conversão, que nem sempre é padronizada.

Porém, a comunicação dos serviços é feita pela Internet, e o tempo desta é afetado pelo tamanho da mensagem transportada. Logo, o tempo de comunicação também se torna um fator de interesse para análise. Portanto, espera-se que, quanto maior o tamanho em bytes da mensagem transportada, maior o tempo de comunicação. Além disso, a

Figura 4.1: Tempo total de comunicação

Latência + Serialização + Transporte = Total

Fonte: O Autor

latência inicial e o *throughput* variam com distância, tráfego e qualidade do canal entre os dois serviços.

Assim, temos três fatores de custo a avaliar sobre o uso de um formato na comunicação entre dois serviços: o custo de serialização e desserialização, o tamanho resultante dos bytes a serem transmitidos, e o tempo total levado nesta comunicação. O tempo total é, sumariamente, dado segundo a fórmula da figura 4.1. Dos três componentes da soma, apenas a serialização não é afetada pelo canal de comunicação; a latência depende apenas do canal de comunicação entre os pares, e o tempo de transporte depende tanto do *throughput* do canal como da quantia de dados a serem transportados proveniente da serialização.

4.2 Tipos de dados

Para melhor explorar otimizações de um formato para tipos específicos, os experimentos são executados sobre os tipos mais comuns, especificamente: números inteiros, números em ponto flutuante e textos (strings). Os experimentos também cobrem estruturas de dados complexas. Todos estes são apresentados na tabela 4.1 e detalhados nas seções 4.2.1, 4.2.2, 4.2.3, 4.2.4, respectivamente. Os exemplos dados utilizam a sintaxe da linguagem Go[1], utilizada na implementação dos experimentos.

Tabela 4.1: Tipos analisados

Tipo em Go	Descrição
$int64$	Números inteiros em 64 bits
$float64$	Números em ponto-flutuante em 64 bits
$string$	Texto
$struct$	Estrutura de dados complexa

Fonte: O Autor

[1] https://golang.org/

4.2.1 Números inteiros

Para a representação de números inteiros, utilizou-se o tipo `int64`, isto é, número inteiro de 64 bits. Para este tipo, valores aleatórios foram gerados cobrindo todo seu intervalo, utilizando a biblioteca padrão de Go para números aleatórios `math/rand`.

Em Go, tipos inteiros com sinal também são representados em 8, 16 e 32 bits através dos tipos `int8`, `int16` e `int32` respectivamente. Estes números tem grande potencial de otimização quando seu valor é correspondido por intervalos menores, e.g., o número 127 é representável por `int64` que utiliza 16B, mas também é pelo tipo `int8`, que utiliza apenas 1B.

4.2.2 Números em ponto-flutuante

A linguagem Go suporta a representação de números em ponto-flutuante de 32 bits e de 64 bits especificados pelo padrão IEEE-754 (IEEE, 2019). A conversão entre estes, apesar de suportada pela linguagem, não é trivial e pode gerar perda de precisão. De forma geral, o tipo `float64` predomina na representação de números em ponto-flutuante, sendo o utilizado nos experimentos.

Os números gerados para os experimentos seguem uma distribuição normal no intervalo do mínimo ao máximo representáveis pelo tipo, como apresentado na função `NormFloat64` da biblioteca padrão de números aleatórios de Go.

4.2.3 Texto

Para a representação de texto, utilizou-se o tipo `string` de Go, que não se restringe a uma codificação específica para representação dos caracteres, mas segue a representação em UTF-8 convencionalmente. Para este tipo, foram geradas sequências de 100 caracteres alfabéticos, maiúsculos e minúsculos. Todos estes, segundo a codificação UTF-8, utilizam apenas 1 byte para sua representação (YERGEAU, 2003).

Como a representação destes caracteres utilizados já está restrita a 1B, sem considerar compressões sobre sequências de caracteres, não há possibilidade para otimização. Por isso, para este tipo de dado, espera-se um resultado próximo entre formatos binários e textuais, ao contrário de outros tipos, onde formatos binários possuem vantagens em

Figura 4.2: `struct` em Go em uso nos experimentos

```
struct {
    I int64
    F float64
    T bool
    S string
    B []byte
}
```

Fonte: O Autor

relação ao espaço em bytes necessário para representação.

4.2.4 Estruturas complexas

Como maior parte dos dados reais são compostos e complexos, uma avaliação de estruturas complexas também é necessária. Para estes, utilizou-se o tipo `struct` de Go, contendo uma propriedade para cada um dos tipos vistos e também para valores booleanos (`bool`) e valores puramente binários (`[]byte`), como visto na figura 4.2. O mesmo método de valores aleatórios foi seguido com os tipos já expostos anteriormente. Para a propriedade `bool`, utilizou-se o método de números inteiros aleatórios, onde pares resultaram no valor `true`, e ímpares em `false`. Com relação ao tipo `[]byte`, foi usado o método de string aleatória, aplicando-se a devida conversão.

4.3 Formatos analisados

Esta seção apresenta os formatos analisados neste trabalho. Os formatos escolhidos originam-se dos mais usuais entre os trabalhos pesquisados. Também foi incluído o formato *MessagePack*, utilizado pelo autor no contexto específico que motivou este trabalho. Dos formatos textuais, apenas JSON foi considerado por ser o melhor segundo os trabalhos relacionados, e para se ter um parâmetro de comparação de formatos textuais para com os formatos binários testados.

Dos formatos, alguns requerem uma definição de *schema* para serializar e desserializar uma mensagem, isto é, uma pré-definição da estrutura dos dados da mensagem. Para exemplos de schemas e valores nesta seção, utiliza-se a estrutura apresentada na figura 4.3.

Desta forma, distinguem-se os formatos quanto aos seguintes fatores:

Figura 4.3: Exemplo de `struct` em Go

```
type Dish struct {
    Name string
    Ingredients []string
    Serves int64
    Weight float64
    Veggie bool
}

var exampleDish = Dish{
    Name: "Mashed Potatoes",
    Ingredients: []string{"4 Potatoes", "Salt"}
    Serves: 3,
    Weight: 0.6,
    Veggie: true,
}
```

Fonte: O Autor

- Textual ou binário
- Possibilidade e/ou necessidade de *schema*
- Origem da biblioteca Go (nativa, oficial ou de terceiros)

Os formatos analisados são:

1. JSON
2. Protocol Buffers (Protobuf)
3. Apache Avro
4. MessagePack

Estes formatos são apresentados nas seções 4.3.1, 4.3.2, 4.3.3 e 4.3.4, respectivamente.

4.3.1 JSON

JSON, ou JavaScript Object Notation, é um formato textual baseado em um subconjunto da linguagem JavaScript. Apesar disso, é um formato independente de linguagem, de fácil leitura e escrita para humanos, mas também de processamento simples para máquinas (BRAY, 2017). Possui duas estruturas básicas:

1. Vetor: uma coleção ordenada de valores;
2. Objeto: um conjunto não-ordenado de pares nome/valor.

Figura 4.4: Exemplo de objeto em JSON

```
{
    "Name": "Mashed Potatoes",
    "Ingredients": ["4 Potatoes", "Salt"],
    "Serves": 3,
    "Weight": 0.6,
    "Veggie": true
}
```

Fonte: O Autor

Valores podem ser qualquer um dos tipos básicos: string, número, booleano (`true` ou `false`) ou nulo (`null`). Também podem ser um objeto ou vetor, permitindo aninhamentos. Por padrão, JSON não possui uma definição de schema para as estruturas representadas. Segundo o valor exemplo, o equivalente em JSON é apresentado na figura 4.4.

Para serialização e desserialização dos experimentos, foi utilizada a biblioteca padrão de Go para JSON (GO, 2018).

4.3.2 Protocol Buffers (Protobuf)

Inicialmente introduzido na Google para resolver um problema de versionamento de mensagens e para facilitar o uso destas entre múltiplas diferentes linguagens, Protocol Buffers, ou Protobuf na sua forma mais breve, é um formato de serialização para comunicação ou armazenamento de dados, que utiliza arquivos `.proto` com a definição dos schemas das mensagens a serem serializadas ou desserializadas (Protocol Buffers, 2018). A partir destes arquivos, é possível gerar código para diversas linguagens para que possam processar a mensagem especificada. A geração de código para uma mensagem numa linguagem específica permite maior eficiência no tratamento desta, ao invés de tratar a estrutura da mensagem em tempo de execução.

Protocol Buffers, na data deste trabalho, é suportado nas versões 2 e 3. Por ser a mais recente, é considerada apenas a versão 3. Na definição do schema das mensagens, é obrigatório especificar a versão utilizada, e também cada propriedade com seu índice e tipo. O uso de índices permite adicionar mais propriedades à mensagem mantendo-a compatível com versões antigas, desde que índices já utilizados não sofram alterações. Os tipos disponíveis para cada propriedade possuem uma grande gama, como visto em Language... (2019). A tabela 4.2 apresenta a equivalência entre os tipos de Go já apresen-

Figura 4.5: Exemplo de schema em Protobuf

```
syntax = "proto3";

message dish {
    string name = 1;
    repeated string ingredients = 2;
    int64 serves = 3;
    double weight = 4;
    bool veggie = 5;
}
```

Fonte: O Autor

tados e os tipos deste formato disponíveis. Note que para o tipo `int64` em Go, existem duas possibilidades em Protobuf:

- `int64`: inteiro de 64 bits com tamanho variável, ineficiente para números negativos.
- `sint64`: inteiro de 64 bits otimizado para número negativos. Este tipo foi o escolhido para avaliação de inteiros neste trabalho, já que os números gerados podem ser negativos.

Tabela 4.2: Mapeamento de tipos em Go para Protobuf

Tipo em Go	*Tipos em Protocol Buffers*
`float64`	`double`
`int64`	`int64`, `sint64`
`string`	`string`
`bool`	`bool`
`[]byte`	`bytes`

Fonte: Language... (2019)

O arquivo `.proto`, necessário para representar a estrutura de exemplo, se encontra na figura 4.5. Note que para a propriedade `serves`, foi escolhido o tipo `int64`, já que são esperados valores positivos, dada sua semântica (números de pessoas servidas pelo prato). A definição de uma mensagem ainda requer um nome como identificador, no exemplo, `dish`. O uso de um identificador permite a reutilização desta mensagem dentro de outras, como para definir uma lista de pratos com um respectivo *chef*, sem necessitar redefinir a estrutura de um prato.

Para os valores resultantes do exemplo, a figura 4.6 apresenta-os preenchidos na sua opção textual. Note que a opção textual, útil para casos de debug e testes, é uma

Figura 4.6: Exemplo de objeto em Protobuf

```
dish{
    name: "Mashed Potatoes"
    ingredients: ["4 Potatoes", "Salt"]
    serves: 3
    weight: 0.6
    veggie: true
}
```

Fonte: O Autor

Figura 4.7: Exemplo de schema em Avro

```
{
    "type": "record",
    "name": "Dish",
    "fields" : [
        {"name": "Name", "type": "string"},
        {"name": "Serves", "type": "long"},
        {"name": "Weight", "type": "double"},
        {"name": "Veggie", "type": "boolean"}
    ]
}
```

Fonte: O Autor

alternativa menos eficiente à opção padrão binária. A opção binária, que é a padrão, possui uma definição diferente de codificação para cada tipo de dado. Por exemplo, o tipo `int64` utiliza uma codificação de *varints*, ou seja, otimizada para representação de números inteiros de precisão variada, onde cada byte tem seu primeiro bit destinado à identificar o fim de uma sequência de bytes que representam o número, enquanto os sete restantes são utilizados para o próprio valor numérico, assim permitindo um tamanho variável de bytes na representação.

4.3.3 Avro

Mantido pela Apache Software Foundation, Apache Avro, ou simplemente Avro, é um formato de serialização similar a Protocol Buffers e Thrift. Apesar de também usar um schema de definição, se diferencia por não necessitar de geração de código, já que a definição do schema sempre acompanha o dado serializado. A definição do schema usa o formato JSON, exposto na seção 4.3.1, conforme visto no exemplo da figura 4.7. O valor serializado em Avro é sempre binário, não possuindo uma alternativa textual, por isso, não são dados exemplos para os valores serializados (AVRO, 2012).

Os tipos primitivos disponíveis em Avro são mapeados para os utilizados em Go segundo a Tabela 4.3. Ao contrário de Protobuf, não há mais de um tipo Avro que represente um mesmo tipo primitivo de Go, resultando em um mapeamento um para um. Para a definição de estruturas de dados, utilizam-se `records`, como visto na figura 4.7.

Tabela 4.3: Mapeamento de tipos em Go para Avro

Tipo em Go	*Tipos em Avro*
`float64`	`double`
`int64`	`long`
`string`	`string`
`bool`	`boolean`
`[]byte`	`bytes`

Fonte: O Autor

4.3.4 MessagePack

MessagePack, ou MsgPack, é um formato de serialização binário baseado em JSON, e, por isso, herda muitas de suas propriedades, das quais se destacam:

- Não necessitar de um schema para definir seus dados.
- Utilizar os mesmos tipos de JSON, porém dividindo números entre suas representações em inteiros de tamanho variável e ponto flutuante de 64 bits.

O formato também permite "tipos de extensão", que suportam a definição de um tipo arbitrário. O tipo `timestamp`, que define uma medida de tempo (i.e., data, hora e fuso-horário), é o único tipo de extensão definido por padrão pelo formato (FURUHASHI, 2013).

Oficialmente, são mantidas apenas a definição do formato e bibliotecas para um pequeno conjunto de linguagens. Em Go, apenas bibliotecas de terceiros estão disponíveis para serialização e desserialização em MsgPack. A página oficial aponta para cinco bibliotecas diferentes, das quais foi escolhida a `ugorji/go`[2], por ser a mais madura, apresentando melhor documentação includindo benchmarks, e ser mantida ativamente até a data deste trabalho.

Como MsgPack é puramente binário, não se tem um exemplo legível, porém a estrutura resultante é similar à apresentada para JSON na figura 4.4.

[2] https://github.com/ugorji/go

4.3.5 Comparação dos formatos

Esta seção compara aspectos qualitativos dos formatos apresentados. A comparação se baseia na tabela 4.4. Para a linha da tabela "Total de linguagens suportadas", os valores foram baseados nas linguagens apresentadas pela documentação oficial de cada formato, considerando tanto bibliotecas oficiais quanto terceiras apontadas nestes documentos.

Tabela 4.4: Comparação dos formatos analisados

max width=

	JSON	Protobuf	Avro	MsgPack
Textual ou Binário	Textual	Ambos	Binário	Binário
Schema	Não suportado	Obrigatório	Opcional	Não suportado
Biblioteca Go	Nativa	Oficial	Terceira	Terceira
Total de linguagens suportadas	59	6	7	40
Mantido por	ECMA International	Google Developers	Apache Software Foundation	MessagePack™

Fonte: O Autor

Apesar de Protobuf suportar formatos ambos binário e textual, utiliza-se o binário nos experimentos. Assim, JSON é o único formato textual avaliado.

O suporte à definição do schema dos dados pode ser um fator favorável a um formato, oferecendo verificações sobre os tipos dos dados recebidos, tornando-os mais confiáveis. Por outro lado, restringe a flexibilidade dos dados transmitidos quanto a definição do schema é obrigatória, já que quem recebe estes dados precisará já possuir o schema em tempo de compilação para poder interpretá-los.

O número de linguagens suportadas não necessariamente reflete a realidade, já que Avro e Protobuf apontam apenas as linguagens suportadas oficialmente, enquanto os outros formatos listam bibliotecas de terceiros em suas páginas oficiais. Considerar bibliotecas terceiras para Protobuf e Avro requer uma pesquisa detalhada à parte para validar cada biblioteca encontrada, que não foi desenvolvida neste trabalho, com excessão da biblioteca Avro para Go utilizada neste trabalho, já que a linguagem não é listada na página oficial e a biblioteca utilizada é mantida por terceiros.

4.4 Estrutura dos experimentos

Para a execução dos experimentos, são implantados dois serviços Web na nuvem, onde um serviço, o *consumidor*, consome dados recebidos de outro serviço, o *produtor*. O consumidor é o agente passivo, aguardando mensagens do produtor para processá-las, o que é feito através de um *endpoint* HTTP onde o consumidor recebe requisições do produtor.

O produtor gera uma mensagem cujo conteúdo é um vetor de tamanho predefinindo de um dos tipos já apresentados (números inteiros, números em ponto flutuante, texto, e estruturas complexas), e.g., 100 números inteiros. Foram utilizados três tamanhos de vetor:

- Pequeno: vetor com 10 elementos.
- Médio: vetor com 10^3 elementos.
- Grande: vetor com 10^5 elementos.

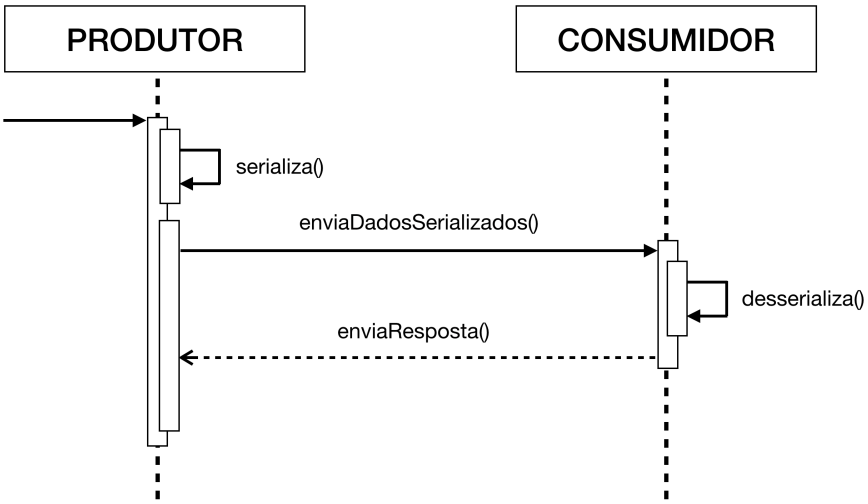
Números maiores de elementos não foram considerados por limitações da infraestrutura, que falha com requisições de 10^6 elementos ou mais em função do tempo de cada requisição.

O produtor realiza uma iteração dos experimentos, serializando a mensagem em um dos formatos apresentados e enviando no corpo de uma requisição HTTP para o consumidor. O consumidor, ao receber a requisição, desserializa a mensagem, e envia a resposta da requisição ao produtor.

Para cada iteração, o produtor agrega um conjunto de medidas: duração da serialização da mensagem, duração do envio da requisição, e duração total da iteração, além de agregar as medidas recebidas do consumidor. O consumidor mede a duração do recebimento do corpo da mensagem (i.e., tempo entre o recebimento da requisição e a leitura de toda a mensagem recebida) e a duração da desserialização. Estas medidas são enviadas de volta ao produtor no corpo da resposta da requisição. Esta iteração é elucidada no diagrama da figura 4.8, onde cada passo tem sua duração medida.

Os serviços foram implantados no *Google App Engine* (GAE), que permite configurar serviços Web na nuvem para serem escalados de forma dinâmica ou estática. Para manter a estabilidade dos experimentos executados neste trabalho, optou-se pela escalação estática (manual), utilizando-se apenas uma instância *B4_HIGHMEM*, que disponibiliza um máximo de 2048MB de memória e 2.4GHz de CPU para cada serviço.

Figura 4.8: Diagrama de uma iteração dos experimentos



Fonte: O Autor

Baseados em duas regiões disponibilizadas pelo GAE, *South Carolina* (`us-east-1`) e *Belgium* (`europe-west`) apresentadas na figura 4.9, construiu-se dois cenários para a execução dos experimentos:

1. Serviços na mesma região: ambos consumidor e produtor foram alocados na região `europe-west`.
2. Serviços em regiões distintas: o consumidor foi alocado para a região `europe-west`, e o produtor para `us-east-1`.

Figura 4.9: Regiões no *Google Cloud Platform*

Fonte: (GOOGLE, 2019c)

Do primeiro cenário, espera-se o melhor resultado em relação ao canal de comunicação entre os dois serviços, já que estão na mesma região e sob o mesmo provedor. No segundo, presume-se uma alta latência em comparação. Deste modo, pode-se comparar o impacto da qualidade do canal de comunicação entre os serviços e qual parâmetro possui maior influência em cada uma das duas situações: tamanho da mensagem ou tempo de serialização e desserialização.

Para cada combinação de parâmetros, são executadas 100 iterações, onde, por fim, todos os parâmetros são:

1. Tamanho do vetor: 3 possibilidades (10, 10^3, 10^5).
2. Tipo do dado do vetor: 4 possibilidades (`int64`, `float64`, `string`, `struct`).
3. Formato de serialização: 4 possibilidades (JSON, Protobuf, Avro, MsgPack).
4. Região de implantação dos serviços: 2 possibilidades (na mesma região, em regiões distintas).

O número de iterações foi mantido em 100, já que, para o maior tamanho do vetor, os experimentos tomaram cerca de 12 horas. Também se tentou iterações com vetores de 1000 elementos, porém estes excederam a cota fornecida pelo GAE, dado o tempo total de execução e o total do tráfego de dados.

5 AVALIAÇÃO DOS RESULTADOS

Este capítulo apresenta os resultados obtidos pelos testes segundo a metodologia apresentada no capítulo 4. Cada fator analisado é apresentado numa seção: o tamanho resultante dos formatos serializados na seção 5.1, o tempo de serialização e desserialização na seção 5.2 e o tempo total das iterações na seção 5.3. Cada seção apresenta uma análise pontual dos resultados. Por fim, a análise geral, considerando todos os resultados, é dada na seção 5.4.

5.1 Tamanho resultante

O tamanho resultante dos dados serializados depende apenas do formato e dos dados em si, logo, não se diferencia entre as regiões onde os testes foram executados. Para os três tamanhos de entrada, são apresentados os resultados nas tabelas 5.1, 5.2 e 5.3.

Tabela 5.1: Tamanho resultante em bytes – dataset pequeno (*B*)

	`int64`	`float64`	`string`	`struct`
JSON	204,95	197,27	1031	3097,3
Protobuf	96,92	82	1020	2290
Avro	99,38	84	1024	2234
MsgPack	91	91	1031	2361

Fonte: O Autor

Tabela 5.2: Tamanho resultante em bytes – dataset médio (*kB*)

	`int64`	`float64`	`string`	`struct`
JSON	19,9029	19,1741	100,5869	302,3820
Protobuf	9,2765	7,8154	99,6093	223,6328
Avro	9,3132	7,8525	99,6591	217,8232
MsgPack	8,7919	8,7919	100,5888	230,4718

Fonte: O Autor

Note que os valores, apesar de representarem bytes, possuem casas decimais por serem a média do tamanho resultante para as 100 iterações, onde cada uma usou um novo conjunto de dados. A relação entre os valores também está expressa no gráfico da figura

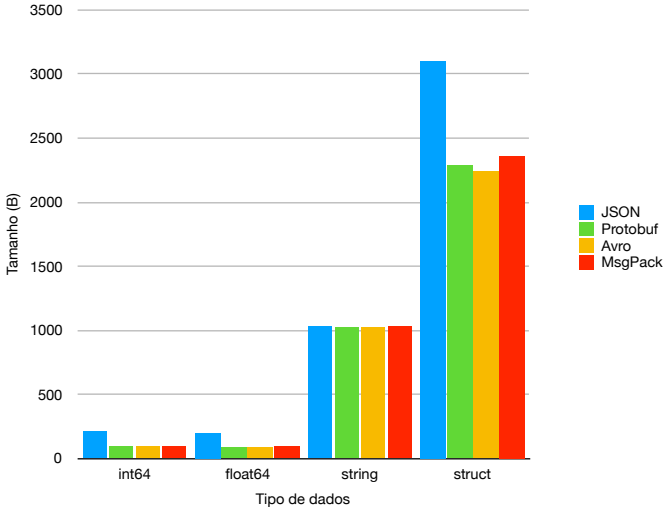
Tabela 5.3: Tamanho resultante em bytes – dataset grande (*MB*)

	`int64`	`float64`	`string`	`struct`
JSON	1,9435	1,872	9,8228	29,5287
Protobuf	0,9056	0,7629	9,7274	21,8391
Avro	0,9094	0,7667	9,7322	21,2717
MsgPack	0,8583	0,8583	9,8228	22,5067

Fonte: O Autor

5.1, construído sobre os valores resultantes do dataset pequeno. Para os outros tamanhos de dataset, os gráficos apresentaram a mesma relação, como visto nas figuras 5.2 e 5.3.

Figura 5.1: Gráfico de tamanhos resultantes com dataset pequeno

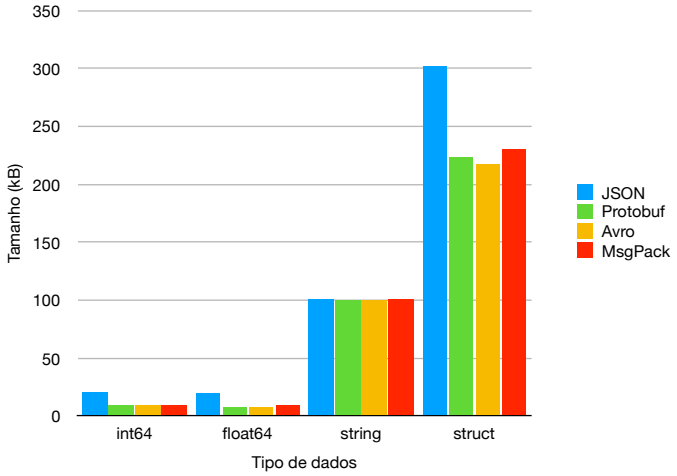


Fonte: O Autor

Destes resultados, nota-se o custo da representação textual do formato JSON em comparação aos outros formatos, que são binários. Com o tipo `string`, a representação é natural para o formato JSON, já que o tipo não passa de texto puro, tal que o resultado se aproxima aos dos outros formatos. Porém, para a representação de números, especialmente para número inteiros de alto valor absoluto ou números com um grande número de casas decimais, a necessidade do uso de um caractere por algarismo se torna cara quando comparada à representação binária destes.

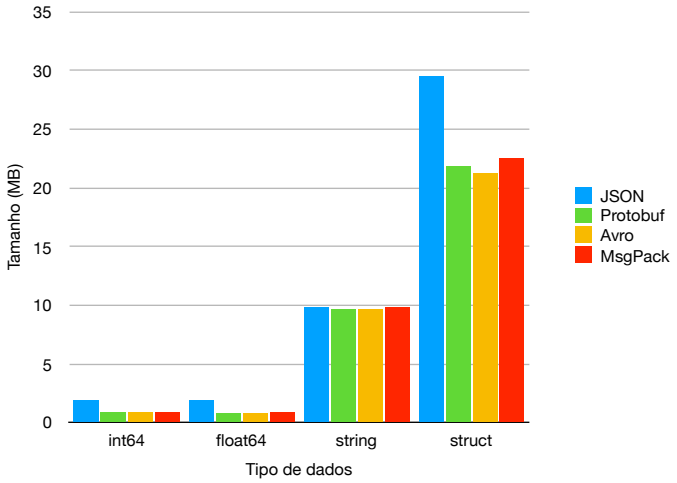
Considerando-se apenas os formatos binários, os valores são muito próximos. MsgPack se destaca na representação de números inteiros, porém não performa tão bem para outros tipos. Isso é consequencia da otimização presente em todos formatos para números inteiros, que utilizam representações de tamanho variável. Já para os outros tipos primitivos, números em ponto flutuante e texto, os valores tem tamanho fixo, e apenas

Figura 5.2: Gráfico de tamanhos resultantes com dataset médio



Fonte: Os Autores

Figura 5.3: Gráfico de tamanhos resultantes com dataset grande



Fonte: Os Autores

MsgPack utiliza um byte adicional para informar o tipo do dado, justificando os resultados. Isto impacta também no resultado de `struct`, já que este engloba os outros tipos apresentados.

Protobuf e Avro apresentam resultados extremamente similares, com a maior diferença ocorrendo com o tipo `struct`, onde Avro apresenta melhores resultados. Enquanto Avro simplesmente concatena o valor de cada propriedade da `struct` em sua ordem, Protobuf não necessariamente mantém estes em ordem quando serializa, usando um byte adicional por campo para o tipo e identificação destes, impactando no tamanho resultante.

5.2 Serialização e desserialização

Durante os testes, os tempos de serialização e desserialização foram obtidos separadamente. Porém, como uma operação não faz sentido sem a outra, e também para fins de uma apresentação mais compacta e clara, os dois valores foram somados para a apresentação de resultados nesta seção.

Novamente, os valores apresentados são uma média das 100 iterações para cada conjunto de parâmetros. Vale lembrar que a serialização ocorre no serviço produtor, enquanto a desserialização no serviço consumidor, e que ambos serviços estavam sujeitos à mesma configuração de hardware na plataforma. Para os três tamanhos de entrada, são apresentados os resultados na tabela 5.4.

Tabela 5.4: Tempo de serialização + desserialização

	`int64`	`float64`	`string`	`struct`
	dataset pequeno – tempo em *ms*			
JSON	0,6725	0,0074	0,0062	0,0197
Protobuf	0,0026	0,0022	0,0031	0,0072
Avro	0,0236	0,0032	0,0048	0,0102
MsgPack	0,0043	0,0038	0,0050	0,0122
	dataset médio – tempo em *ms*			
JSON	5,6291	3,8193	20,5916	14,5347
Protobuf	2,0991	0,0523	2,8566	10,9358
Avro	2,2553	0,5797	7,1484	9,7925
MsgPack	1,5441	0,4830	3,4402	9,5713
	dataset grande – tempo em *ms*			
JSON	84,977	120,0930	260,7970	1351,4940
Protobuf	10,4264	6,8489	120,1438	240,0885
Avro	21,1921	10,6958	137,1582	487,383
MsgPack	11,572	11,61	113,2744	537,5145

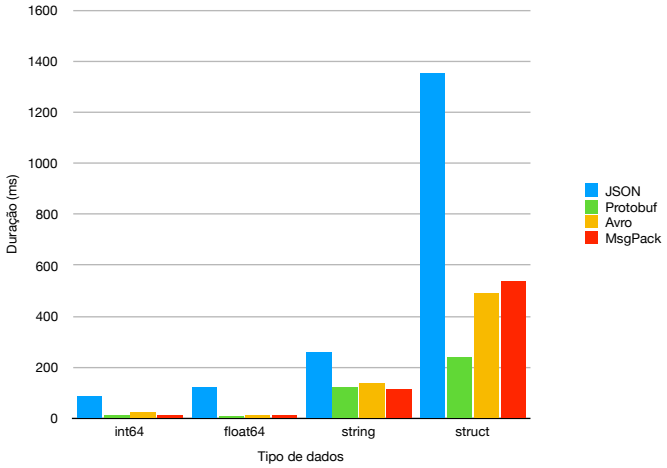
Fonte: O Autor

Por apresentar os resultados mais estáveis, a figura 5.4 utiliza os valores provenientes das iterações com o dataset grande. Os gráficos para os outros datasets podem ser encontrados no Apêndice 9.

Em tempo de processamento, o formato JSON apresentou uma baixa performance quando comparado aos formatos binários. Ao contrário do tamanho resultante, o tipo `string` não foi uma exceção para estes resultados, também sofrendo com sua performance.

Abordando apenas os formatos binários, Protobuf foi o mais eficiente na maioria

Figura 5.4: Gráfico de tempo de serialização + desserialização com dataset grande



Fonte: O Autor

dos casos, mas com um resultado discrepante para o tipo `struct` com dataset médio, já que foi o pior dos três formatos, enquanto para ambos datasets pequeno e grande, este se mostrou o melhor por uma margem considerável. Uma potencial causa para o formato Protobuf se destacar de tal forma é a biblioteca utilizada nos testes, que é mantida pela Google, autora do formato e também da linguagem utilizada. Os outros dois formatos binários utilizam bibliotecas terceiras, que são mantidas por usuários da comunidade e não possuem auxílio corporativo.

O formato Avro, apesar de ter se mantido com tempo similar aos outros formatos binários no geral, apresentou discrepâncias para o tipo `int64` com o dataset pequeno, e para o tipo `string` com o dataset médio. O primeiro pode ser explicado por um *overhead* fixo que é ignorável em maior quantidades de valores, considerando que o formato mantém a definição do schema junto aos dados, porém é necessária maior investigação para tal afirmação. Para o tipo `string`, considerando o resultado apresentado para os datasets pequeno e grande, não se chega a uma conclusão lógica, tornando-o um resultado desconfiável, e requerendo um novo conjunto de iteração dos testes.

5.3 Duração total

Ao se analisar a duração total, é importante se ter em mente os fatores consequentes do canal de comunicação utilizado: a latência e o *throughput* (TANENBAUM et al., 2003). Enquanto a latência representa um valor fixo de tempo a cada iteração, o *throughput* escala o tempo de comunicação proporcionalmente à quantidade de dados

transportados, como expressado na fórmula da figura 5.5. Assim, quanto maior a quantia de dados, maior o impacto do *throughput*, e, por consequência, menos significante é o tempo perdido em função da latência.

Figura 5.5: Fórmula relacionando tempo de comunicação com latência e throughput

$$tempo \propto latencia + \frac{bytes}{throughput} \tag{5.1}$$

Fonte: O Autor

Como o canal afeta os resultados diretamente, esta seção apresenta-os em duas partes: na seção 5.3.1, encontram-se os resultados dos testes executados com ambos consumidor e produtor numa mesma região; na seção 5.3.2, são mostrados os resultados com os serviços alocados em regiões diferentes.

A fim de se obter a latência aproximada nos dois cenários, criou-se um *endpoint* no consumidor que não realiza nenhum processamento, enviando uma resposta diretamente ao receber uma requisição. Com isso, o produtor envia uma requisição HTTP sem um corpo, obtendo o tempo mínimo aproximado de uma requisição. Essa operação foi executada 100 vezes, das quais resultaram as médias:

- Mesma região: 14*ms*.
- Em regiões distintas: 105*ms*.

5.3.1 Serviços numa mesma região

Os resultados obtidos nestes testes são observados na Tabela 5.5. A partir destes valores, geraram-se os gráficos analisados nesta seção.

Ao se analisar as figuras 5.6 e 5.7, nota-se que ambas possuem uma escala similar, tendo resultados com valores relativamente próximos, apesar de a primeira usar um conjunto de apenas 10 valores, enquanto a segunda usa um total de 1000. Embora a escala seja similar, o padrão apresentado nos gráficos é bem diferente, tal que os resultados na primeira não seguem o padrão esperado. Nesta, `float64` apresenta os piores resultados, apesar de o tipo `struct` conter uma propriedade `float64`, além de outras, o que implicaria em `struct` sempre levar mais tempo que `float64`. Como os valores resultantes se aproximam do tempo mínimo esperado, conclui-se que o tempo de serialização

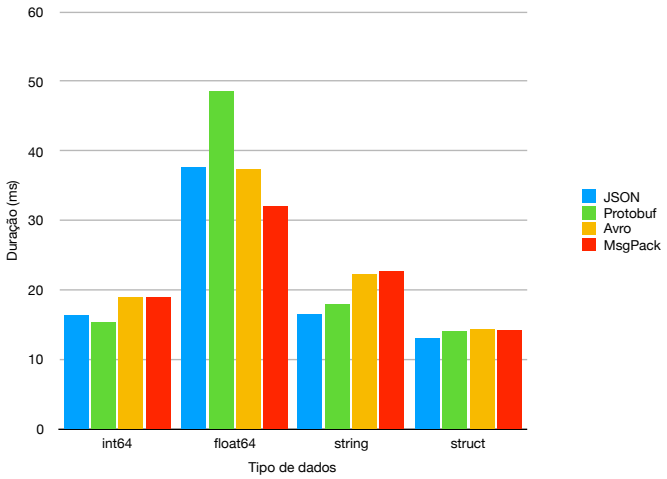
Tabela 5.5: Tempo total para mesma região

	`int64`	`float64`	`string`	`struct`
	dataset pequeno – tempo em *ms*			
JSON	16,3193	37,6685	16,5026	13,1143
Protobuf	15,3626	48,6993	18,0495	14,0634
Avro	19,01443	37,4013	22,3473	14,4178
MsgPack	19,03192	32,1372	22,6122	14,3025
	dataset médio – tempo em *ms*			
JSON	38,8665	29,7598	18,9565	36,7017
Protobuf	30,2198	29,3785	17,8411	22,4226
Avro	26,8808	23,6634	20,195	23,6811
MsgPack	26,5378	29,143	18,5839	24,7451
	dataset grande – tempo em *ms*			
JSON	181,5	174,7	740,7	2761,9
Protobuf	65,8	372,9	549,6	1303,8
Avro	50,6	37,86	591,9	1454,1
MsgPack	44,7	42,6	567,2	1586,9

Fonte: O Autor

e desserialização, assim como o tamanho dos dados transportados, não é relevante frente ao tempo tomado por outros fatores na iteração do teste, como a latência no tempo de comunicação para o dataset pequeno.

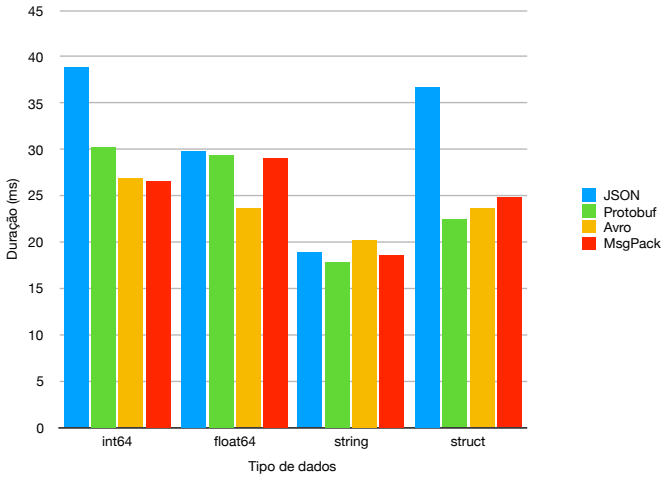
Figura 5.6: Gráfico de tempo total com dataset pequeno para mesma região



Fonte: O Autor

Entre os gráficos das figuras 5.7 e a 5.8, o oposto ocorre. Nestes, a escala é diferente, porém o padrão apresentado por formato e tipo é similar, com JSON se destacando por tomar o maior tempo em todos os casos. Para o tamanho médio, Avro exibe um melhor resultado geral, com destaque para o tipo `float64`. Contudo, para o tamanho

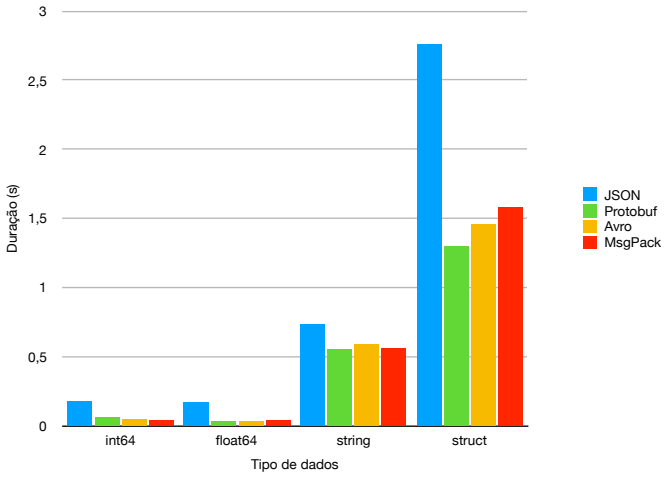
Figura 5.7: Gráfico de tempo total com dataset médio para mesma região



Fonte: O Autor

grande, Protobuf apresenta melhor eficiência.

Figura 5.8: Gráfico de tempo total com dataset grande para mesma região



Fonte: O Autor

5.3.2 Serviços em regiões distintas

Os resultados obtidos nestes testes são observados na Tabela 5.6. A partir destes valores, geraram-se os gráficos analisados nesta seção.

Os valores apresentados para o dataset pequeno se aproximam do tempo mínimo esperado, ratificando a conclusão apresentada no cenário de mesma região: os fatores consequentes do formato de serialização não são relevantes dado o tamanho ínfimo do dataset. O gráfico resultante destes valores pode ser encontrado no Apêncice 9.

Tabela 5.6: Tempo total para regiões diferentes

	`int64`	`float64`	`string`	`struct`
	dataset pequeno – tempo em *ms*			
JSON	106,7336	102,9256	104,2792	108,7998
Protobuf	108,0676	108,3868	103,5477	108,1843
Avro	104,9167	102,6518	117,0377	113,8268
MsgPack	106,2408	102,5908	103,6600	106,5564
	dataset médio – tempo em *ms*			
JSON	193,4662	151,2875	449,5381	1006,8081
Protobuf	133,1255	127,1015	418,7221	746,7087
Avro	131,7404	118,2974	422,7630	729,2308
MsgPack	133,0181	129,2308	414,9813	805,1955
	dataset grande – tempo em *ms*			
JSON	6000,7	5789,7	29800,9	90006,7
Protobuf	2810,1	2429,1	29307,6	66011,0
Avro	2871,0	2418,6	29368,6	64577,0
MsgPack	2685,6	2646,1	29657,6	68322,9

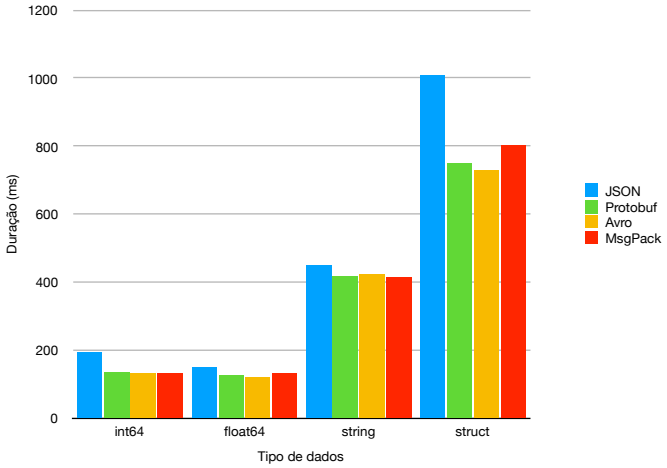
Fonte: O Autor

Com os gráficos das figuras 5.9 e 5.10, constata-se o mesmo padrão entre os formatos e tipos. Porém, com um dataset maior, a variação de tempo entre os tipos com tamanho resultante menor, `int64` e `float64`, e maior, `string` e `struct`, também é maior. Atribui-se essa diferença ao impacto do `throughput` no tempo resultante.

Para melhor explicar a diferença entre os dois gráficos, usa-se o exemplo: para o dataset médio de `int64`, o formato JSON levou $193,5ms$, em média. Destes, esperam-se apenas $105ms$ para o tempo de requisição e resposta, sem considerar o corpo da mensagem. Adiciona-se, ainda, $5,6ms$ do tempo de serialização e desserialização, resultando em, aproximadamente $82,9ms$ para o transporte dos dados na requisição, que representa $42,8\%$ do tempo resultante. Já para o dataset grande, com o mesmo formato e tipo, o tempo total foi de $6.000ms$, e o de serialização e desserialização, $14,5ms$; logo, o transporte toma 98% do tempo, se aplicarmos o mesmo cálculo.

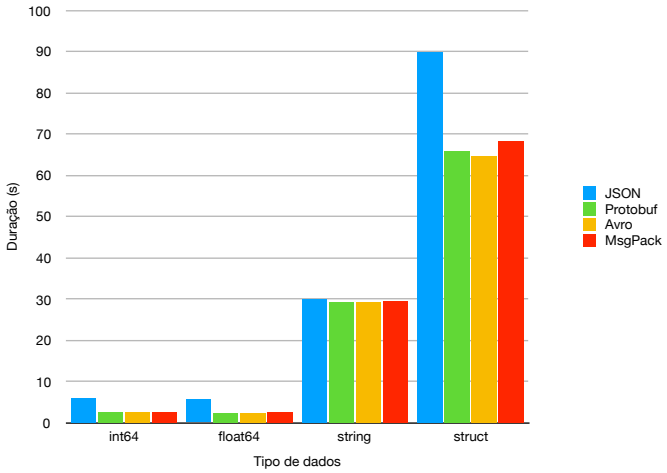
Em outras palavras, no gráfico para o dataset médio, mais de 50% do valor bruto é constante para o caso apresentado, e similarmente para os outros casos, diminuindo a diferença relativa entre eles. O que não ocorre no dataset grande, onde os valores constante representam apenas 2%, destacando a diferença relativa, consequente do tamanho da mensagem e *throughput* do canal.

Figura 5.9: Gráfico de tempo total com dataset médio para regiões diferentes



Fonte: O Autor

Figura 5.10: Gráfico de tempo total com dataset grande para regiões diferentes



Fonte: O Autor

5.4 Análise geral

A partir dos resultados, fica claro o custo de se usar um formato textual, já que o formato JSON foi o menos eficiente em tamanho e em tempo de serialização e desserialização em todos os resultados, com exceção do tamanho para `string`. Porém, para dados suficientemente pequenos, esses valores são ignoráveis frente ao custo de comunicação entre dois serviços, como visto no tempo total para o dataset pequeno.

Quanto aos três formatos binários analisados, Protobuf, Avro e MsgPack, foram apresentados resultados próximos, mas com Protobuf sendo mais eficiente em sua maioria. MsgPack se destacou por sua eficiência com números inteiros, enquanto com dados estruturais se afastou bastante dos resultados apresentados pelo Protobuf.

Para os datasets médio e grande, o tempo total segue o padrão apresentado pela eficiência de cada formato. Como, entre todos os formatos, a relação de eficiência em tamanho foi similar à relação de eficiência em tempo de serialização e desserialização, deve-se analisar cada caso para perceber qual fator foi o principal a afetar o tempo total.

Com os testes de mesma região, dada a boa qualidade do canal, o custo para uma quantia maior de dados é barato quando comparado ao tempo de serialização e desserialização. Por exemplo: para o tipo `struct` em JSON, levou-se, em média, $2,7619s$, enquanto o tempo de serialização e desserialização deste foi de $1,3515s$, constituindo $48,93\%$ do tempo total.

Já com os testes realizados com serviços em regiões distintas, a qualidade do canal e, consequentemente, o tamanho de dados transportados, afetam bruscamente o tempo de execução, tornando o tempo de serialização e desserialização uma parcela pequena do tempo total. E.g.: para uma `struct` em JSON, uma iteração levou $90,0067s$ em média, com o mesmo tempo para conversão de $1,3515s$, que, nesse caso, constitui apenas $1,5\%$ desta duração.

6 CONCLUSÃO

Este trabalho abordou a eficiência de formatos de serialização com respectivas bibliotecas em Go quando se comunicando na nuvem. Ao serem apresentados os conceitos relevantes ao trabalho, notou-se a relevância da interoperabilidade na nuvem, fator que afetou a escolha dos formatos para a análise. Também, sobre serialização, foram destacadas as variáveis que afetam o cenário proposto ao se variar o formato, sendo estas variáveis o tamanho dos dados serializados e o tempo de serialização e desserialização.

Como outros trabalhos já avaliaram diferentes formatos, foram apresentados seus resultados, utilizados também como parâmetro de comparação. Nestes, o custo adicional trazido por um formato textual em comparação a um binário é unânime entre os resultados. Protocol Buffers predominou em sua eficiência em tempo de serialização e desserialização, porém, em relação ao tamanho, os resultados variaram entre Protocol Buffers, Avro e MessagePack.

Com base na metodologia dos trabalhos relacionados e na comunicação sobre HTTP de dois serviços na nuvem, se desenvolveu a metodologia para os experimentos deste trabalho. Nela, buscou-se relacionar as variáveis já presentes em outros trabalhos com os fatores trazidos por este canal de comunicação, ou seja, a relação entre o tamanho dos dados e o tempo das requisições. Desta forma, relacionou-se o tempo de serialização e desserialização com o tempo restante da comunicação. Como a eficiência da comunicação é limitada pela distância física dos pares, explorou-se cenários com diferentes disposições geográficas. Construída a metodologia foram desenvolvidos os experimentos e apresentados os resultados.

Nos resultados dos experimentos deste trabalho, o padrão geral seguiu o que foi apresentado pelos trabalhos relacionados. Porém, como diferencial, pode-se notar a irrelevância da eficiência de um formato na comunicação sobre HTTP quando os dados são suficientemente pequenos. Por outro lado, com o crescimento do tamanho dos dados, notou-se a diferença entre os formatos e também da distância física. Em comunicações numa mesma região, o tempo de serialização e desserialização predominou, já que o tempo de requisição e resposta foram baixos devido à baixa latência e alto *throughput* do canal. Enquanto, em regiões diferentes, o tamanho dos dados foi o principal fator trazido pelo formato, dado o maior impacto do canal.

Também pode-se analisar a eficiência de diferentes tipos de dados em cada formato. Nestes, notou-se a diferença para o tamanho resultante do tipo `struct`, onde Avro

foi o mais eficiente, ao contrário dos outros, onde Protobuf esteve à frente, porém com Avro apresentando resultados muito próximos. Já em relação ao tempo de serialização e desserialização, Protobuf predominou em todos os tipos de dados.

Através dos resultados, obteve-se parâmetros para a escolha de um dos formatos apresentados em cenários reais, considerando a disposição geográfica (região) dos serviços e o tipo e tamanho dos dados comunicados. Também se pode afirmar que a eficiência de um formato é relevante para serviços na nuvem, principalmente em cenário com baixa latência de comunicação e para maiores volumes de dados.

Reafirmou-se que, em geral, formatos textuais são ineficientes quando contrapostos a formatos binários. Ainda assim, a escolha deste também pode depender de parâmetros qualitativos, como, por exemplo, o requisito de um formato legível num cenário onde os dados são usualmente acessados diretamente por humanos, ou a necessidade de lidar com dados dinâmicos e flexíveis, que não possuam um schema os definindo. Por fim, Protobuf se apresentou o melhor candidato, considerando sua eficiência, mas também é o mais restrito quanto a necessidade de definir um *schema*.

6.1 Trabalhos futuros

Como extensão deste trabalho, pode-se buscar os mesmos experimentos utilizando uma segunda linguagem de programação, a fim de evitar possíveis vieses trazidos por Go e as bibliotecas usadas.

Também existem diversos formatos não abordados neste trabalho, que se apresentam semelhantes aos analisados e poderiam ser sujeitados aos mesmos experimentos, como o Thrift, citado em resultados dos trabalhos relacionados.

Apenas uma infraestrutura na nuvem foi avaliada, porém existem provedores concorrentes que fornecem serviços similares e poderiam ser utilizados para a mesma avaliação. A mudança entre estes provedores pode impactar em diversos aspectos de performance, podendo-se ainda considerar uma diferença de preço entre que seriam avaliados.

Enquanto este trabalho propôs uma análise genérica na nuvem, uma análise para cenários específicos, ao exemplo de processamento de mensagens de Smart Grids, deve ser considerada. Para isso, pode-se considerar a mesma metodologia de experimento apresentada, variando apenas os datasets utilizados. A proposta de um cenário específico também pode impor restrições aos tipos analisados, excluindo algum dos tipos testados e propondo outros como alternativa.

REFERÊNCIAS

ARSANJANI, A. Service-oriented modeling and architecture. **IBM developer works**, v. 1, p. 15, 2004.

AVRO, A. **Apache Avro**. 2012. <https://avro.apache.org/docs/1.8.2/> and <https://avro.apache.org/docs/1.8.2/gettingstartedpython.html>. Acessado: 2018-10-06.

BIEBER, G.; CARPENTER, J. Introduction to service-oriented programming (rev 2.1). **OpenWings Whitepaper, April**, 2001.

BRAY, T. STD, **The JavaScript Object Notation (JSON) Data Interchange Format**. [S.l.]: RFC Editor, 2017. <https://buildbot.tools.ietf.org/html/rfc8259>. Acessado: 2018-10-06.

BROWN, P. C. **Implementing soa: total architecture in practice**. [S.l.]: Addison-Wesley Professional, 2008.

COMMITTEE, I. S. C. et al. Ieee standard glossary of software engineering terminology (ieee std 610.12-1990). los alamitos. **CA: IEEE Computer Society**, v. 169, 1990.

COSTELLO, K. **Gartner Forecasts Worldwide Public Cloud Revenue to Grow 17.5 Percent in 2019**. 2019. Disponível em: <https://www.gartner.com/en/newsroom/press-releases/2019-04-02-gartner-forecasts-worldwide-public-cloud-revenue-to-g>. Acessado em: 2019-11-26.

DRAGONI, N. et al. Microservices: yesterday, today, and tomorrow. In: **Present and ulterior software engineering**. [S.l.]: Springer, 2017. p. 195–216.

FURUHASHI, S. **MessagePack**. 2013. <https://msgpack.org/>. Acessado: 2018-10-06.

GO. **The Go Programming Language - Package json**. 2018. <https://golang.org/pkg/encoding/json/>. Acessado: 2018-10-06.

GOOGLE. **App Engine**. 2019. Disponível em: <https://cloud.google.com/appengine/>. Acessado em: 2019-11-29.

GOOGLE. **GCP Free Tier**. 2019. Disponível em: <https://cloud.google.com/free/docs/gcp-free-tier>. Acessado em: 2019-11-29.

GOOGLE. **Global Locations**. 2019. Disponível em: <https://cloud.google.com/about/locations/?tab=regions>. Acessado em: 2020-01-03.

IEEE. STD, **Extensible Markup Language (XML)**. 2019. <https://standards.ieee.org/standard/754-2019.html>. Acessado: 2019-11-29.

LANGUAGE Guide (proto3). 2019. <https://developers.google.com/protocol-buffers/docs/proto3>. Acessado: 2019-11-29.

Maeda, K. Performance evaluation of object serialization libraries in xml, json and binary formats. In: **2012 Second International Conference on Digital Information and Communication Technology and it's Applications (DICTAP)**. [S.l.: s.n.], 2012. p. 177–182. ISSN null.

MELL, P.; GRANCE, T. et al. The nist definition of cloud computing. 2011.

MICROSOFT. **Chapter 1: Service Oriented Architecture (SOA)**. 2016. Disponível em: <https://msdn.microsoft.com/en-us/library/bb833022.aspx>. Acessado em: 2016-02-06.

MICROSOFT. **Serialization (C#)**. 2018. <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/serialization/>. Acessado: 2018-10-06.

MOZILLA. **Serialization - Glossary**. 2017. <https://developer.mozilla.org/en-US/docs/Glossary/Serialization>. Acessado: 2018-10-06.

ORACLE. **The Java™ Tutorials: Serializable Objects**. 2017. <https://docs.oracle.com/javase/tutorial/jndi/objects/serial.html>. Acessado: 2018-10-06.

Petersen, B. et al. Smart grid serialization comparison: Comparision of serialization for distributed control in the context of the internet of things. In: **2017 Computing Conference**. [S.l.: s.n.], 2017. p. 1339–1346. ISSN null.

Protocol Buffers. **Protocol Buffers**. 2018. <https://developers.google.com/protocol-buffers/>. Acessado: 2018-10-06.

SUMARAY, A.; MAKKI, S. K. A comparison of data serialization formats for optimal efficiency on a mobile platform. In: **Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication**. New York, NY, USA: ACM, 2012. (ICUIMC '12), p. 48:1–48:6. ISBN 978-1-4503-1172-4. Disponível em: <http://doi.acm.org/10.1145/2184751.2184810>.

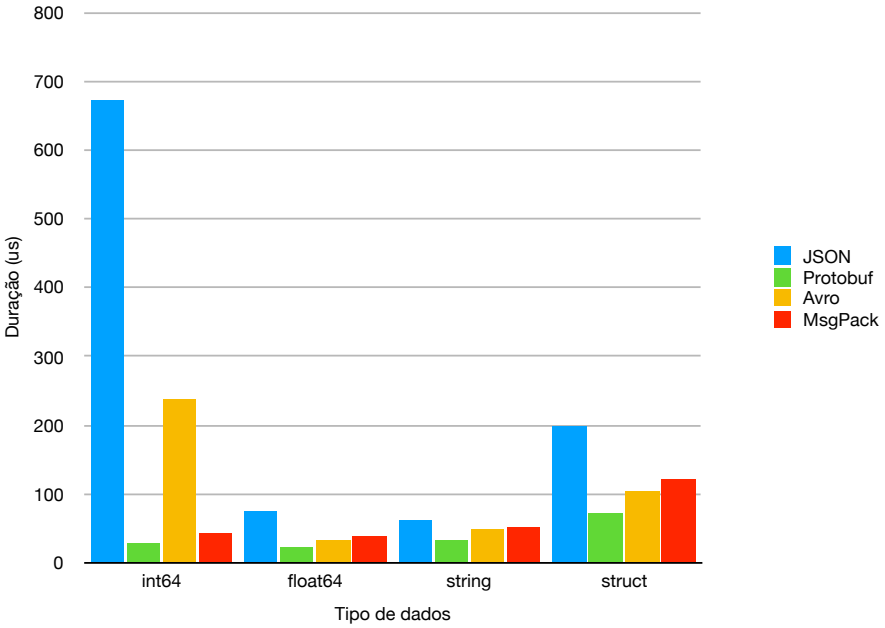
TANENBAUM, A. S. et al. Computer networks, 4-th edition. **ed: Prentice Hall**, 2003.

W3C. STD, **Extensible Markup Language (XML)**. 2016. <https://www.w3.org/XML/>. Acessado: 2019-11-29.

YERGEAU, F. STD, **Extensible Markup Language (XML)**. 2003. <https://tools.ietf.org/html/rfc3629>. Acessado: 2019-11-29.

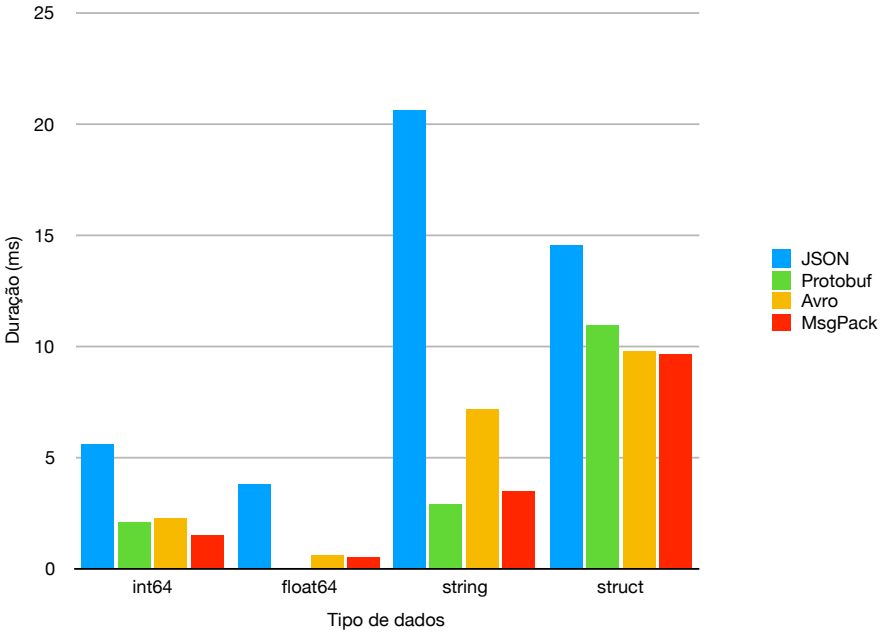
APÊNDICE GRÁFICOS COMPLEMENTARES

Figura 1: Gráfico de tempo de serialização + desserialização com dataset pequeno



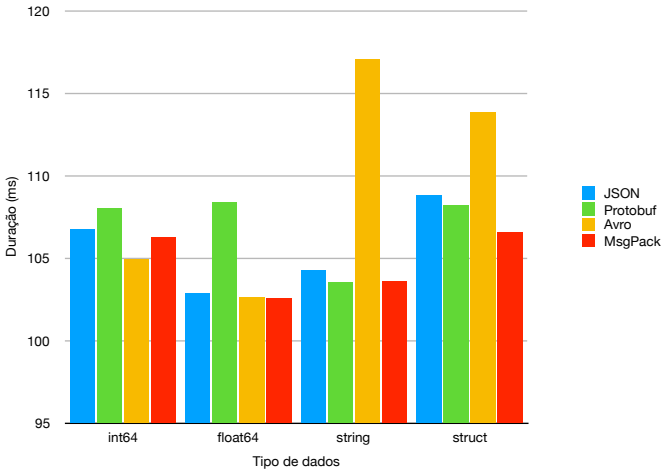
Fonte: O Autor

Figura 2: Gráfico de tempo de serialização + desserialização com dataset médio



Fonte: O Autor

Figura 3: Gráfico de tempo total com dataset pequeno para regiões diferentes



Fonte: O Autor