UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
GRADUAÇÃO EM SISTEMAS DE INFORMAÇÃO

Matheus Miranda Torres Gomes

Avaliação de estratégias de serialização para comunicação em sistemas distribuídos

Florianópolis
2023

Matheus Miranda Torres Gomes

Avaliação de estratégias de serialização para comunicação em sistemas distribuídos

Trabalho de Conclusão do Curso de Graduação em Sistemas de Informação do Centro Tecnológico da Universidade Federal de Santa Catarina para a obtenção do título de Bacharel em sistemas de informação.
Orientador: Prof. Odorico Machado Mendizabal, Dr.

Florianópolis
2023

Ficha de identificação da obra

A ficha de identificação é elaborada pelo próprio autor.

Orientações em:

http://portalbu.ufsc.br/ficha

Matheus Miranda Torres Gomes

Avaliação de estratégias de serialização para comunicação em sistemas distribuídos

O presente trabalho em nível de foi avaliado e aprovado por banca examinadora composta pelos seguintes membros:

Prof.(a) Odorico Machado Mendizabal, Dr(a).
Instituição Universidade Federal de Santa Catarina

Prof.(a) Patricia Della Mea Plentz, Dr(a).
Instituição Universidade Federal de Santa Catarina

Prof.(a) Wyllian Bezerra da Silva, Dr(a).
Instituição Universidade Federal de Santa Catarina

Certificamos que esta é a **versão original e final** do trabalho de conclusão que foi julgado adequado para obtenção do título de Bacharel em sistemas de informação.

_____

Coordenação do Programa de Graduação
em Sistemas de Informação

_____

Prof. Odorico Machado Mendizabal, Dr.
Orientador

Florianópolis, 2023.

AGRADECIMENTOS

Agradeço ao meu orientador Prof. Dr. Odorico Machado Mendizabal, pelas orientações, ensinamentos, mentoria e revisões, além dos membros da banca Profª. Drª. Patricia Della Mea Plentz e Prof. Dr. Wyllian Bezerra da Silva com as contribuições à monografia. Agradeço à minha família, namorada e amigos por todo o apoio durante minha graduação, e agradeço de forma geral à Universidade Federal de Santa Catarina pela oportunidade de poder estudar e contribuir no âmbito acadêmico.

RESUMO

Sistemas distribuídos são essenciais para o processamento e distribuição de informações em uma sociedade tecnologicamente avançada. Projetar um sistema distribuído requer considerações complexas, especialmente em relação à eficiência da comunicação. Escolher o formato de comunicação correto é crucial, pois afeta o desempenho, custos e disponibilidade do serviço. Este trabalho se propõe a realizar uma análise exploratória das bibliotecas de serialização Newtonsoft.Json, MessagePack, Apache Avro, Apache Thrift, Protocol Buffers, FlatBuffers, e Cap'n Proto, para transporte de dados em sistemas distribuídos no escopo de *microbenchmark* e comunicação entre serviços no modelo cliente-servidor, construindo um protótipo na linguagem C# que represente um sistema de tráfego intensivo de dados. Com testes de desempenho, é explorado o comportamento das mesmas com grandes volume de serialização, de forma sequencial e concorrente, para determinar a eficiência em diversos cenários, além de uma análise quanto aspectos de implementação e de forma qualitativa sobre as funcionalidades de cada uma. As métricas coletadas envolveram, tempo de processamento, taxa de compressão e vazão do sistema, assim como métricas qualitativas, como abrangência em linguagens e interoperabilidade. Dentre as bibliotecas avaliadas, FlatBuffers, MessagePack e Protocol Buffers desempenharam de forma eficiente em todos os cenários, enquanto Apache Avro e Thrift obtiveram tempos mais lentos. FlatBuffers apresentou tempos de desserialização e um consumo muito baixo de memória RAM, mesmo trabalhando com grandes volumes de dados. Newtonsoft.Json, apesar de ser uma serialização textual, apresentou um importante ganho de desempenho fazendo uso da concorrência, assim como outras bibliotecas. No âmbito distribuído, Newtonsoft.Json, com carga menor de clientes simultâneos, apresentou bom desempenho, Apache Avro teve maior vazão na serialização, enquanto Protocol Buffers teve menor vazão média na serialização observada.

Palavras-chave: Sistemas distribuídos, Serialização, *Client-server*, *Benchmark*, C#.

ABSTRACT

Distributed systems are essential for processing and distributing information in a technologically advanced society. Designing a distributed system requires complex considerations, especially in relation to communication efficiency. Choosing the right communication format is crucial, as it affects performance, costs and service availability. This work aims to carry out an exploratory analysis of the serialization libraries Newtonsoft.Json, MessagePack, Apache Avro, Apache Thrift, Protocol Buffers, FlatBuffers, and Cap'n Proto, for data transport in distributed systems in the scope of *microbenchmark* and communication between services in the client-server model, building a prototype in the C# language that represents a data-intensive traffic system. Performance tests explore their behaviour with large volumes of serialization, sequentially and concurrently, to determine their efficiency in different scenarios, as well as an analysis of the implementation aspects and qualitative analysis of the functionalities of each one. The metrics collected included processing time, compression rate and system throughput, as well as qualitative metrics such as language coverage and interoperability. Among the libraries evaluated, FlatBuffers, MessagePack and Protocol Buffers performed efficiently in all scenarios, while Apache Avro and Thrift had slower times. FlatBuffers showed deserialization times and very low RAM consumption, even when working with large volumes of data. Newtonsoft.Json, despite being a textual serialization, showed a significant performance gain by using concurrency, as did other libraries. In the distributed environment, Newtonsoft.Json, with a lower load of concurrent clients, performed well, Apache Avro had the highest serialization throughput, while Protocol Buffers had the lowest average serialization throughput observed.

Keywords: Distributed systems, Serialization, Client-server, Benchmark, C#.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

4G	*4th-Generation Wireless*
5G	*5th-Generation Wireless*
AMPQ	*Advanced Message Queuing Protocol*
API	*Application Programming Interface*
ASCII	*American Standard Code for Information Interchange*
BE	*Big-Endian*
BSD	*Berkeley Software Distribution*
BSON	*Binary JSON*
CCSID	*Coded Character Set Identifier*
CoAP	*Constrained Application Protocol*
CPU	*Central Processing Unit*
DDD	*Domain-Driven Design*
DIP	*Dependency Inversion Principle*
DOS	*Distance Oracle System*
EBCDIC	*Extended Binary Coded Decimal Interchange Code*
GB	*gigabyte*
HDF	*Hierarchical Data Format*
HDT	*Header, Dictionary, Triples*
HPC	*High Processing Computing*
HTTP	*Hypertext Transfer Protocol*
IoT	*Internet of Things*
IoV	*Internet of Vehicles*
JSON	*Javascript Object Notation*
k	*milhar*
LE	*Little-Endian*
LEB128	*Little Endian Base 128*
LTE	*Long Term Evolution*
MPI	*Message Passing Interface*
MQTT	*Message Queuing Telemetry Transport*
QoS	*Quality of Service*
RAM	*Random Access Memory*
RDF	*Resource Description Framework*
REST	*Representational State Transfer*
RFC	*Request for Comments*
RPC	*Remote Procedure Call*
RTT	*Round trip Time*
SOA	*Service-Oriented Architecture*

SRP	*Single-Responsability Principle*
TB	*terabyte*
TCP	*Transmission Control Protocol*
URL	*Uniform Resource Locator*
UTF-8	*8-bit Unicode Transformation Format*
UUID	*Universally Unique Identifier*
VU	*Virtual User*
Wi-Fi	*Wireless Fidelity*
XML	*eXtensible Markup Language*

SUMÁRIO

1 INTRODUÇÃO

O rápido avanço tecnológico devido à globalização nos dias atuais exige aos sistemas atuais cada vez mais velocidade de processamento, entrega e recebimento de dados, de forma a atender às demandas da sociedade como um todo. Para conseguir dar vazão e atender às demandas de processamento, os sistemas distribuídos empenham papel chave ao proporcionar a distribuição de dados e processamento para conseguir trabalhar com grandes volumes de dados, a exemplo das redes sociais.

Ao se considerar aplicações intensivas em dados, é possível trabalhar com distintos paradigmas arquiteturais distribuídos. Entre eles há o paradigma de microsserviços (WEERASINGHE; PERERA, 2021), que pode ser ilustrado como a distribuição da aplicação em vários serviços executando independentemente e comunicando entre si, para satisfazer os requisitos do negócio. Também é possível elencar o paradigma *Service-Oriented Architecture* (SOA), definido por meio de provedores que publicam interfaces de seus serviços para os serviços consumidores, utilizando comunicação via *middleware*, assim como outros paradigmas.

Serialização é considerado o processo que traduz uma representação de distintas estruturas de dados em memória para formatos independentes de linguagem e arquitetura, que são enviados por meio da Internet ou armazenados em disco (WOLNIKOWSKI *et al.*, 2021). O objeto final da serialização é um fluxo de bytes (SAYAR *et al.*, 2022). A serialização pode se dar de forma textual, tendo como exemplos *JavaScript Object Notation* (JSON) e *eXtensible Markup Language* (XML) (BRAY, Tim *et al.*, 2000), como também formatos binários, a exemplo dos formatos de serialização MessagePack (FURUHASHI, 2008) e Protocol Buffers (GOOGLE, 2022b). Já a desserialização consiste na leitura da fluxo de bytes serializada de forma a reconstruir a estrutura original da informação (SAYAR *et al.*, 2022).

O processo de serialização e desserialização de dados, em diversos cenários, pode ser considerado um gargalo dentro do sistema. Na desserialização, a aplicação tipicamente está acessando dados codificados, recebidos por meio da Internet e que precisam ser decodificados. Além de precisar atravessar a hierarquia de acesso à memória, a *Central Processing Unit* (CPU) executa computações para traduzir a informação novamente, fazendo o inverso do processo de serialização (WOLNIKOWSKI *et al.*, 2021). Percebe-se, portanto, que a computação eficiente e rápida nos processos de serialização e desserialização impactam diretamente na vazão de um sistema.

Uma das características de um sistema distribuído é a interoperabilidade. Interoperabilidade é a capacidade de sistemas conseguirem operar mutualmente entre si e fazer uso de dados produzidos por diversos sistemas, sem esforço pelo usuário final dos sistemas (COSTIN, A.; EASTMAN, 2019). Sendo assim, um formato de dados agnóstico é essencial para alcançar esta característica. No entanto, considerando

sistemas distribuídos, também é preciso tratar sobre heterogeneidade. Um sistema distribuído consiste em uma coleção de computadores distintos e fracamente acoplados, chamados sites, os quais estão distribuídos geograficamente e conectados por uma rede de comunicação (XIE; QIN, 2007). Dentro do escopo desta definição, considera-se que sistemas distribuídos podem ser compostos por diferentes linguagens, a partir das decisões arquiteturais de cada serviço. Para alcançar heterogeneidade e interoperabilidade, faz-se importante a abrangência em linguagens por cada biblioteca de serialização, assim como outros aspectos referentes à sua implementação.

Utilizando os conceitos de bibliotecas de serialização e comunicação distribuída, foram realizados testes de desempenho em *microbenchmark* e em protótipo que implementa comunicação em modelo cliente-servidor, além de analisadas os características de implementação de cada serializador no protótipo, com os resultados compilados em cada cenário, para se obter avaliações qualitativa e quantitativa das bibliotecas de serialização, com o uso da linguagem C#.

Como principais contribuições deste trabalho, foi possível agregar diversas informações sobre bibliotecas de serialização binárias disponíveis para a linguagem C#, elencando suas funcionalidades e características disponíveis. Quanto ao âmbito quantitativo, realizou-se a agregação dos resultados das bibliotecas analisadas em comparação à biblioteca de serialização textual Newtonsoft.Json, tanto com a serialização individual quanto múltipla. Além disso, foram extraídas métricas de desempenho de cada biblioteca com o uso de concorrência em múltiplas *threads*, buscando apresentar o ganho proporcional de desempenho ao usar uma abordagem que busque melhor utilização das *threads* disponíveis. Ao mesmo tempo, foram apresentadas características técnicas e empíricas que visam enriquecer a análise e escolha de um formato de serialização de acordo com a necessidade observada, levando em consideração as funcionalidades disponíveis, assim como possíveis dificuldades de manutenção e implementação. Dado um ambiente com comunicação distribuída, foi possível observar a vazão e latência média na serialização e desserialização de um sistema com recursos computacionais das características apresentadas, implementado em C#.

1.1 OBJETIVOS

Este trabalho tem como objetivo principal prover uma análise qualitativa e quantitativa que vise identificar e determinar estratégias de otimização na comunicação em sistemas distribuídos.

1.1.1 Objetivos Específicos

a) Elencar, por meio de estudo comparativo, as bibliotecas de serialização com maior destaque para os cenários avaliados;

b) Gerar uma avaliação comparativa de desempenho das bibliotecas selecionadas por meio de testes em ambiente de *microbenchmark* e em protótipo local com comunicação distribuída no modelo cliente-servidor;

c) Analisar aspectos qualitativos sobre cada experiência de implementação;

d) Identificar e destacar as estratégias e bibliotecas de serialização mais eficientes e adequadas para o desenvolvimento de sistemas distribuídos na arquitetura cliente-servidor, com base nas conclusões e métricas obtidas durante a análise comparativa.

1.2 ORGANIZAÇÃO DO TRABALHO

O restante do trabalho está organizado como segue. No Capítulo 2, serão revisados os conceitos e fundamentos teóricos de serialização, tipos de formato, interoperabilidade e compatibilidade de esquemas de serialização. Seguindo, o Capítulo 3 revisa trabalhos acadêmicos e pesquisas relevantes do estado da arte. No Capítulo 4, as bibliotecas de serialização com maior popularidade no cenário de arquiteturas distribuídas são elencadas e exploradas. Já o Capítulo 5 descreve a configuração de ambiente, sistema e métricas a serem avaliadas, além da apresentação e análise dos resultados obtidos por meio dos testes de de desempenho. O Capítulo 6 sumariza os principais resultados e contribuições do trabalho, contendo *insights* obtidos a partir da avaliação e implementação das bibliotecas. Por fim, serão apresentadas as limitações encontradas durante o estudo, e trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo são apresentados os principais conceitos, teorias e estudos relacionados ao tema de serialização, que alicerçam o trabalho em questão.

2.1 SERIALIZAÇÃO E DESSERIALIZAÇÃO

A serialização e desserialização são processos essenciais para converter dados em um formato legível por máquinas, tornando-os transferíveis e utilizáveis por diferentes softwares. Essas técnicas permitem codificar ou traduzir dados em formatos de arquivo neutros, que podem ser recuperados ou utilizados por outras aplicações (ADIBFAR; COSTIN, Aaron M., 2021). De acordo com a definição de Wolnikowski *et al.* (2021), a serialização é o processo de traduzir a representação em memória de estruturas de dados em formatos independentes de arquitetura e linguagem, permitindo o envio dessas estruturas por meio de redes ou o armazenamento em disco. Viotti *et al.* (2022) descreve a serialização como o processo de tradução de uma estrutura de dados em uma sequência de bits para fins de armazenamento ou transmissão, com a possibilidade de reconstruir a estrutura original por meio do processo de desserialização.

As bibliotecas de serialização fornecem a tradução bidirecional entre estruturas de dados e sequências de bits, sendo categorizadas de acordo com a forma como a informação é representada nesta sequência. Há uma grande variedade de formatos de serialização, diferenciando-se em termos de serem ou não orientados a esquema, textual ou binária, e no suporte a coleções e outros tipos de dados.

Os formatos *Extensible Markup Language* (XML) (BRAY, Tim *et al.*, 2000) e *Javascript Object Notation* (JSON) (BRAY, T., 2014) são exemplos de formatos de serialização textual amplamente utilizados para a transferência de informações na Web. JSON, em particular, tornou-se uma alternativa popular ao XML devido à sua legibilidade e facilidade de uso. Ambos os formatos desempenham um papel significativo na representação de dados atualmente. No entanto, em detrimento da legibilidade, eles podem sacrificar a eficiência espacial e a rapidez na serialização.

2.1.1 Interoperabilidade

A interoperabilidade refere-se à capacidade de interconexão, compartilhamento e utilização contínua e eficiente de dados entre softwares (ADIBFAR; COSTIN, Aaron M, 2019). Weichhart *et al.* (2021) destaca a importância da interoperabilidade no design de sistemas e enfatiza sua necessidade em sistemas compostos por múltiplos sistemas. O autor define a interoperabilidade por meio de três escopos: organizacional, semântico e técnico. Pelo viés de um escopo organizacional, (FIRESTONE; MCELROY,

2005) encara a interoperabilidade como um processo contínuo a ser seguido nos sistemas de uma organização, para que a mesma seja mantida mesmo com a evolução dos sistemas de uma organização. No âmbito técnico, a interoperabilidade entre sistemas pode ser alcançada por meio do uso de padrões de protocolos que suportam a transferência de dados entre módulos de software desenvolvidos de forma independente. Exemplos desses padrões são os protocolos baseados em troca de mensagens, como o *Message Queuing Telemetry Transport* (MQTT) e os *middlewares* que permitem o desacoplamento entre dados e serviços computacionais (WEICHHART; PANETTO; MOLINA, 2021). Já a interoperabilidade semântica diz respeito ao fluxo de dados e à compreensão dos dados transmitidos entre remetente e destinatário (WEICHHART; PANETTO; MOLINA, 2021).

2.1.2 Serialização orientada e não orientada a esquema

A serialização de dados pode ser classificada como: (i) orientada a esquema; (ii) não orientada a esquema. Essa distinção se baseia no uso ou não de uma definição formal de estrutura de dados, chamada de esquema. Assim, no contexto de serialização, as bibliotecas podem ou não utilizar uma definição de esquema para facilitar a codificação e decodificação, a depender do tipo de resultado que se espera obter. Se a serialização utiliza esquemas, ela é considerada orientada a esquema, caso contrário será uma biblioteca não orientada a esquema.

No entanto, nem sempre uma definição de esquema permite que a cadeia de bits seja auto-descritiva. O nível de detalhamento dentro de um esquema em relação a uma estrutura de dados pode ser considerada inversamente proporcional à quantidade de informações que precisam ser codificadas na sequência de bits para representar esses dados. Quando um esquema não consegue capturar todas as informações essenciais sobre os dados, é necessário embutir essas informações na sequência de bits resultante, o que aumenta o tamanho final da serialização (VIOTTI; KINDERKHEDIA, 2022b). Também é possível estender esta relação, em maior nível de abstração, ao considerar o custo de processamento para codificar e decodificar a informação contida na cadeia de bits de modo que, quanto maior o nível de detalhamento de um esquema, menor será o custo para interpretar a informação contida na sequência binária.

Serialização não orientada a esquema: um formato que define determinada serialização não orientada a esquema permite a codificação e decodificação dos dados sem conhecimento prévio da estrutura de dados. Essa abordagem é considerada autodescritiva, pois a própria sequência de bits contém informações suficientes para interpretar os dados. Exemplos de formatos de serialização não orientados a esquema são JSON (BRAY, T., 2014) e o MessagePack (FURUHASHI, 2008).

Do ponto de vista de implementação e manutenção, bibliotecas de serialização não orientadas a esquema são consideradas vantajosas, pois conseguem descrever

as informações transmitidas sem a necessidade da definição de uma estrutura formal de dados, tanto no emissor como no receptor. Além disso, evitam a necessidade de atualizar as definições em cada sistema ou serviço adjacente quando há uma mudança na estrutura de dados.

No entanto, a falta de um esquema torna o processo de integração, validação, recuperação e análise dos dados mais complexa e lenta (FROZZA; SANTOS MELLO; COSTA, 2018). Além disso, a ausência de uma definição de estrutura requer a descrição dessa estrutura dentro da cadeia binária resultante, o que não é eficiente em termos de espaço e transmissão.

Apesar de ser chamada serialização não orientada a esquema, a falta de uma definição de estrutura explícita não significa a ausência total de um esquema. De acordo com (VIOTTI; KINDERKHEDIA, 2022b) e (FROZZA; SANTOS MELLO; COSTA, 2018), a serialização não orientada a esquema pode ser categorizada como um subconjunto da serialização orientada a esquema, cuja definição de esquema é tão genérica que pode ser considerada vazia, não necessitando carregar informações relevantes sobre a estrutura de dados transmitida. Assim, a serialização não orientada a esquema contém um esquema genérico o suficiente para conseguir trafegar qualquer sequência de bits.

Serialização orientada a esquema: formatos de serialização orientados a esquema buscam evitar inserir informações referentes à estrutura de dados na sequência de bits resultante. Desta forma, é possível obter eficiência de espaço, reduzindo o tamanho da serialização. Para aplicações que exigem eficiência na transmissão de dados pela rede, tende-se a utilizar formatos de serialização orientados a esquema, como compara (POPIĆ *et al.*, 2016).

A escrita e manutenção das definições de esquema são peças chave de especificações de serialização orientadas a esquema. Em sua maioria, as especificações utilizam sua própria linguagem de definição de esquema, que pode descrever a estrutura de dados em alto ou baixo nível. A forma como o esquema é definido pode variar de acordo com cada especificação, podendo ser fornecido em um arquivo de definição de esquema, como é feito em Protocol Buffers (GOOGLE, 2022b), ou ser inferido implicitamente a partir dos dados.

2.1.3 Compatibilidade de esquema

Sistemas de alta qualidade estão sujeitos a mudanças constantes, seja por novos requisitos, avanços tecnológicos ou conhecimento adquirido durante o desenvolvimento e manutenção do software (RAJLICH, 2014). Juntamente com o código, evoluções nos modelos de dados também são necessárias durante o processo de desenvolvimento e manutenção de software, o que traz desafios na gestão de compatibilidade. A necessidade de evolução de esquema pode surgir de diversas fontes, e

para sistemas que utilizam formatos de serialização orientados a esquema, a compatibilidade entre versões deve ser atentamente observada.

Roddick (1995) define três termos relacionados à modificação, evolução e versionamento de esquemas:

- *Modificação de esquema* - atingida quando um sistema permite modificações na definição de esquema;
- *Evolução de esquema* - atingida quando um sistema facilita a modificação do esquema sem perda de dados existente;
- *Versionamento de esquema* - atingida quando um sistema permite o acesso à informação original, retrospectivamente e prospectivamente, por meio de interfaces de versão definidas pelo usuário.

Com base nessas definições, desserializar (VIOTTI; KINDERKHEDIA, 2022b) descreve que os esquemas são compatíveis quando um dos um deles consegue desserializar uma cadeia de bits produzida por outro e recuperar a informação original. Também classifica a compatibilidade em três níveis, comparando dois esquemas:

- *Para trás:* o primeiro esquema é compatível com versões anteriores em relação ao segundo esquema se o primeiro esquema consegue desserializar os dados produzidos pelo segundo;
- *Para frente:* o primeiro esquema é compatível com versões posteriores em relação ao segundo esquema se o segundo esquema consegue desserializar os dados produzidos pelo primeiro;
- *Completo:* um esquema é completamente compatível com o esquema anterior se o mesmo é tanto compatível com versões anteriores, como com versões posteriores.

Portanto, para garantir a evolução adequada dos esquemas de dados em sistemas com serialização orientada a esquema, é importante considerar a compatibilidade em diferentes direções e níveis, a fim de evitar problemas de desserialização e perda de dados.

2.1.4 Serialização binária e baseada em texto

Serialização binária e serialização baseada em texto são abordagens distintas para representar e armazenar dados durante o processo de serialização. Uma serialização é considerada textual se a cadeia de bits resultantes da codificação corresponde a caracteres em codificação textual como *American Standard Code for Information Interchange* (ASCII), *Extended Binary Coded Decimal Interchange Code* (EBCDIC), *Coded character set identifier* (CCSID) ou *8-bit Unicode Transformation Format* (UTF-8). Caso não, a especificação da serialização é considerada binária (VIOTTI; KINDERKHEDIA, 2022b).

A serialização textual tem a vantagem de ser legível e amigável ao ser humano, tanto por respeitar os limites das codificações textuais, como também pela ampla gama de aplicações e ferramentas que facilitam a leitura e decodificação da informação serializada baseada em texto. Por outro lado, a serialização binária não precisa respeitar os limites da codificação textual, o que permite que sejam utilizadas distintas e flexíveis estruturas de codificação mais eficientes. No entanto, isto faz com que a quantidade de ferramentas disponíveis para leitura e representação sejam reduzidas, sendo necessário fazer o uso de ferramentas especializadas para decodificação da informação serializada e formato binário, além do acompanhamento e estudo da documentação (VIOTTI; KINDERKHEDIA, 2022b).

Serialização baseada em texto: na serialização baseada em texto, a cadeia final de bits codificada corresponde a caracteres em codificação textual. Dois exemplos populares são XML e JSON. Esses formatos de serialização são amplamente utilizados devido à disponibilidade de ferramentas e suporte em várias linguagens de programação. O trabalho (MOONEY; MINGHINI, 2022) mostra que, apesar de que os formatos de serialização binários apresentam melhor performance, os formatos textuais como XML e JSON apresentam vantagem devido à quantidade 'quase universal' de ferramentas e suporte às bibliotecas nas principais linguagens de programação, provendo rápido acesso a arquivos nos formatos textuais. Esta é uma vantagem em relação à serialização binária, visto que, devido à necessidade de apresentar um contrato de definição de esquema, a exemplo de Apache Avro e Protocol Buffers, não é possível iniciar a comunicação imediatamente, fazendo com que os provedores tenham que publicar as definições junto de cada *Application Programming Interface* (API).

Serialização binária: é o processo de transformação de estruturas de dados de um objeto em uma cadeia binária que pode ser transmitida e/ou armazenada, para ser reconstruída posteriormente (MOONEY; MINGHINI, 2022). Esta cadeia binária, por sua vez, não precisa respeitar os limites impostos por codificações textuais, o que garante flexibilidade para a criação de um mecanismo de serialização. Por conta disso, especificações de serialização binária também costumam permitir a troca de grandes volumes de dados de forma interoperável (VANURA; KRIZ, 2018).

Embora os formatos binários sejam mais eficientes em termos de custos computacionais, tempo de processamento e tamanho da cadeia de bits resultante, há *overheads* a serem considerados durante a implementação. Alguns exemplos são software especializado, configurações adicionais e definições de esquema (VIOTTI; KINDERKHEDIA, 2022a), a necessidade de atualização manual ou semi-automatizada das informações de esquema, assim como a necessidade de se ter conhecimento especializado (MOONEY; MINGHINI, 2022).

Diversos estudos têm comparado formatos de serialização binária e textual, considerando aspectos como interoperabilidade, vantagens e desvantagens de acordo

com o propósito e facilidade de uso. Por exemplo, uma análise comparativa de desempenho conduzida por Vanura *et al.* (2018) demonstra que formatos de serialização como Apache Avro e Protocol Buffers têm melhor desempenho em relação a bibliotecas XML e JSON, apesar de demandarem uma definição de esquema. Ainda, mostra que bibliotecas XML apresentam resultados inferiores em relação a JSON. No entanto, os resultados variam em relação ao tempo de serialização e desserialização.

Outro estudo mencionado por Mooney *et al.* (2022) destacou problemas relacionados à gestão de espaço em definições de ontologia (MARTÍNEZ-PRIETO; ARIAS GALLEGO; FERNÁNDEZ, 2012; GIMENEZ; FERNANDEZ; MARTINEZ, 2017), majoritariamente usando *Resource Description Framework* (RDF), propondo o uso de formato de serialização binário, como *Hierarchical Data Format* (HDF) e RDF binário chamado *Header, Dictionary, Triples* (HDT) (KRIJNEN; BEETZ, 2017). O trabalho (MOONEY; MINGHINI, 2022) também aponta que formatos de serialização binários orientados a esquema podem ser melhor aproveitados onde não há mudança constante em seus conjuntos de dados.

Apesar das dificuldades envolvidas, tornar as abordagens de serialização binária mais amigáveis ao usuário pode ter grandes impactos positivos em sua adoção futura (MOONEY; MINGHINI, 2022).

3 TRABALHOS RELACIONADOS

Este capítulo apresenta trabalhos correlatos ao tema abordado neste trabalho de conclusão de curso. Especialmente, foram selecionadas pesquisas que tratam de estratégias de otimização de comunicação envolvendo diferentes estratégias de serialização e uso de protocolos.

Foi realizada a verificação do estado da arte, seguida de uma revisão extensiva dos principais conceitos ao redor da literatura aos quais esse trabalho alinha-se dentre eles: serialização e desserialização de dados, análise comparativa de desempenho, comunicação cliente-servidor e produtor-consumidor. A revisão bibliográfica do estado da arte foi realizada através do Google Scholar[1], onde foi feita a seleção de artigos recentes e de relevância no meio acadêmico-científico, sempre quando possível. O escopo da pesquisa envolveu os conceitos e trabalhos ao redor dos temas descritos, justamente com a extração e estudo dos conceitos principais que embasam este trabalho. Desta forma, foi possível obter uma visão clara e transparente do estado da arte, permitindo a realização do trabalho. Além disso, foi realizada a coleta de três trabalhos relacionados ao tema deste trabalho, onde foram expostos e analisados transversalmente para identificação de destaques e possibilidades de evolução em cada pesquisa, além de comparação de resultados.

3.1 OTIMIZAÇÃO DE COMUNICAÇÃO EM AMBIENTES *INTERNET OF THINGS*

O trabalho de (POPIĆ *et al.*, 2016) avalia o desempenho da comunicação em um sistema de rastreio de carros em ambiente de *Internet of Vehicles* (IoV), paradigma de comunicação que envolve a comunicação entre um veículo e a nuvem. Desta forma, o objetivo do estudo visa verificar a aplicabilidade de *Binary JSON* (BSON) (MONGODB, 2009) e Protocol Buffers como alternativas mais qualificadas para o cenário de IoV, também comparando com o formato proprietário implementado no sistema avaliado.

Segundo o autor, a arquitetura do sistema de rastreio veicular utilizado como base é protegida por direitos autorais e não pode ser exposta. No entanto, mostra que são trafegados quatro tipos de mensagens. O experimento consiste em observar a troca de mensagens de dispositivos conectados ao servidor. Um processo em paralelo realiza a cópia das mensagens em tempo real e o converte nos três tipos de formato de serialização selecionados, sendo assim salvos em uma base de dados para análise.

Após a análise dos resultados, observou-se que o formato proprietário conseguiu uma taxa de compressão aproximadamente 35:1 menor em comparação com JSON. Protocol Buffers ficou em segundo lugar com taxa de compressão próxima de 5,5:1 mais eficiente em relação à serialização em JSON. BSON, em último lugar, teve taxa de compressão pouco significativa, e não se destacou.

[1] `https://scholar.google.com/`

Ao final, o autor conclui que apesar de Protocol Buffers não ter sido o melhor colocado na comparação, o formato é um bom candidato como ferramenta de serialização, e a melhor escolha dentre os formatos avaliados como forma de se obter interoperabilidade. Ainda inclui que o campo de serialização em IoV pode ser melhor explorado, e novas ferramentas podem ser analisadas, com mais métricas a serem observadas.

O trabalho de (PROOS; CARLSSON, 2020) também realiza uma análise sobre o uso de diferentes protocolos de serialização e também protocolos de troca de mensagem no contexto de comunicação em ambiente de IoV, com foco na manutenção de gêmeos digitais, representações digitais de um veículos armazenados na nuvem, estas que necessitam de soluções de comunicação rápidas e sensíveis ao tempo. O autor investiga os protocolos de troca de mensagens: (i) *Message Queuing Telemetry Transport* (MQTT) (IBM, 1999); (ii) *Advanced Message Queuing Protocol* (AMPQ) (O'HARA, 2013); (iii) *Constrained Application Protocol* (CoAP) (BORMANN; SHELBY, 2014). Quanto aos formatos de serialização, examina Protocol Buffers e FlatBuffers.

Este trabalho se concentra principalmente na comunicação veículo-para-nuvem por meio de *Wireless Fidelity* (Wi-Fi) e redes móveis, como *4th-Generation Wireless* (4G), *Long Term Evolution* (LTE) e *5th-Generation Wireless* (5G). São considerados dois cenários principais: um com grandes pacotes de dados sem restrição de latência e outro com pequenos pacotes com restrição de latência, ambos com variação de carga e perda de pacotes. A serialização é avaliada quanto ao uso de *Random Access Memory* (RAM), tempo de transmissão, ou *Round trip Time* (RTT), eficiência de espaço e capacidade de carga. Como cenário de testes, foi configurado um Raspberry Pi [2] se comunicando com uma máquina virtual Windows com Wireshark acoplado (COMBS, 1998), e NetEm (THE LINUX KERNEL, 2003) na máquina virtual para emulação do tráfego de rede. O cliente e servidor são implementados em Java, e o modelo de fila de mensagem é implementado em RabbitMQ (RABBITMQ, 2007).

Os resultados da análise da serialização mostram que Protocol Buffers apresenta melhor RTT em relação à Flatbuffers. No entanto, o tempo de desserialização de Flatbuffers foi quase imperceptível. Quanto ao tamanho da mensagem, Protocol Buffers teve uma distribuição de tamanho menor. Para o uso de memória, Flatbuffers apresentou menor e mais estável consumo. O autor conclui que Protocol Buffers leva vantagem levando-se em conta a comunicação ponta-a-ponta.

Em conclusão, o autor destaca que Protocol Buffers se mostra mais vantajoso para o contexto de IoV do que FlatBuffers, levando em consideração o cenário onde a comunicação ocorre do veículo para a nuvem. Para o cenário inverso, Flatbuffers, devido à sua desserialização rápida e ao baixo uso de memória, demonstra ser uma opção interessante para comunicação nuvem-para-veículo. No que diz respeito aos

[2] `https://www.raspberrypi.com/`

protocolos de mensagem, CoAP apresenta menor latência, porém não garante a entrega das mensagens, mesmo com mecanismo próprio dede mensagens confirmadas. O protocolo também não se comportou bem com o aumento da taxa de perda de pacotes, também por conta da ineficiência de seu mecanismo de confiabilidade. Os protocolos baseados em *Transmission Control Protocol* (TCP) garantem a confiabilidade na entrega, e MQTT com *Quality of Service* (QoS) de nível zero se destaca pelo baixo *overhead* e latência. Para cenários onde a confiabilidade na entrega não é uma preocupação, o CoAP pode ser escolhido.

3.2 OTIMIZAÇÃO DE SERIALIZAÇÃO PARA ANÁLISE DE PROTEÍNAS EM HPC

A dissertação de (CASEY *et al.*, 2022) aborda o tema de comunicação eficiente em um ambiente de *High Processing Computing* (HPC), paradigma de computação para cenários de alto processamento com diversos nós conectados por meio de uma rede de alta vazão. O autor busca avaliar o desempenho e encontrar um mecanismo de serialização mais eficiente no contexto de HPC das bibliotecas de serialização Protocol Buffers, FlatBuffers e MessagePack em comparação com a serialização padrão na troca de mensagens entre nós de um sistema distribuído de análise de interação de proteínas em organismos. O mecanismo utilizado para a troca de mensagens e coordenação de nós no sistema é o *Message Passing Interface* (MPI), sendo o projeto codificado em C++. Os aspectos considerados são: (i) tempo de execução; (ii) pegada de memória; (iii) utilização de Hardware; (iv) tamanho do código gerado; (v) métricas qualitativas de utilização das bibliotecas.

O funcionamento do pacote de software *HPCMetaMorpheus* baseia-se na coleta de dados de proteínas gerados por um espectrômetro de massa, que são usados para comparação em repositórios genéticos públicos. Cada sequência proteica é repassada aos nós via MPI e o trabalho é paralelizado. Foi criado um mecanismo genérico que permite a implementação e acoplamento dinâmico das bibliotecas por meio de uma interface. Para os testes, é configurado um servidor com distribuição Linux openSUSE[3], e nós de trabalho com configurações idênticas. Os testes variam o número de instâncias a serem serializadas em cada lote.

Analisando os dois cenários de teste, MessagePack apresenta taxa de compressão mais alta, resultando em um tamanho final de *buffer* menor, enquanto o Protocol Buffers ocupa o segundo lugar. Quanto ao RTT, os resultados foram semelhantes. MessagePack obteve o menor tempo de transmissão, seguido por Protocol Buffers e Flatbuffers. Para serialização, enquanto MessagePack teve desempenho rápido e consistente, Protocol Buffers e FlatBuffers apresentaram maior lentidão, com Protocol Buffers fazendo má gestão de memória. No entanto, FlatBuffers desempenhou de

[3] `https://www.opensuse.org/`

forma consistente e sua desserialização teve melhor velocidade, acompanhada de perto por MessagePack. Quanto ao tempo de execução, observou-se que Protocol Buffers é mais lento no processo ponta-a-ponta. Em relação ao uso de recursos, nenhuma biblioteca apresentou comportamento inesperado, e todas tiveram um uso médio semelhante. Protocol Buffers gerou um código adicional maior, enquanto FlatBuffers teve um tamanho reduzido. MessagePack não gera código adicional.

Analisando os aspectos qualitativos de cada implementação, o autor destacou que a documentação de MessagePack exigiu mais esforço para compreensão e navegação, enquanto FlatBuffers e Protocol Buffers apresentaram melhor exposição e demonstração de exemplos. A API de serialização de Protocol Buffers é robusta e oferece um conjunto maior de funcionalidades, obtendo maior destaque.

De forma geral, o autor conclui que MessagePack desempenhou de forma satisfatória em todas as métricas dos cenários analisados. Para desserialização, FlatBuffers apresentou alta eficiência, embora seja mais lento na serialização. Protocol Buffers apresentou variação no uso de memória durante as cargas de teste e gerou uma quantidade significativamente maior de código adicional em comparação com as outras bibliotecas. Por sua vez, MessagePack não precisa gerar código, pois pode ser implementado diretamente na aplicação. Como sugestão para trabalhos futuros, o autor inclui a necessidade de finalizar a paralelização do pacote de software usando Open MPI (GABRIEL *et al.*, 2004), uma biblioteca de troca de mensagens de alto desempenho. Além disso, sugere que sejam realizados mais testes de desempenho para garantir confiabilidade nos testes.

3.3 ANÁLISE CRÍTICA DOS TRABALHOS RELACIONADOS

Nesta análise crítica dos trabalhos relacionados, foram encontrados uma série de estudos significativos que buscaram comparar diferentes estratégias de comunicação eficiente em sistemas. Esta área tem recebido visibilidade constante, e as estratégias podem ser adaptáveis e sujeitas a pesquisa em diversas áreas e cenários. A presente análise discutirá os principais pontos destacados pelos estudos, buscando encontrar similaridades, diferenças, estratégias e desafios encontrados na literatura.

Dentre as pesquisas destacadas neste capítulo observou-se que, seja em ambientes de *Internet of Things* (IoT) (PROOS; CARLSSON, 2020) (POPIĆ *et al.*, 2016), ou em ambientes de HPC (CASEY *et al.*, 2022), todas as bibliotecas selecionadas implementam serialização orientada a esquema. A serialização orientada a esquema geralmente proporciona maior eficiência de espaço e velocidade, devido à ausência da necessidade de enviar informações sobre a estrutura da mensagem juntamente com a mensagem serializada. Apesar da necessidade de se escrever e compartilhar as definições de esquema em cada nó ou dispositivo, para os cenários de IoV ou serialização em HPC para análise de sequência de proteínas, a flexibilidade de uma

serialização não orientada a esquema não foi um fator determinante. Isto indica que a capacidade de versionamento pode ser uma característica importante para utilização em cenários reais.

Nos três trabalhos, foram utilizados os seguintes formatos de serialização: Protocol Buffers (3), FlatBuffers (2), MessagePack (1), BSON (1), JSON (1). Protocol Buffers se destaca entre os trabalhos que envolvem serialização binária eficiente devido à sua ampla utilização e tempo de mercado, sendo considerado às vezes como *baseline* de comparação. Já FlatBuffers foi escolhido devido à sua rápida desserialização e baixo uso de memória. MessagePack chama a atenção devido à sua performance com e sem esquema, sem necessitar de um arquivo de definição, além de ser relativamente novo em comparação com outros formatos testados. Observa-se que, mesmo analisando alguns trabalhos, não há grande variação nos formatos utilizados, nem na quantidade de formatos avaliados em cada trabalho. Novos formatos com grande potencial e formatos consolidados não foram considerados nestes trabalhos, o que poderia ter sido aprofundado e expandido nos cenários analisados.

Os trabalhos que utilizaram bibliotecas em comum para análise não apresentaram divergências em seus resultados, sendo as métricas quantitativas de certa forma parecidas. Os trabalhos (PROOS; CARLSSON, 2020) e (CASEY *et al.*, 2022) apresentaram testes de carga média para demonstrar o comportamento dos mecanismos de serialização em cenários com maior uso. No entanto, nenhum dos trabalhos realizou testes de estresse para analisar a vazão máxima das bibliotecas em seus respectivos ambientes. As métricas quantitativas selecionadas para análise não tiveram muita variação, e o trabalho de (PROOS; CARLSSON, 2020) adicionalmente apresenta gráficos de variância, a fim de identificar inconstâncias na execução da serialização de acordo com a carga. O trabalho (POPIĆ *et al.*, 2016) analisa somente a métrica de tamanho final da mensagem, o que, para o cenário analisado, parecia ser o suficiente. Porém, do ponto de vista de pesquisa, é limitado. Quanto às métricas qualitativas, observou-se que nenhum dos trabalhos estruturou tópicos previamente para análise e apenas analisou sua experiência informalmente, o que pode ser considerado uma oportunidade de melhoria para futuros trabalhos.

O trabalho de (PROOS; CARLSSON, 2020) se destaca por adicionar a eficiência de protocolos de mensagem como mais um vetor de análise na eficiência de comunicação entre sistemas, a fim de identificar também o melhor protocolo possível para este cenário. Foram analisados três protocolos e suas respectivas variações de configuração, juntamente com testes de carga e cenários com variação na taxa de perda de pacotes e tamanho da mensagem. A amostragem de protocolos conseguiu cobrir distintas características entre protocolos, trazendo assim uma boa visualização das possibilidades de implementação.

Todos os trabalhos definiram claramente a estrutura de seus respectivos ambi-

entes de teste, versões de código, configurações de hardware e software das máquinas utilizadas, além da rede utilizada.

Embora tenham sido abordadas bibliotecas e cenários distintos, percebe-se que há campos a serem explorados com mais profundidade no que diz respeito a métricas qualitativas, variedade de mecanismos de serialização e melhores cenários de teste, principalmente no contexto de estresse. Além disso, é possível aprofundar as pesquisas em cenários em que a variação das mensagens é relevante e comparar o comportamento das bibliotecas de forma mais categorizada. Embora a análise dos protocolos de mensagem como parte da problemática da eficiência na comunicação em sistemas distribuídos tenha sido incluída nos trabalhos, existem mais cenários onde essa comparação pode ser realizada.

4 BIBLIOTECAS DE SERIALIZAÇÃO

Após a estruturação da base teórica, foi realizado o estudo aprofundado e descrição dos aspectos de distintas bibliotecas de serialização com relevância para o estado da arte e cenário avaliado, considerando, entre outras informações: características gerais, formato de codificação, tipos aceitos e suas respectivas formas de codificação, comportamento com *streams*, informações de esquema, evolução de esquema e linguagens aceitas. As mesmas estão descritas a seguir.

4.1 APACHE AVRO

Apache Avro (FOUNDATION, T. A. S., 2022a) é um protocolo RPC e especificação de serialização binária orientada a esquema, criada em 2009 por Douglass Cutting[1], sendo parte do arcabouço Apache Hadoop (FOUNDATION, A. S., 2023c) e outros projetos do cenário de *Big Data* como Apache Spark (FOUNDATION, A. S., 2023f) e Apache Kafka (FOUNDATION, A. S., 2023e). Faz parte da Apache Software Foundation (FOUNDATION, A. S., 2023a) e está sob a licença Apache License 2.0 (FOUNDATION, T. A. S., 2004).

Apache Avro se destaca por realizar a serialização e desserialização utilizando o esquema de entrada em tempo de execução, permitindo com que seja possível trabalhar com novos tipos de dados sem a etapa de recompilação. Além disso, sua codificação produz cadeias binárias eficientes em termos de espaço. Sua adoção em sistemas como Apache Hadoop e Apache Kafka, que lidam com múltiplos produtores e consumidores enviando *streams* de dados, fazem com que tenha boa aceitação neste cenário. Apache Avro também suporta evolução de esquema.

Seu formato de codificação forma uma sequência de valores em que a ordem é definida pelo esquema. Codifica apenas valores que não podem ser inferidos por esquema, por exemplo o número de elementos de uma lista. A codificação é realizada também em campos que têm seu valor definido como padrão.

Apache Avro define os tipos aceitos como:

Booleano: valores codificados como inteiros 0 (Falso) ou 1 (Verdadeiro) representados em valores de tipo 0x00 e 0x01, respectivamente. Não há comprimento em bytes para estes valores.

Inteiros: para inteiros com sinal, suporta as variações de 32 bits e 64 bits com codificação Zig-Zag em *Little Endian Base 128* (LEB128) IEEE 754.

Pontos Flutuantes: suporta variações de precisão entre 32 e 64 bits codificados em formato *Big-Endian* (BE) IEEE-754. Números de precisão arbitrária podem ser codificados como *array* de bytes de comprimento fixo ou variável, em que o *ar-*

[1] `https://github.com/cutting`

ray de bytes representa um inteiro com sinal em representação *Big Endian* usando complemento de dois. O esquema também declara a escala e precisão.

Strings: sequência de bytes em UTF-8 não delimitados com o caractere ASCII *NUL*. A codificação define o número de pontos de código dentro de uma *string* prefixando um inteiro com sinal de 64 bits e tamanho variável codificado em Zig-Zag em LEB128. Strings vazias são representadas com um prefixo de tamanho 0 (zero) sem caracteres seguidos. Apache Avro não realiza a deduplicação de múltiplas ocorrências dentro de uma *string*.

Enumerações: constantes inteiros de 32 bits com sinal e tamanho variável, codificados em Zig-Zag LEB128.

Uniões: representadas com um prefixo que define o índice de tipo de dado dentro de uma lista de tipos na união, seguidas de seu valor. O prefixo é um inteiro de 32 bits com sinal e tamanho variável codificado em Zig-Zag LEB128.

Listas (Array): sequência de blocos, cada um prefixado com seu valor lógico em um inteiro com sinal de 64 bits e comprimento variável, codificado como Zig-Zag LEB128, seguido pelos valores contidos no bloco, em ordem. Não são tratadas duplicações em *array*.

Além dos tipos apresentados, a biblioteca também aceita outros tipos:

Mapa: sequência de pares chave-valor. Codificado como uma sequência de blocos, cada um prefixado com seu valor lógico em um inteiro com sinal de 64 bits e comprimento variável, codificado como Zig-Zag LEB128, seguido pelos pares chave-valor contidos no bloco, em ordem.

Record: um registro é uma definição de estrutura, com a declaração de seus campos e respectivos tipos. Sua serialização codifica valores de seus campos em ordem de declaração, de forma concatenada e de acordo com o tipo de cada valor.

Bytes: sequência de 8-bits sem sinal.

Fixed: define um tipo com comprimento de bytes específico. Codificados usando o número de bytes declarados no esquema.

***Universally Unique Identifier* (UUID):** tipo lógico representando um identificador único universal em conformidade com a *Request for Comments* (RFC) 4122[2], codificado da mesma maneira que uma *string*.

***Date*:** representação de data dentro do calendário, sem informações de fuso horário ou hora. Codificado como um inteiro com sinal de 32 bits e comprimento variável codificado em Zig-Zag LEB128 IEEE 754.

***Time*:** representação um horário no dia, sem informações de calendário, data ou fuso horário. Possui informações de precisão de milissegundos ou microssegundos. É codificado como um inteiro com sinal de 32 bits e comprimento variável codificado em Zig-Zag LEB128 IEEE 754. Para precisão de microssegundos, o inteiro terá com-

[2] `https://www.ietf.org/rfc/rfc4122.txt`

primento de 64 bits.

***Timestamp*:** representa um instante na linha temporal global, descrito por meio da diferença em número de milissegundos a partir da data Era Unix (MIZRAHI; FABINI; MORTON, 2020b) como um inteiro com sinal de 64 bits e comprimento variável codificado em Zig-Zag LEB128 IEEE 754. Há variações de precisão que permitem a consideração de fuso horário, em milissegundos ou microssegundos. A codificação é a mesma.

Duração: determina um período de duração de tempo representando três inteiros informando o número em meses, dias e milissegundos, respectivamente. Seu tipo lógico é um *Fixed* com comprimento de 12 bytes, e os três inteiros são codificados como sem sinal de 32 bits e comprimento variável codificado em Zig-Zag LEB128 IEEE 754.

Valores nulos são definidos como não tendo nenhum valor. A representação em bloco de Listas e Mapas permite com que seja possível ler e escrever Mapas e Listas maiores do que o *buffer* de memória alocada, podendo assim serem escritas sem ter conhecimento total do comprimento total da Lista ou Mapa. Além disso, Apache Avro aceita os seguintes valores de algoritmos de compressão: (i) *null* (sem compressão); (ii) *deflate* (IETF, 1996); (iii) *bzip2* (SEWARD, 1996); (iv) *snappy* (GOOGLE, 2011); (v) *xz* (TUKAANI PROJECT, 2009); (vi) *zstandard* (FACEBOOK, 2016).

O esquema é definido em um arquivo *.avsc* em formado de definição *Avro IDL*, para construção da definição de esquema para leitura em tempo de execução. A definição pode ser realizada em um arquivo *.avdl*, enquanto a definição de um protocolo ocorre em arquivo *.avpr*. A especificação está disponível para as linguagens C, C++, C# e Java.

4.2 APACHE THRIFT

Apache Thrift (FOUNDATION, T. A. S., 2022b) é um protocolo RPC e formato de serialização binário orientado a esquema desenvolvido em 2006 pelo Facebook, agora chamado Meta[3], e posteriormente doado à Apache Software Foundation (FOUNDATION, A. S., 2023a). Agora, está sob a licença Apache License 2.0 (FOUNDATION, T. A. S., 2004). O formato é usado atualmente em grande parte dos serviços escaláveis da Meta, assim como em projetos de processamento de dados em larga escala, como no projeto Carat (OLINER *et al.*, 2013) e serviços de bases de dados, como em (CHANG *et al.*, 2015), que desenvolve um serviço de backup em larga escala de *datacenters* usando Apache Thrift nas bases de dados Apache HBase (FOUNDATION, A. S., 2023d) e Apache Cassandra (FOUNDATION, A. S., 2023b). Também há trabalhos explorando Apache Thrift em, cenários de *streaming* em arquitetura de microsserviços

[3] `https://about.meta.com/br`

(GAN; DELIMITROU, 2018).

Apache Thrift tem como principais características o uso dos tipos de dados implementados nativamente nas linguagens, de modo a facilitar a integração e implementação, além do suporte a um grande número de linguagens. Apache Thrift também suporta evolução de esquema por meio do uso dos identificadores de cada campo, permitindo assim a remoção ou adição de propriedades sem quebra de compatibilidade.

Seu formato de cadeia de bits é definido por uma sequência de valores, cada um prefixado pela sua definição de tipo e comprimento, se aplicável. Cada definição de tipo é encapsulada em campo delta contendo um identificador de campo numérico único em seu escopo, e influenciará na escolha do formato de codificação. Então, pode-se considerar a codificação de um campo como sendo uma tripla, contendo um valor Delta, o tipo de campo e valor. (VIOTTI; KINDERKHEDIA, 2022b) observa que valores padrão também são codificados.

O protocolo de compactação *Apache Thrift Compact Protocol* define duas definições possíveis, *Short form* e *Long form*. *Short form* terá 1 byte de tamanho, onde os primeiros 4 bits representam o delta sendo um inteiro com sinal maior que zero, e os 4 bits restantes definem o tipo do campo a partir de valor constante predefinido. Já *Long form* tem de 2 a 4 bytes de tamanho, com os 4 primeiros bits preenchidos com zero, os próximos 4 bits definindo o time, e os bits restantes codificam o identificador único como inteiro com sinal e comprimento variável em Zig-Zag LEB128. Apache Thrift escolherá a forma *Short form* se os campos estão com identificadores únicos declarados, ou o delta do identificador único supera o valor 15. Se não, optará pela forma *Long form*.

Apache Thrift suporta os tipos principais:

Booleano: valores são codificados como inteiros 0 (Falso) ou 1 (Verdadeiro) representados em valores em nível de definição de tipo, sem comprimento em bytes. Se o valor booleano faz parte de uma lista, é codificado como inteiro com sinal de 8 bits com os valores *0x02* (Verdadeiro) e *0x00* (Falso).

Inteiros: para inteiros com sinal, suporta as variações de 32 bits e 64 bits com codificação Zig-Zag em LEB128 IEEE 754 de comprimento variável. Apache Thrift não suporta inteiros de comprimento fixo, porém possui um tipo específico para representar inteiros de 8 bits.

Pontos Flutuantes: suporta somente a variação de 64 bits codificado em formato *Little-Endian* IEEE-754.

***Strings*:** sequência de bytes em UTF-8 não delimitados com o caractere ASCII *NUL*. A codificação define o comprimento em bytes da *string* prefixando um inteiro positivo com sinal de 32 bits e tamanho variável codificado em Zig-Zag LEB128. *Strings* vazias são representadas com um prefixo de tamanho 0 (zero) sem caracteres seguidos. Apache Thrift não realiza a deduplicação de múltiplas ocorrências dentro de uma

string.

Listas: sequência ordenada de elementos, codificadas com a definição de tipo, seguida pela definição da lista, seguido pela concatenação de seus elementos codificados em ordem. Há variações de codificação baseando-se no tamanho da lista, com as delimitações variando entre *arrays* de até 14 elementos e *arrays* que armazenam até $2^{32} - 1$ elementos. No primeiro cenário codifica-se a lista usando um inteiro sem sinal 4 bits para definição do comprimento da lista, seguida por outros 4 bits informando o identificador do tipo do elemento. O segundo tipo de codificação usa de 2 a 6 bytes, com os 4 primeiros bytes de valor *1111*, seguida por outros 4 bits informando o identificador do tipo do elemento. O restante dos bits é usado para registrar o comprimento da lista como um inteiro positivo com sinal de 32 bits e tamanho variável codificado em Zig-Zag LEB128. Apache Thrift não realiza a deduplicação de múltiplas ocorrências dentro de uma *string*. Para registro de listas heterogêneas, é possível usar listas de listas e listas de uniões como alternativa.

Enumerações: constantes inteiros positivos com sinal de 32 bits e comprimento variável, codificados em Zig-Zag LEB128.

Uniões: cada alternativa da união possui um identificador único de tipo. A codificação é definida com base em seu tipo.

Além dos tipos apresentados, a biblioteca também aceita outros tipos:

Mapa: sequência de pares chave-valor. Codificado com o número de elementos do Mapa em inteiro positivo com sinal de 32 bits e tamanho variável codificado em Zig-Zag LEB128, e definições de tipo de chave e tipo de valor distribuídos em inteiro sem sinal de 4 bits cada. Em seguida, é codificado o par chave valor. Não há limite máximo para tamanho de um mapa, sendo o limite assim configurável.

***Struct*:** é uma definição de estrutura, com a declaração de seus campos e respectivos tipos. Sua serialização codifica valores de seus campos em qualquer tipo de ordem, de forma concatenada e de acordo com o tipo de cada valor. Cada campo é dividido entre seu cabeçalho e seu valor, e o cabeçalho por sua vez é dividido entre seu tipo e identificador único. O fim da *struct* em uma serialização é feita com um campo de parada. O protocolo de compactação define variações de acordo com o tamanho da *Struct*, em *structs* de até 15 elementos usando *Short form* e cabeçalho de 8 bits, e *structs* com mais elementos usando *Long form* com cabeçalho de 1 a 3 bytes.

***Set*:** lista não ordenada de elementos únicos, codificadas da mesma maneira que uma lista.

Byte: sequência de 8 bits com sinal codificados em Zig-Zag LEB128.

Binário: codificado como um tipo com comprimento de bytes em inteiro de tamanho variável positivo, seguido do *array* de bytes.

Apache Thrift permite o uso de compressão *zlib* para cenários em que a eficiência de espaço é crítica (ABERNETHY, 2019). Para compilação, usa a ferramenta *thrift*

para transformar as definições de esquema *Thrift IDL* de extensão *.thrift* em código fonte, na linguagem desejada. Está disponível atualmente para as linguagens ActionScript, C, C++, C#, Common LISP, D, Dart, Erlang, Haskell, Haxe, Go, Java, JavaScript, Lua, OCaml, Perl, PHP, Python, Ruby, Rust, Smalltalk e Swift.

4.3 CAP'N PROTO

Cap'n Proto (VARDA, 2022) é um protocolo RPC e formato de serialização binário orientado a esquema desenvolvido em 2013 por um dos autores também de Protocol Buffers, Kenton Varda, atualmente sob a licença MIT (TECHNOLOGY, 1989). Com o objetivo de ser uma evolução a Protocol Buffers, visa corrigir problemas na implementação antiga e melhorias identificadas ao longo do tempo. O formato tem o intuito de promover eficiência de memória na serialização e desserialização por meio de *zero-copy*, e tem como algumas características o bom desempenho em eficiência de memória e execução de leituras incrementais e de acesso randômico, além de uma pegada de código reduzida com poucas dependências e pequeno código de codificação gerado.

Sua estrutura baseia-se no acesso direto à memória por meio de ponteiros baseados em *offsets*, o que permite com que dados sejam acessados sem os passos de codificação e decodificação. Os dados são arranjados de maneira análoga ao trabalho de um compilador, fazendo com que seja possível atingir a interoperabilidade mesmo com acesso à memória. Por padrão, a codificação de inteiros é *Little-Endian*, por considerar que mesmo CPUs *Big-Endian* têm instruções para a leitura de dados *Little-Endian*.

Cap'n Proto permite processar partes do dado antes de o receber de forma completa, visto que objetos externos aparecem antes de objetos internos. Também é possível fazer a leitura de um campo específico por meio do cálculo de *offset* da mensagem, visto que os dados estão dispostos em memória. Cap'n Proto faz a gestão eficiente de memória por meio do uso de uma região, ou arena, de memória para alocação e desalocação, o que é mais rápido e promove localidade de cache. Além disso, pode-se fazer uso da função de mapeamento de memória para a leitura de grandes conjuntos de dados ou *streams*.

Cap'n Proto define sua cadeia de bits, ou mensagem, como uma hierarquia de segmentos de ponteiro em formato de árvore. Cada segmento pode ser do tipo composto ou tipo escalar, sendo que os segmentos devem levar, eventualmente, a um tipo escalar. Cada unidade de informação é definida como uma palavra de 64 bits. Sua definição de esquema baseia-se na enumeração de campos de forma incremental à medida em que são adicionados, o que facilita a interoperabilidade e compatibilidade de esquema para trás, visto que novos campos são sempre adicionados ao final de uma estrutura, ou substituindo um espaço preenchido. O receptor apenas precisa

realizar uma checagem de fronteiras.

A estrutura de um objeto Cap'n Proto consistem em uma palavra de 64 bits de definição de tipo, seguido de uma ou mais palavras de 64 bits do tipo escalar ou do tipo ponteiro para valores compostos. As palavras não necessitam estar dispostas de forma contígua na memória, dado que os 30 bits mais significantes da palavra consistem em um ponteiro para a seção de dados. Os próximos dois bits declaram que a definição de tipo corresponde a uma estrutura, e os 32 bits restantes codificam em formato *Little-Endian* de 16 bits o tamanho em palavras das seções de dados e ponteiros, respectivamente.

Por questões de desempenho, valores escalares são alinhados para ainda assim usarem palavras de 64 bits, e por conta disso a cadeia binária tende a ter um preenchimento significativo de bytes zero. Devido a isso, é disponibilizado um esquema de empacotamento chamado *Packed Encoding* para remoção de zeros à esquerda onde cada palavra de 64 bits é reduzido a um byte de etiqueta, seguido de zero a oito bytes de conteúdo. Os bits do byte etiqueta correspondem aos bytes da palavra desempacotada, com o bit menos significante correspondendo ao primeiro byte. Para sequências de bytes zero na palavra, o empacotamento realiza a compressão usando um byte *0x00* seguido da quantidade de bytes zerados menos 1, representado em um inteiro de 8 bits sem sinal. O esquema de compressão pode codificar dados não empacotados prefixando a palavra descompactada de 64 bits com o byte *0xff* e sufixando-a com a quantidade de palavras descompactadas a seguir como um número inteiro sem sinal de 8 bits.

Cap'n Proto suporta os tipos principais:

***Void*:** representação de valor nulo. Possui somente um valor possível, e pode ser representado e comprimido em zero bits.

Booleano: codificados como 0 (Falso) ou 1 (Verdadeiro) alinhado com um múltiplo de seu valor, na sua estrutura definida. Portanto, como valores booleanos cabem em somente 1 bit, é possível representar 8 valores codificados em um único byte.

Inteiros: suporta as variações de 8 bits, 16 bits, 32 bits e 64 bits, com e sem sinal, codificado em formato *Little-Endian* IEEE-754. Inteiros com sinal utilizam o método de complemento de dois.

Pontos Flutuantes: variações de 32 e 64 bits codificados em formato *Little-Endian* IEEE-754.

Listas: caracterizado por uma definição de tipo em palavra de 64 bits, com os elementos codificados em ordem. Como já reforçado, não é necessário que a definição de tipo e elementos fiquem em espaço contíguo em memória. São definidos dois tipos de codificação, a depender se o tipo de elemento contido na lista é escalar ou composto.

Enumerações: constantes de enumeração são representados por meio de um

inteiro sem sinal de 16 bits, codificado em *Little-Endian*. Portanto, não suporta definição de valores de enumeração negativos.

Uniões: utiliza o artifício de identificadores únicos de campo garantir que somente um valor de união está presente em determinado momento. Por motivos de eficiência de codificação, são definidos dentro de estruturas e não são considerados cidadãos de primeira classe. Não precisam ocupar espaços contíguos de memória.

Além dos tipos apresentados, a biblioteca também aceita outros tipos:

Grupos: conjunto de campos encapsulados em seu próprio escopo. São melhores usados com uniões, em que pode-se definir um conjunto de campos em cada definição de união, o que promove organização e facilita a adição de novos campos em estruturas específicas de uma união. Assim como uniões, não precisam ocupar espaços contíguos de memória.

Bolhas: Podem ser do tipo *Text* ou *Data*. *Text* deve ser codificado em UTF-8 e delimitados com o caractere ASCII *NUL*, enquanto *Data* é uma sequência arbitrária de bytes.

Tanto a biblioteca em tempo de execução quanto o código gerado são de pequeno tamanho. Cap'n Proto também permite a compressão de propósito geral, para cenários em que a largura de banda é importante ou a mensagem pode conter dados repetidos, como em *blobs* de texto grandes. Alguns formatos de compressão aceitos são *LZ4* e *zlib*.

Cap'n Proto usa a ferramenta de compilação *capnp* para transformar as definições de esquema escritos em arquivo *.capnp* para código fonte, na linguagem desejada. Está disponível atualmente para as linguagens C, C++, C#, D, Erlang, Go, Haskell, Java, JavaScript, Lua, Nim, OCaml, Python, Ruby, Rust e Scala.

4.4 FLATBUFFERS

Flatbuffers (GOOGLE, 2022a) é um protocolo RPC e formato de serialização binário orientado a esquema fortemente tipado, criado pela Google em 2014, com o objetivo de melhorar o desempenho de tecnologias para jogos na plataforma *Android*. Seu propósito principal reside na serialização e desserialização eficientes em termos de memória, no contexto de jogos e plataformas mobile, além de aplicações de desempenho crítico. Flatbuffers também é usado no contexto de *Machine Learning* como parte do arcabouço *TensorFlow Lite* [4] como arcabouço para dispositivos IoT desenvolvidos pela Google (USHAKOV *et al.*, 2018) e no sistema espacial *Distance Oracle System* (DOS) (PENG; SANKARANARAYANAN; SAMET, 2018). Foi lançado sob a licença Apache License 2.0 (FOUNDATION, T. A. S., 2004). O formato é geralmente é comparado com Cap'n Proto, devido aos princípios similares no formato de

[4] `https://www.tensorflow.org/lite/api_docs/cc/class/tflite/flat-buffer-model`

serialização[5].

Flatbuffers destaca-se por produzir implementações que tem desempenho eficiente em tempo de execução e em termos de memória, tanto para leituras incrementais como para leituras de acesso randomizado. Dado ao seu desempenho eficiente no processo de desserialização utilizando *zero-copy*, (UK *et al.*, 2018) sugere sua utilização em sistemas de transmissão de video e dados em veículos não tripulados em cenários de desastres naturais. Sua biblioteca tem dependências mínimas em tempo de execução e gera uma quantidade pequena de código fonte, como aferido em (CASEY *et al.*, 2022) e (PROOS; CARLSSON, 2020). Também oferece compatibilidade para frente e para trás, permitindo um grande flexibilidade na gestão dos dados, em conjunto com uma definição de tipagem forte que garante maior taxa de identificação de erros em tempo de compilação, e não em tempo de execução. Além disso, funciona em multiplataforma, promovendo interoperabilidade.

A cadeia binária de Flatbuffers define-se por uma hierarquia em formato de árvore de ponteiros de 32 bits escalares ou compostos, iniciando por um ponteiro que aponta ao elemento raiz. Eventualmente, os ponteiros devem apontar a tipos escalares. A estrutura principal de dados é uma Tabela, que é uma sequência ordenada de valores alinhados prefixados com um ponteiro para uma estrutura de *vtable* que define o *layout* da Tabela. Uma *vtable* consiste em dois inteiros de 16 bits sem sinal codificados em *Little-Endian* que descrevem o comprimento de bytes da *vtable*, o comprimento de bytes da tabela, seguidas por uma sequência de *offsets* inteiros de 16 bits sem sinal para cada elemento na tabela, em relação à *vtable*. Em resumo, a *vtable* aponta os *offsets* de cada campo na tabela.

Como alternativa ao uso de Tabelas, Flatbuffers suporta o conceito de *structs* como sendo uma sequência de suas propriedades escalares ou de outras estruturas, alinhadas ao seu maior elemento escalar que possui. *Structs* são mais eficiente que Tabelas, porém não disponibilizam recursos como versionamento e extensibilidade de *features*.

Flatbuffers suporta os tipos principais:

Booleano valores codificados como 0 (Falso) ou 1 (Verdadeiro) em representação de inteiro sem sinal *Little-Endian*.

Inteiros: suporta as variações de 8 bits, 16 bits, 32 bits e 64 bits, com e sem sinal, codificado em formato *Little-Endian* IEEE-754. Inteiros com sinal utilizam o método de complemento de dois.

Pontos Flutuantes: variações de 32 e 64 bits codificados em formato *Little-Endian* IEEE-754.

Strings e Vetores: strings podem ser consideradas como um Vetor de bytes em UTF-8 e delimitados com o caractere ASCII *NUL*. Vetores são armazenados como

[5] `https://capnproto.org/news/2014-06-17-capnproto-flatbuffers-sbe.html`

uma sequência contígua escalar de elementos prefixados por um inteiro sem sinal de 32 bits codificado em *Little-Endian* que representa o comprimento em bytes da *string*, sem levar em consideração o caractere ASCII *NUL*. Strings vazias são representadas como tendo comprimento zero, logo seguidas por caractere ASCII *NUL*. Flatbuffers, por sua vez, não realiza tratamento na duplicação de múltiplas ocorrências de uma mesma *string*, no entanto disponibiliza uma interface para que o usuário identifique e compartilhe valores duplicados.

Listas: conjunto de elementos de tipo parametrizável, em que o parâmetro é o tipo do elemento. Caracterizado pela concatenação dos seus elementos prefixados pelo tamanho lógico do vetor como um inteiro de 32 bits sem sinal em *Little Endian*. Cada vetor de elemento é representado como um ponteiro de 32 bits. Assim como para vetores, não são tratadas múltiplas ocorrências de mesmo valor, porém a interface de serialização permite o tratamento.

Enumerações: constantes de enumeração são representados por meio de um inteiro sem sinal de 16 bits, codificado em *Little-Endian*. Portanto, não suporta definição de valores de enumeração negativos.

Uniões: codificam-se uniões por meio da combinação de dois valores, sendo o primeiro representando a alternativa de valor de união, seguido pelo *offset* do valor da união. O valor 0 é reservado para definir que o valor não foi definido. Não suportam tipos escalares, porém pode aceitar uma *struct* contendo um único valor escalar dentro, sem causar *overhead* de espaço.

O fluxo de serialização de Flatbuffers ocorre de forma que objetos internos sejam criados primeiro e armazenados por último, enquanto objetos externos sejam criados por último, e armazenados antes. Isto faz com que não seja possível processar partes do dado antes de o receber de forma completa, porém há o conceito de *Stream Buffers*, que busca permitir o envio de dados parciais de um *buffer* por meio da rede.

Observa-se que Flatbuffers também funciona em CPUs *Big-Endian*, porém com desempenho ligeiramente menor, devido a troca de bytes adicionais intrínsecos. Flatbuffers também suporta a definição de campos opcionais, em que caso um campo opcional não tenha valor, ele é apenas considerado nulo por padrão. Portanto, é possível enviar um objeto com apenas alguns valores definidos.

Em se tratando de evolução de esquema, cada campo dentro de uma tabela possui um identificador, e novos campos são adicionados ao final da tabela, resultando em *vtables* maiores. Para enumerações, não é possível mudar valores em versões futuras, enquanto Uniões somente permitem a adição de forma sufixada de um novo campo na lista de União. Uma *Struct* não pode alterar de tamanho nem conteúdo, portanto não são evoluíveis. Campos obrigatórios não podem deixar de ser obrigatórios, nem depreciados.

Para transformar as definições de esquema em arquivo de extensão *.fbs* em có-

digo fonte na linguagem desejada, FlatBuffers usa a ferramenta de compilação *FlatCC*. Existem bibliotecas disponíveis para C++, C#, C, Go, Java, Kotlin, JavaScript, Lobster, Lua, TypeScript, PHP, Python, Rust e Swift.

4.5 MESSAGEPACK

MessagePack (FURUHASHI, 2008) é um formato de serialização binário orientado ou não a esquema, desenvolvido por Sadayuki Furuhashi[6] em 2009. MessagePack é usado como forma de representação de dados interna em sistemas como Fluentd[7], um coletor de dados para camada de *logging* unificada em código aberto, e também Pinterest[8]. Além disso, também possui suporte desenvolvido para Redis (ANTIREZ, 2009). Está registrado sob a licença Apache License 2.0 (FOUNDATION, T. A. S., 2004).

MessagePack tem como características principais uma especificação de simples compreensão e implementação do ponto de vista de desenvolvimento, como citado em (OLEG *et al.*, 2019). Também, a possibilidade de poder implementar um esquema, parcialmente ou totalmente, dentro do código fonte do sistema, é uma vantagem. Sua popularidade resulta em uma grande gama de bibliotecas e implementações de terceiros de sua especificação, o que resulta na abrangência de mais de quarenta linguagens de programação, ultrapassando por muito concorrentes como FlexBuffers (OORTMERSSEN, 2017) e Microsoft Bond (MICROSOFT, 2018).

Sua estrutura de cadeia binária baseia-se numa sequência de pares chave-valor, podendo ser aninhada ou não. Cada chave ou valor deve ser prefixada com uma definição de tipo. Um mapa, ou objeto é o resultado da concatenação de seus pares chave-valor. Cada definição de tipo de elemento ocupa de 1 a 9 bytes na cadeia, a depender do tipo de dado. Se aplicável, a definição de tipo é acompanhada de um inteiro sem sinal codificado em formato *Big-Endian* representando o comprimento em bytes do elemento, sendo o comprimento do inteiro sem sinal determinado pela primeira parte da definição de tipo.

MessagePack suporta os tipos principais:

Booleano: codificado como inteiro 0 (Falso) ou 1 (Verdadeiro) representados em valores hexadecimais *0xc2* e *0xc3*, respectivamente. Não há comprimento em bytes para estes valores.

Inteiros: para inteiros sem sinal, suporta as variações de 7 bits, 8 bits, 16 bits, 32 bits e 64 bits. Para inteiros com sinal, suporta as variações de 5 bits, 8 bits, 16 bits, 32 bits e 64 bits, todos em codificação *Big-Endian* IEEE-754. Inteiros com sinal usam complemento de dois. Como forma de trazer eficiência de espaço. Inteiros sem

[6] `https://github.com/frsyuki`
[7] `https://www.fluentd.org/`
[8] `https://www.pinterest.com/`

sinal com valor menor que 128 e inteiros com sinal de valor maior ou igual a -32 são codificados como inteiros de 8 bits.

Pontos Flutuantes: variações de precisão entre 32 e 64 bits codificados em formato *Big-Endian* IEEE-754.

Strings: sequência de bytes em UTF-8 não delimitados com o caractere ASCII *NUL*. Há 5 variações de codificação para otimização de espaço, a depender do comprimento da *string*. As representações de tamanho da *string* em bytes variam entre 5 bits, 8 bits, 16 bits, 32 bits e 64 bits, codificados como inteiros sem sinal e precedidos por seus respectivos valores de tipo. Seu comprimento máximo em bytes é de $2^{32} - 1$ bytes.

Listas: as listas são codificadas como uma concatenação de seus elementos prefixado com uma definição que inclui o tamanho lógico do *array*. Há variações de codificação baseando-se no tamanho do *array*, com as delimitações variando entre *arrays* de até 15 elementos, *arrays* de 16 a $2^{16} - 1$ elementos, e *arrays* que armazenam até $2^{32} - 1$ elementos.

Enumerações: não há codificação específica para representação de enumerações.

Uniões: não há codificação específica para representação de uniões.

Além dos tipos apresentados, a biblioteca também aceita outros tipos:

Mapas: MessagePack define o tipo Mapa como uma sequência de pares chave-valor, que pode ser representado em variações de 4 bits, 16 bits ou 32 bits para caracterização do comprimento em bytes do Mapa. O número máximo de elementos permitidos em um mapa é $2^{32} - 1$ elementos.

***Bin* :** o tipo de extensão *Bin* representa um *array* de bytes com representação de comprimento seu variando entre 8 bits, 16 bits ou 32 bits, codificados como inteiro sem sinal *Big-Endian*. O comprimento máximo em bytes de um Bin é de $2^{32} - 1$ elementos.

TimeStamp: *TimeStamp* (MIZRAHI; FABINI; MORTON, 2020a) é um tipo de extensão usado para representações de tempo com nanosegundos, possuindo variações codificação de 32 bits, 64 bits e 96 bits para diferentes tipos de precisão de tempo.

Há um valor específico para representação de nulo chamado Nil. Este possui tipo com valor *0xc0* e não tem comprimento em bytes.

MessagePack tem uma implementação que tenta sempre atribuir a um determinado valor o seu menor tipo de dado possível que ainda possa codificar, como forma de trazer eficiência de espaço. Não realiza o tratamento de duplicatas. Não possui mecanismo específico para *streams*. Atualmente há bibliotecas que implementam MessagePack nas linguagens ActionScript, C, C++, C#, Clojure, Crystal, D, Dart, Delphi, Elixir, Erlang, F#, Go, GNU Guile, Haskell, Haxe, HHVM, J, Java, JavaScript, Julia, Kotlin, Nim, MATLAB, OCaml, Objective-C, Pascal, PHP, Perl, Pony, Python, R, Racket, Ruby, Rust, Scala, Scheme, Smalltalk, SML e Swift.

4.6 PROTOCOL BUFFERS

Protocol Buffers (GOOGLE, 2022b) é um protocolo RPC e especificação de serialização binária desenvolvido pela Google, que o utiliza em grande parte de suas aplicações e necessidades. Posteriormente, foi lançada a versão 2 'Proto2' do protocolo que teve uma completa reescrita, com adição de *features* e melhorias na otimização. Atualmente o protocolo está na versão 3 'Proto3', lançado em 2016. Como uma especificação amplamente usada na indústria, Protocol Buffers é objeto de diversas pesquisas e implementações. Melhorias e estudos de caso do protocolo envolvem a otimização para o contexto de data centers, eficiência em tempo de execução por meio da aceleração de hardware. Protocol Buffers também é um componente central do arcabouço de aprendizagem profunda *Caffe* (JIA *et al.*, 2014), desenvolvido pela *Universidade de California, Berkeley*[9], em código aberto sob a licença *Berkeley Software Distribution* (BSD) (THE REGENTS OF THE UNIVERSITY OF CALIFORNIA, 1999).

Protocol Buffers define sua cadeia binária como uma sequência de pares chave-valor. Cada registro é estruturado com o um identificador numérico único em seu escopo, seguido por uma definição de tipo de fio e seu *payload*, se aplicável. A ordem dos campos na cadeia binária ocorre de forma não determinística, portanto, não é possível acessar partes da mensagem antes de sua completa desserialização. Cada registro de mensagem possui de forma prefixada o seu tipo de fio, que serve para agrupar características de tamanho do *payload*. Atualmente, há 4 tipos de fios disponíveis, descritos na Tabela 1 a seguir:

Tabela 1 – Tipos de fio disponíveis em *Protocol Buffers*

Identificador	**Identificador Binário**	**Nome**	**Tipos disponíveis**
0	*000*	*VARINT*	Booleanos, enumerações, inteiros de 32 bits e 64 bits de tamanho variável com e sem sinal em LEB128 IEEE-754
1	*001*	*I64*	Inteiros de 64 bits de tamanho fixo com e sem sinal em LEB128 IEEE-754, pontos flutuantes de 64 bits em formato *Little Endian* IEEE-754
2	*010*	*LEN*	Bytes, mensagens, strings e listas
5	*101*	*I32*	Inteiros de 32 bits de tamanho fixo com e sem sinal em LEB128 IEEE-754, pontos flutuantes de 32 bits em formato *Little Endian* IEEE-754

O protocolo tem como principais características a robustez, trazida pela ampla utilização e testagem em implementações e ambientes de produção Google de alta

[9] `https://www.berkeley.edu/`

escala, segurança e popularidade. Protocol Buffers pode ser descrito como um dos formatos de serialização orientado a esquema mais populares e relevantes no contexto de serialização binária. Como resultado, é acompanhado de ótima documentação, ferramentas relevantes e uma comunidade ativa.

Protocol Buffers suporta os tipos principais:

Booleano: valores codificados como inteiros 0 (Falso) ou 1 (Verdadeiro) em representação de inteiro de tamanho variável em LEB128. Sendo assim, o resultado é um byte codificado com a constante hexadecimal *0x00* ou *0x01*.

Inteiros: suporta as variações de 32 bits e 64 bits, com e sem sinal, de comprimento fixo e variável. Para inteiros de comprimento fixo sem sinal e variável sem sinal, é usada codificação LEB128 IEEE-754. Inteiros com sinal e comprimento variável utilizam o método de Complemento de Dois em formato LEB128 IEEE-754. Enquanto isso, inteiros de comprimento variável e com sinal, é usada codificação Zig-Zag.

Pontos Flutuantes: Variações de 32 e 64 bits codificados em formato *Little-Endian* IEEE-754.

Strings: Strings são uma sequência de bytes em UTF-8 não delimitados com o caractere ASCII *NUL*, codificados usando o tipo de fio *length-delimited*. A *string* é armazenada como uma sequência de elementos prefixados por um inteiro sem sinal de comprimento variável de 32 bits codificado em LEB128 que representa o comprimento em bytes. Para strings vazias, codifica a sequência com um prefixo de comprimento zero e nenhum dado adicional. Protocol Buffers não faz o tratamento de strings de múltipla ocorrência na mesma *string*.

Listas: as listas em Protocol Buffers são codificados por meio do registro de mais de um valor com o mesmo identificador único. As definições de tipo correspondentes a cada elemento na lista usam o tipo de fio *length-delimited*, incluindo o comprimento de bytes como um inteiro de 32 bits de tamanho variável sem sinal em LEB128. Listas de tipos escalares de mesmo tipo podem ser codificadas como *length-delimited*, seguido pelo comprimento cumulativo de bytes dos elementos como um inteiro de 32 bits de tamanho variável sem sinal em LEB128. Assim como para strings, não são tratadas múltiplas ocorrências de mesmo valor. Listas heterogêneas ou multidimensionais não são suportadas, porém é possível utilizar uma lista de *structs* que possuam Listas ou Uniões.

Enumerações: constantes de enumeração são representados por meio de um inteiro de comprimento variável com sinal de 32 bits, codificado em LEB128 com complemento de dois. Desta forma, suporta definição de valores de enumeração negativos.

Uniões: baseia-se no uso de identificadores únicos para a implementação dos tipos em uma união chamados *oneof*. Sendo assim, cada tipo em uma união deve possuir um identificador próprio presente em determinado momento.

Valores nulos são codificados com os respectivos valores padrão de cada tipo.

Protocol Buffers não foi projetado ou incluem qualquer suporte incorporado para lidar com a transmissão de grandes conjuntos de dados, de forma que quanto a mensagem ultrapassa um megabyte, é interessante considerar outras alternativas de serialização. O formato considera que uma mensagem inteira pode ser carregada para dentro de um espaço de memória, de uma vez só. No entanto, é possível contornar a situação ao fragmentar a mensagem e enviar pequenos pacotes de dados. Adicionalmente, ao realizar uma transmissão de mensagens em um arquivo ou fluxo, é preciso que o controle de delimitação de início e fim de mensagem seja feito pelo desenvolvedor. Pelo fato do protocolo não ser auto delimitador, os *parsers* não são capazes de fazer o controle.

Protocol Buffers não realiza compressão de mensagens, portanto não é possível usar algoritmos de compressão. Por considerar que uma mensagem inteira pode ser carregada para dentro de um espaço de memória, é possível que haja uso excessivo de memória por conta da quantidade de cópias de dados durante a serialização. Também, encontra dificuldades em atingir eficiência em cenários em que a distribuição de dados possui grandes *arrays* multidimensionais de pontos flutuante, e para cenários desse tipo, é interessante considerar outras opções. O protocolo implementa compatibilidade de esquema, porém é preciso primeiro verificar peculiaridades nos tipos de dados.

Protocol Buffers usa a ferramenta de compilação *protoc* para transformar as definições de esquema, escritos em arquivos *.proto*, em código fonte na linguagem desejada. Existem bibliotecas disponíveis para C++, C#, Dart, Go, Java, Objective-C, Python, Ruby, Kotlin e PHP.

4.7 COMPARAÇÃO E AVALIAÇÃO DE BIBLIOTECAS

Com base no estudo aprofundado feito sobre as bibliotecas de serialização, é possível definir e observar características gerais de cada linguagem, de modo que seja possível indicar se as mesmas podem ser candidatas interessantes para o estudo. De modo inicial, os critérios observados para escolha das bibliotecas foram: (i) adesão em grande sistemas; (ii) desempenho geral; (iii) abrangência em linguagens; (iv) coerência com o cenário avaliado.

Na Tabela 2, é feita uma agregação das principais características de cada biblioteca, de modo a facilitar a compreensão sobre as capacidades das bibliotecas e orientar a seleção para o estudo. Cada biblioteca conterá informações principais quanto ao propósito, formato de dados e técnicas utilizadas no processo de serialização.

A Tabela 3 continua a exposição das informações sobre cada biblioteca, indicando as características de *zero-copy*, tipo de codificação, uso de *padding* e suporte a RPC. Cap'n Proto utiliza *padding*, porém possui compactação adicional de modo a mitigar os efeitos do enxerto. MessagePack apresenta suporte a RPC somente para algumas linguagens.

Tabela 2 – Esquema, Propósito e Formato das Bibliotecas de Serialização

Biblioteca	**Esquema**	**Propósito**	**Formato**
Apache Avro	Não, Sim	Big Data, Intercâmbio de Dados	JSON, Binário
Apache Thrift	Sim	Comunicação, Interoperabilidade	JSON, Binário, Compacto, Multiplexado
Cap'n Proto	Sim	Comunicação, Interoperabilidade	JSON, Binário
FlatBuffers	Sim	Jogos e Mobile	JSON, Binário
MessagePack	Não, Sim	Geral	JSON, Binário
Protocol Buffers	Sim	Geral	JSON, Binário

Tabela 3 – *Zero-copy*, Codificação, *Padding* e RPC das Bibliotecas de Serialização

Biblioteca	**Zero-copy**	**Codificação**	**Usa Padding**	**Suporta RPC**
Apache Avro	Não	Zig-Zag, LEB128 , BE	Sim	Sim
Apache Thrift	Não	Zig-Zag, LEB128	Sim	Sim
Cap'n Proto	Sim	LE	Sim	Sim
FlatBuffers	Sim	Zig-Zag, LEB128	Não	Não
MessagePack	Não	Zig-Zag, LEB128 , BE	Não	Sim
Protocol Buffers	Não	LE IEEE754[10]	Sim	Não

A Tabela 4 apresenta uma matriz de suporte para compreender as capacidades e limitações nativas das bibliotecas, de modo a observar possíveis cenários e características em que cada uma pode se destacar ou não ser a escolha ideal. O símbolo de adição seguido de um número representa a quantidade de linguagens a mais que cada formato está disponível, sendo possível consulta na seção de cada biblioteca. Observa-se que a coluna CPU apresenta se o formato possui serialização nativa em BE ou LE. Apesar disso, é possível que as bibliotecas que implementam cada formato podem fazer adaptações para diferentes arquiteturas.

Tabela 4 – Suportes Nativos de Bibliotecas de Serialização

Características	**CPU**	***Streams***	**Evolução de Esquema**	**Compressão**	**Linguagens**
Apache Avro	LE, BE	Sim	Frente, Trás	Nativa	C#, Java + 2
Apache Thrift	LE, BE	Sim	Frente, Trás	Nativa	C#, Java + 20
Cap'n Proto	LE	Sim	Frente, Trás	Nativa, Externa	C#, Java + 14
FlatBuffers	LE, BE	Sim	Frente, Trás	Externa	C#, Java + 12
MessagePack	BE	Sim	Não	Externa	C#, Java + 38
Protocol Buffers	LE	Não	Frente, Trás	Externa	C#, Java + 8

Observando as características de cada biblioteca, foi observado que, de forma geral, todas já foram implementadas em cenários de larga escala e são usados comercialmente, conforme apresentado na literatura. Isto traz mais segurança de que

possuem robustez e garantem confiabilidade para assumir cargas de trabalho de cenários distribuídos de alta vazão. Flatbuffers, apesar de ter mais implementações em jogos e aplicativos *mobile*, apresentou desempenho interessante mesmo em outras esferas, como demonstrado no Capítulo 3. Todas as bibliotecas apresentaram suporte às linguagens C++, C# e Java, demonstrando que podem ser ser implementadas em linguagens amplamente utilizadas. Há bibliotecas que possuem alta portabilidade em linguagens, podendo-se elencar MessagePack e Apache Thrift.

Devido à natureza do trabalho que busca melhor desempenho na comunicação, os formatos escolhidos costumam apresentar serialização binária orientada a esquema. MessagePack, apesar de ser originalmente um formato não orientado a esquema, permite flexibilidade no detalhamento de definições de esquema, o que se mostrou interessante no quesito de facilidade de adesão.

Desta forma, portanto, exploram-se as capacidades das bibliotecas de serialização: MessagePack (FURUHASHI, 2008), Apache Avro (FOUNDATION, T. A. S., 2022a), Apache Thrift (FOUNDATION, T. A. S., 2022b), Protocol Buffers (GOOGLE, 2022b), FlatBuffers (GOOGLE, 2022a), e Cap'n Proto (VARDA, 2022). Os formatos escolhidos se mostraram bons candidatos para análise de desempenho em *microbenchmark* e comunicação distribuída, com possibilidade de riqueza de extração de resultados.

Os formatos Microsoft Bond (MICROSOFT, 2018) e ASN.1 (ITU TELECOMMUNICATION STANDARDIZATION SECTOR, 1984) foram avaliados, porém alguns critérios da avaliação não foram atendidos, entre eles adesão em grandes sistemas e coerência com cenário avaliado.

5 AVALIAÇÃO EXPERIMENTAL

A realização deste trabalho consistiu na aplicação do método de avaliação experimental para análise e comparação do desempenho de diferentes bibliotecas de serialização no contexto de aplicações distribuídas, especificamente no modelo de arquitetura cliente-servidor. Foram extraídas métricas de desempenho em diversos aspectos de cada biblioteca de serialização em um cenário de sistema, de formas isolada e distribuída, com variação de carga e concorrência. Paralelamente, foi feita uma análise das características principais de cada biblioteca, assim como aspectos de implementação das mesmas no protótipo desenvolvido.

Esta pesquisa não envolve o uso de dados pessoais ou questões de privacidade, portanto não há considerações éticas a serem observadas.

5.1 CENÁRIOS E MÉTRICAS DE AVALIAÇÃO

Nesta seção, são estabelecidas as configurações do sistema e do ambiente, bem como as métricas de avaliação, para a realização dos testes de desempenho. Os testes serão conduzidos em duas abordagens: em um contexto de *microbenchmark* e em cenário distribuído. Os cenários definidos têm como objetivo compreender as capacidades de cada formato e replicar o funcionamento de sistemas distribuídos com alto tráfego de dados, abrangendo diferentes volumes de dados e níveis de concorrência. É utilizado um formato de serialização em JSON da linguagem escolhida como base de comparação, para análise de desempenho.

Após a implementação e teste dos protótipos, são apresentadas métricas qualitativas quanto aos aspectos de: (i) funcionalidades disponíveis; (ii) qualidade da documentação; (iii) facilidade de implementação e integração com o sistema existente (iv) interoperabilidade com diferentes linguagens de programação e plataformas.

Ao final da avaliação, foi possível obter uma visão abrangente do desempenho e adequação das bibliotecas de serialização avaliadas em sistemas distribuídos, permitindo a identificação de suas vantagens, limitações e possíveis aplicações dentro dos cenários avaliados.

5.1.1 *Microbenchmark*

Serialização individual: aqui visa-se obter informações quanto à rapidez e taxa de compressão de cada serializador considerando somente 1 objeto por vez, em cenário que exige pouco poder computacional. *Métricas:* (i) Tamanho final da mensagem; (ii) Uso de memória RAM para serialização e desserialização; (iii) *Makespan* de serialização; (iv) *Makespan* de desserialização.

Múltiplas serializações em série: neste caso, são aplicadas variações quanto

ao número de mensagens a serem serializadas de uma única vez, sem o uso de concorrência. Aqui são trabalhados cenários em que o fator a ser observado é o comportamento do serializador com maior volume de objetos, para medir a eficiência de serialização em cenários com maior exigência de recursos. *Número de mensagens:* 128.000, 256.000, 384.000, 512.000, 640.000, 768.000, 896.000, 1.024.000, 1.152.000 e 1.280.000. *Métricas:* (i) *Makespan* de serialização; (ii) *Makespan* de desserialização.

Múltiplas serializações concorrentes: Alternam-se os parâmetros de quantidade de objetos e *threads* que disputarão a serialização. Aqui é observado como o serializador trabalha em cenários de exclusão mútua quando múltiplos nós estão disputando o serviço. Serão instanciadas *threads* para a serialização e desserialização do conjunto de mensagens, com o uso de concorrência. *Número de mensagens:* 1.280.000. *Número de threads:* 2, 4, 8 e 16. *Métricas:* (i) *Makespan* de serialização; (ii) *Makespan* de desserialização.

5.1.2 Modelo de Comunicação Cliente/Servidor

Foi realizada a utilização de chamadas via API REST (FIELDING, 2000) em controladores *Hypertext Transfer Protocol* (HTTP) para implementar a comunicação entre o cliente e o servidor de forma síncrona, em que o nó servidor é responsável por receber e lidar com uma ou mais requisições enviadas pelos nós clientes. O protocolo de comunicação estabelecido na camada de aplicação é HTTP, e há métodos *POST* disponíveis como interface entre cliente e servidor, e controle de métricas de avaliação. Na camada de transporte, o protocolo utilizado é TCP. Não foi necessário implementação de mecanismo de persistência, exceto para armazenar as requisições para o gerador de carga. Não está sendo aplicado mecanismo de escalabilidade de servidor, ocorrendo assim somente para o número de clientes simultâneos. Não há mecanismos de segurança e autenticação implementados. Não é implementado mecanismo de tolerância à falhas. É importante mencionar que o cenário implementado não simula com fidelidade o comportamento de uma comunicação entre sistemas pela rede, visto que tratou-se de uma comunicação dentro do próprio hospedeiro, não acionando todos os mecanismos e camadas de rede comumente ativados em comunicação entre diferentes hospedeiros.

Para extração das métricas de avaliação, é observado o comportamento dos nós que disparam requisições HTTP e o nodo que recebe as mensagens. Por meio da ferramenta de geração de carga, são criados usuários virtuais, como forma de representação semelhante a *threads* que podem enviar requisições HTTP simultâneas enquanto o teste ocorre. Os clientes têm a característica de apresentar um *thinking time* de 100ms, fazendo assim que que haja cadência na geração de requisições HTTP. Para o servidor que as recebe, observa-se a vazão, que é medida no lado servidor, e representa a quantidade de requisições executadas em um determinado período de

tempo. A latência, por sua vez, é medida no cliente, e representa o tempo entre o envio de uma requisição e o recebimento de sua resposta. *Número de clientes simultâneos:* 200, 500, 1000, 2000, 3000, 3500. *Métricas para o servidor:* (i) Vazão na serialização; (ii) Vazão na desserialização. *Métricas para o cliente*: (i) Latência na serialização; (ii) Latência na desserialização.

5.2 FORMATO DE DADOS

O sistema avaliado conta com uma estrutura de dados que visa representar uma plataforma de compartilhamento de vídeos. O objeto principal é uma mensagem de nome `Video`, que conta com informações sociais (`SocialInfo`), do canal autor (`Channel`), além de informações referentes ao vídeo em questão (`VideoInfo`). Usando a ferramenta de descrição de formato de dados *JSON Schema* [1], foi possível extrair em JSON a representação dos formatos de dados usados:

A classe `Channel`, apresentada na Figura 1, representa as informações base do canal autor de um vídeo. O objeto contém o nome do canal de comprimento em caracteres variando entre 10 e 80 caracteres (`Name`), seu número de inscritos (`Subscribers`) e seu identificador único universal em conformidade com a RFC 4122[2] (`ChannelId`).

Figura 1 – Descrição da classe `Channel` em formato *JSON Schema*

```
"properties": {
  "Name": { "type": "string", "miminum": 10, "maximum": 80 },
  "Subscribers": { "type": "integer", "minimum": 0,
    "maximum": 1000000000 },
  "ChannelId": { "type": "string", "format": "uuid" }
},
"required": ["Name", "Subscribers", "ChannelId"]
```

Fonte: Elaborado pelo autor (2023).

As informações que definem dados sociais, como número de visualizações e comentários de cada vídeo, estão contidas em `SocialInfo`, apresentado na Figura 2. O objeto contém o número de curtidas do vídeo (`Likes`), seu número de não curtidas (`Dislikes`), número de visualizações (`Views`) e uma lista de comentários (`Comments`):

A classe que define as informações gerais do vídeo, `VideoInfo` está descrita na Figura 3. A classe contém a duração do vídeo em milissegundos (`Duration`) com duração de até 10 dias, uma descrição associada entre 50 e 3000 caracteres (`Description`), o tamanho do arquivo de vídeo (`Size`) de até 1 *terabyte* (TB), e uma lista de itens representando as qualidades de visualização disponíveis (`Qualities`):

[1] `https://json-schema.org/`
[2] `https://www.ietf.org/rfc/rfc4122.txt`

Figura 2 – Descrição da classe `SocialInfo` em formato *JSON Schema*

```
"properties": {
  "Likes": { "type": "integer", "minimum": 0 },
  "Dislikes": { "type": "integer", "minimum": 0 },
  "Comments": { "type": "array", "items": { "type": "string",
    "mininum": 5, "maximum": 300 }, "minItems": 1, "maxItems": 10 },
  "Views": { "type": "integer", "minimum": 0 }
},
"required": ["Likes", "Dislikes", "Comments", "Views"]
```

Fonte: Elaborado pelo autor (2023).

Figura 3 – Descrição da classe `VideoInfo` em formato *JSON Schema*

```
"properties": {
  "Duration": { "type": "integer", "format": "int64", "minimum": 0,
    "maximum": 864000000 },
  "Description": { "type": "string", "mininum": 50, "maximum": 3000 },
  "Size": { "type": "integer", "format": "int64", "minimum": 0,
    "maximum": 1099511627776 },
  "Qualities": { "type": "array", "items": { "type": "string",
    "enum": ["Lowest", "Low", "Medium", "SD", "HD", "TwoK", "FourK"],
    "minItems": 1, "maxItems": 7, "uniqueItems": true } }
},
"required": ["Duration", "Description", "Size", "Qualities"]
```

Fonte: Elaborado pelo autor (2023).

Por fim, a classe que agrupa todos os objetos anteriores, e o objeto raiz a ser trafegado, `Video`, com sua estrutura apresentada na Figura 4. A classe contém seu identificador único universal em conformidade com a RFC 4122[3] (`VideoId`), uma *string* representando o liame de acesso ao vídeo, também chamado de *Uniform Resource Locator* (URL), variando entre 80 e 100 caracteres, as informações sociais do vídeo (`SocialInfo`), de canal (`Channel`) e informações gerais do vídeo (`VideoInfo`):

Algumas informações sobre a complexidade e tipos de dados a serem trafegados em uma mensagem `Video` devem ser destacadas. O objeto possui nível de aninhamento 2, e conta com 5 propriedades strings, 3 objetos, 2 *arrays* e 6 inteiros, sendo 2 inteiros de 64 bits e 4 inteiros de 32 bits.

5.3 IMPLEMENTAÇÃO DOS PROTÓTIPOS

Os protótipos desenvolvidos neste trabalho de conclusão de curso visam representar sistemas distribuídos nas arquiteturas cliente-servidor e possivelmente produtor-consumidor, utilizando as bibliotecas de serialização selecionadas. Os protótipos têm

[3] `https://www.ietf.org/rfc/rfc4122.txt`

Figura 4 – Descrição da classe `Video` em formato *JSON Schema*

```
"properties": {
  "VideoId": { "type": "string", "format": "uuid" },
  "Url": { "type": "string", "minimum": 80, "maximum": 100 },
  "SocialInfo": { "$ref": "#/definitions/SocialInfo" },
  "Channel": { "$ref": "#/definitions/Channel" },
  "VideoInfo": { "$ref": "#/definitions/VideoInfo" }
},
"required": ["VideoId", "Url", "SocialInfo", "Channel", "VideoInfo"]
```

Fonte: Elaborado pelo autor (2023).

como base os cenários de teste definidos, e contam com mecanismo que permite o acoplamento de cada biblioteca de serialização separadamente. A estruturação do trabalho foi realizada considerando um ambiente em nuvem que vise simular o comportamento de uma aplicação distribuída na rede, ao mesmo tempo em que permite a realização de *microbenchmarks*. Para cada biblioteca, é gerado, se necessário, uma definição de esquema para a estrutura de dados proposta, seguida da implementação dentro do mecanismo acoplável. Desta forma, foram obtidas, de forma isolada, classes de implementação separadas, de modo a facilitar a compreensão e análise de forma qualitativa.

5.3.1 Ferramentas e Recursos Computacionais

Para a realização dos testes em *microbenchmark* e em cenário distribuído, foi utilizado um notebook para realização dos testes. Em cenário distribuído, tanto o cliente quanto o servidor estão no mesmo nó e sistema operacional. A escolha da ferramenta responsável por extrair as métricas de desempenho das estratégias avaliadas foi uma biblioteca de *benchmark* robusta e popular para a linguagem, e para extrações distribuídas optou-se por extrações em arquivos *.csv*, após instrumentação.

A Tabela 5 apresenta as configurações de hardware e software da máquina usada no trabalho:

Tabela 5 – Configuração de hardware e sistema operacional

Componente	**Especificação**
Modelo	Notebook Avell A62 LIV
Processador	Intel Core i7-10750H 2.6 GHz
Cores	6
Threads	12
Memória RAM	16 *gigabytes* (GB)
Disco Rígido	*Solid State Drive* 512 GB
Sistema Operacional	Microsoft Windows 10 Enterprise 10.0.19042

A partir das estratégias selecionadas, observou-se a serialização e desseriali-

zação por meio do monitoramento da execução dos respectivos métodos configurados para os cenários.

5.3.2 Configuração do Sistema

Linguagem de Programação. O projeto foi desenvolvido na linguagem C#. Durante a implementação, foi buscado seguir boas práticas de desenvolvimento, considerando as características da linguagem. Como todas as bibliotecas apresentaram suporte à linguagem C#, optou-se pelo desenvolvimento nesta linguagem por conta de sua ampla utilização na indústria. Assim, deve-se ter uma visão mais clara do desempenho das bibliotecas em boa parcela dos sistemas atuais. Para o projeto em questão, será utilizada a linguagem C# na versão 9.0, e arcabouço .NET 6.

Microbenchmark com BenchmarkDotNet. Para extração de métricas de desempenho em *microbenchmarks*, foi utilizada a biblioteca BenchmarkDotNet (DOTNET/BENCHMARKDOTNET, 2013), uma biblioteca de *Benchmark* específica para C#. BenchmarkDotNet (DOTNET/BENCHMARKDOTNET, 2013) é uma biblioteca popular para realizar *benchmarks* em C#. Ela fornece uma maneira simples e poderosa de definir, executar e analisar testes de desempenho de maneira precisa. A biblioteca oferece recursos avançados, como controle de iterações, configurações de aquecimento, estatísticas detalhadas e a capacidade de gerar relatórios completos dos resultados obtidos.

Teste de carga com Grafana k6. Para a condução de testes distribuídos de carga, foi utilizada a ferramenta Grafana k6[4], em conjunto com o BenchmarkDotNet e ferramentas suplementares desenvolvidas durante a implementação do projeto. O Grafana k6 é uma ferramenta de código aberto que oferece diversas funcionalidades para testes de desempenho em sistemas, por meio de testes de carga altamente configuráveis. Essa ferramenta incorpora os conceitos essenciais necessários para realizar uma ampla gama de testes de carga. Seu código é otimizado para consumir a menor quantidade possível de recursos, garantindo que não haja interferência nos resultados dos testes. É possível até mesmo construir cenários de avaliação de carga que serão executados durante o fluxo de *predeploy* de aplicações em ambientes de produção. Trata-se de uma ferramenta robusta e de fácil implementação, pois suas configurações de lançamento são armazenadas em simples arquivos no formato *.js*, permitindo o uso das capacidades da linguagem para a construção de cenários.

O fluxo de teste segue, em linhas gerais, quatro etapas principais: (i) Inicialização; (ii) Preparação; (iii) Execução do código em cada *Virtual User* (VU); (iv) Finalização (*teardown*). Paralelamente, é possível criar cenários e estágios de teste que modificam dinamicamente o comportamento dos testes durante a execução, permitindo a composição de *workloads* distintos, de forma paralela ou sequencial. O Grafana k6

[4] `https://k6.io/docs/`

oferece controles de tempo de execução, instanciação e controle simultâneo de VUs, possibilitando a realização de testes de pico, estresse e imersão. Além disso, a ferramenta implementa *outputs* que contêm os resultados das cargas realizadas, podendo ser salvos em formato JSON ou armazenados em bancos de dados *Time-Series*, por exemplo. Ainda há painéis e gráficos disponíveis, apresentando diversas métricas importantes para a análise do comportamento de um sistema em cenários de estresse e concorrência, implementados e mantidos com o apoio da comunidade de código aberto.

A Tabela 6 exibe as versões dos pacotes de cada biblioteca utilizados na implementação dos protótipos em C#.

Tabela 6 – Pacotes de bibliotecas usados no projeto, em C# 9

Biblioteca	**Pacote NuGet**	**Versão**
MessagePack	MessagePack	2.5.129
Apache Avro	Apache.Avro	1.11.3
Apache Avro	Confluent.Apache.Avro.AvroGen	1.7.7.5
Apache Thrift	ApacheThrift	0.19.0
Protocol Buffers	Google.Protobuf	3.24.4
FlatBuffers	Google.FlatBuffers	23.5.26
Cap'n Proto	Capnp.Net.Runtime	1.3.118
Newtonsoft.Json	Newtonsoft.Json	13.0.0.0
Http	System.Net.Http	6.0.0.0
BenchmarkDotNet	BenchmarkDotNet	0.13.19

Com as informações apresentadas acima, é possível ter clareza sobre a estrutura base e artefatos do sistema implementado.

Estrutura do Projeto. Em linhas gerais, o projeto é separado nas seções de Domínio (Entidades e Interfaces), Serviços (Regras de negócio e métodos de uso comum), Infraestrutura (Serializadores), Apresentação (Nós cliente/servidor e controladores HTTP) e *Benchmarks* (Coleções e configurações de teste). Ainda há uma camada de *CrossCutting* para compartilhamento de classes e projetos dentro da solução .NET. A estrutura do projeto está parcialmente orientada a *Domain-Driven Design* (DDD) (VERNON, 2013), porém em alguns casos optou-se pela maior praticidade em alguns cenários, visto que não é o foco do trabalho.

5.3.3 Serializadores

A construção de cada serializador respeitou as diretrizes de uma interface de nome *ISerializer*, que pode ser usada para serializar e desserializar qualquer objeto que é considerado um alvo de serialização, chamado *ISerializationTarget*. O Código 5.1 destaca os métodos principais métodos da interface, usados para serialização e desserialização no projeto. Todos os serializadores criados seguem uma estrutura em

que é possível serializar e desserializar um objeto, e obter como retorno o tamanho em bytes da serialização. Foi buscado disponibilizar assinaturas de métodos que utilizam tanto tipos genéricos quanto tipos estáticos, como é possível ver em seguida. A inspiração na construção das interfaces foi obtida parcialmente na construção de *benchmarks* de serialização do projeto de Johannes Deml[5], que fez um ótimo trabalho generalizando as estruturas e formatos de serialização. O retorno dos métodos é um inteiro de 64 bits representando seu tamanho final em bytes.

Código 5.1 – Interface base para serializador de ISerializationTarget

```
1 public interface ISerializer {
2     long BenchmarkSerialize<T>(T original) where T :
          ISerializationTarget;
3     long BenchmarkSerialize(Type type, ISerializationTarget original)
          ;
4     long BenchmarkDeserialize<T>(T original) where T :
          ISerializationTarget;
5     long BenchmarkDeserialize(Type type, ISerializationTarget
          original);
6     // ...
7 }
```

Para serialização e desserialização de objetos em cenário de concorrência, foram criados os métodos equivalentes, porém com mecanismo de exclusão mútua e *locks*. Os serializadores armazenam os resultados de serialização e desserialização em dicionários, portanto é possível que haja disputa e erros na tentativa de acesso ou escrita, quando duas ou mais *threads* estiverem sob execução. Este comportamento foi observado durante a execução dos testes. Ainda foram construídos mais métodos de apoio, que serão explicados posteriormente.

Código 5.2 – Interface base para serializador de ISerializationTarget em cenário concorrente

```
1 public interface ISerializer {
2     // ...
3     long BenchmarkSerializeThreadSafe<T>(T original) where T :
          ISerializationTarget;
4     long BenchmarkSerializeThreadSafe(Type type, ISerializationTarget
           original);
5     long BenchmarkDeserializeThreadSafe<T>(T original) where T :
          ISerializationTarget;
6     long BenchmarkDeserializeThreadSafe(Type type,
          ISerializationTarget original);
7 }
```

[5] https://github.com/JohannesDeml/SerializationBenchmarkDotNet

Já para os objetos a serem serializados, a interface segue o modelo apresentado no Código 5.3. Para cada objeto *ISerializationTarget*, é possível invocar a serialização ou desserialização de forma desacoplada, passando qualquer serializador disponível. Além disso, foram criados métodos de criação e recuperação de classes intermediárias, necessárias para a implementação de alguns serializadores. Com esta abordagem, foram violadas as regras de *Single-Responsability Principle* (SRP) e *Dependency Inversion Principle* (DIP) (MARTIN, 2009), porém isto permitiu o uso de tipos genéricos para estes serializadores, reduzindo o tempo de desenvolvimento.

Código 5.3 – Interface base para objeto serializável ISerializationTarget

```
1 public interface ISerializationTarget : IEquatable<
      ISerializationTarget>
2 {
3      Type GetType();
4      long Serialize(ISerializer serializer);
5      long Deserialize(ISerializer serializer);
6 }
```

No contexto deste trabalho, os alvos de serialização são as entidades `Video`, `VideoInfo`, `SocialInfo` e `Channel`.

Devido ao formato em que está construída a estrutura de serialização no sistema, a adição e uso de um novo serializador se torna simples, de modo que a implementação de um novo serializador exige somente a codificação na classe serializadora dentro de seu respectivo projeto e, caso seja necessário, no objeto a ser serializado. Detalhes de implementação são abstraídos, quando possível, tanto para o alvo de serialização quanto para quem irá utilizar o serializador. Desta forma, atingem-se parcialmente boas práticas de implementação baseando-se no princípio *SOLID*, usando conceitos de DDD e arquitetura em camadas, além de buscar seguir o princípio DIP, visando o desacoplamento e facilidade na troca do mecanismo de serialização.

Para este projeto, foram criados 8 serializadores, e sua implementação em detalhes será apresentada a seguir.

FlatBuffersSerializer. FlatBuffers baseia-se na compilação de um esquema em formato *.fbs* para geração de classes intermediárias *IFlatBufferObject* como forma de representação da informação. Foi necessário fazer o *download* do executável de compilação disponível em seu repositório oficial. A partir disso e da construção do esquema, a execução do seguinte comando já foi o suficiente para geração das classes intermediárias:

```
flatc.exe --csharp --gen-mutable video.fbs
```

Cada uma destas classes possui uma tabela `Table` para mapeamento de propriedades em cada objeto, e um *buffer* `ByteBuffer` para manipulação de dados, em seus

próprios métodos personalizados. Toda atribuição e consulta de dados baseia-se na manipulação de *offsets*, que são representações de um espaço contíguo na memória. Assim, sabe-se com precisão o tamanho de uma propriedade, ou o objeto inteiro. Primeiramente, é necessário declarar a manipulação de um objeto, atribuir valores a cada uma das propriedades, finalizar a inicialização e, por fim, realizar a escrita no *buffer*. Para objetos complexos, como `Video`, é necessário instanciar e atribuir cada objeto ou propriedade separadamente. Ao final, ocorre a escrita no método. Como `VideoInfo` possui uma a lista de enumerações, foi necessário realizar uma iteração e *casting* para o tipo *IFlatbufferObject*. A escrita de propriedades numéricas ocorre em formato *Little-Endian*, e a escrita de propriedades textuais ocorre por meio da criação de um vetor e inserção da *string* no espaço de memória. Em linhas gerais, a serialização usando FlatBuffers baseou-se na transformação do objeto a ser serializado em uma versão intermediária, a atribuição de valores a cada variável, por fim realizar a escrita no *buffer*. Já a desserialização ocorreu por meio da coleta do *ByteBuffer* e a posição do *offset* do objeto serializado, para por fim realizar a operação de o atribuir à tabela *Table* de forma simples. Somente com a manipulação de posições de memória, foi possível instanciar o objeto intermediário. Por fim, foi necessário converter o objeto intermediário ao manipulável pelo sistema, em uma chamada de construtor por método estático. Aqui, o uso de métodos estáticos pode trazer ganhos de desempenho, em especial para múltiplas serializações, por evitar acesso a dados instanciados. Apesar da grande quantidade de código gerado e necessidade de atribuição direta, não há muitas complexidades a partir do momento em que se estrutura uma boa maneira de realizar as operações.

ApacheAvroSerializer. Apache Avro pode fazer uso de um esquema gerado tanto previamente como durante o tempo de execução. Esta versatilidade faz com que haja pouca necessidade de código gerado em serializações, em especial usando a biblioteca AvroConvert[6], que abstrai a necessidade de esquema, e infere o esquema durante a serialização e desserialização de objetos. Quase não foi necessário compilar o esquema em objetos intermediários, porém com esta biblioteca houve problemas, por motivo não investigado, na serialização de `Video`, especificamente. Portanto, foi obrigatório o uso da serialização orientada a esquema, e com compilação de arquivo em formato *.avsc* e objeto intermediário *ISpecificRecord*. Seu esquema tem formato JSON, portanto é legível e compatível com diversos outros sistemas. Para compilação do esquema em objetos intermediários, foi necessário instalar a biblioteca *Confluent.Apache.Avro.AvroGen* e executar o comando a seguir:

```
avrogen -s avroObjects.avsc
```

Para compilação do esquema. Apache Avro não conseguiu realizar a serializa-

[6] `https://github.com/AdrianStrugala/AvroConvert`

ção do objeto `VideoInfo` sem interferência, pois foi observado que enumerações com valores definidos manualmente não são interpretados pelo desserializador. Apache Avro obtém o valor em ordem natural crescente do enumerador, de acordo com a sua posição no momento de criação do esquema. Sendo assim, foi necessário converter o enumerador a partir de sua posição em um *array* contendo os valores de enumeração possíveis. Além da compilação do esquema, a serialização precisou também gerar objeto intermediário para serialização. Utilizou-se como referência na serialização e desserialização o código presente no repositório Avro, em testes de *benchmark*[7]. Utiliza-se um fluxo de escrita na memória com *SpecificWriter* utilizando o esquema presente no objeto intermediário, e posteriormente ocorre a conversão para um *array* de bytes. Pode-se suspeitar que esta transformação de dados apresente impacto nos tempos finais de serialização. A desserialização ocorre de forma simples, utilizando *SpecificReader* e *BinaryDecoder*, com o apoio dos esquemas de entrada e saída. Por fim, é necessário realizar a conversão para objeto genérico manipulável pelo sistema, caso opte-se pela recuperação completa do objeto.

CapnProtoSerializer. Cap'n Proto realiza o uso de arquivo de compilação em formato *.capnp*, e consegue, com o apoio de biblioteca específica para esta função[8], compilar o esquema em tempo de *build*, o que é bastante interessante. Seu objeto intermediário implementa a interface *ICapnpSerializable*, e sua serialização baseia-se na criação de um *MessageBuilder* e uso de *Readers* e *Writers* presentes nas próprias classes compiladas. Faz o uso de ponteiros para escrita de valores em string, e manipulação de bits usando a classe *BitConverter* na escrita de valores do tipo C# *double*, convertendo-os para inteiros de 64 bits. A escrita de uma lista de comentários, como em `SocialInfo`, utiliza ponteiros como referência deste tipo em forma de grafo de objetos. Sua desserialização inicia-se por meio do estado de desserialização, ou *DeserializerState* de cada classe, e tem a estrutura geral do objeto serializado, assim como métodos para acesso aos dados. O mesmo ocorre na serialização, em que *SerializerState* expõe todas as funcionalidades para codificação do objeto. Na manipulação de lista de de enumerações, trata-o como uma lista de leituras e realiza internamente um *cast* para representá-lo como lista de enumerações. A serialização de `VideoInfo.Qualities` precisa de um segundo serializador, que realiza a construção do ponteiro e link no estado de serialização. Para `Video`, é necessário instanciar um objeto intermediário para cada tipo complexo, e serializá-lo como um objeto comum. Sua codificação para tipos complexos baseia-se na construção e associação de ponteiros no estado de serialização. A desserialização de tipos complexos baseia-se na leitura e decodificação do ponteiro para uma estrutura.

ApacheThriftSerializer. Apache Thrift faz o uso de objetos intermediários

[7] `https://github.com/apache/avro/blob/3e682e999980f718328aa8dc20e7c7052c9280f4/lang/csharp/src/apache/benchmark/Benchmarks.cs`

[8] `https://www.nuget.org/packages/CapnpC.CSharp.MsBuild.Generation`

TBase, compilados a partir de esquema em arquivo *.thrift*. Foi necessário realizar o download de seu executável para Windows, e executar o seguinte comando, dado o contexto deste trabalho:

```
.\thrift-0.19.0.exe --gen netstd thriftObjects.thrift
```

Sua serialização baseia-se no uso de fluxos de transporte e o protocolo de serialização *TCompactProtocol*. Utilizam-se métodos assíncronos na escrita e limpeza do *buffer* de memória, porém para este cenário, não foi tirado proveito do uso de métodos assíncronos. Por fim, a *MemoryStream* é convertida em um *array* binário e retornada como resultado da serialização. A desserialização necessita da ativação de uma instância do tipo a ser serializado, e *cast* para *TBase*. Isto pode não ser tão eficiente para um serializador, apesar de trazer versatilidade na construção do código. Sua desserialização usa os mesmos fluxos de transporte e protocolo de compactação, assim como métodos assíncronos que foram executados de forma síncrona neste experimento. Apache Thrift disponibiliza muitos tipos de transporte e protocolos, trazendo versatilidade no formato de comunicação por meio de uma camada de abstração que permite o tráfego de qualquer informação por qualquer meio de de transporte (como *TBufferedTransport*, *TFramedTransport*, *TStreamTransport*, entre outros) e mecanismo de comunicação (*TCompactProtocol*, *TBinaryProtocol*, *TJsonProtocol*, entre outros). Apesar da grande variedade de implementações possíveis, foi realizada a implementação da serialização para representação em memória, disponibilizando somente a lista de bytes como etapa final, e não foram utilizadas as capacidades de transporte em comunicação distribuída que a biblioteca oferece.

NewtonsoftJsonSerializer. Newtonsoft.Json possui uma construção de código muito simples e não requer esquema na serialização ou desserialização. Sua serialização, por padrão, retorna o objeto em formato de *string* em JSON, porém foi transformada em um *array* de bytes UTF-8 para compatibilidade com os mecanismos construídos, além fazer possível ter-se uma base de comparação. Sua desserialização transforma novamente o *array* de bytes em uma *string* JSON, e depois realiza a conversão para um objeto de tipo definido ou não. Trata-se de uma implementação muito rápida, adaptável e confiável, não necessitando de objetos intermediários ou esquemas.

MessagePackCsharpSerializer. MessagePack possui implementação de apenas uma linha de código, para serialização ou desserialização, de forma adaptável a tipos e esquema, definidos ou não por meio de simples anotações na definição de classe (*MessagePackObject*) e de posição na definição das propriedades (`[Key(0)]`, entre outros), de forma não obrigatória. É o serializador mais versátil e simples apresentado, sem necessidade de compilação, manipulação de objetos intermediários ou esquemas. Não foi possível compreender com clareza a implementação deste formato

de serialização, no entanto, são usados métodos compilados e duas *structs* principais: *MessagePackWriter* e *MessagePackReader*. Aqui, o uso de *structs* chama atenção pois, ao invés de realizar a passagem por referência, faz a passagem diretamente por valor. Imagina-se portanto, que seja necessário somente realizar operações internas, sem acesso à estruturas de dados mais lentas, como a memória RAM, em relação à pilha. Observou-se o uso escrita em *buffer*, e a codificação de inteiros ocorre em BE. De forma interessante, a serialização de inteiros verifica o tamanho do valor e, dependendo do valor, tenta serializar em 2 bytes em *Big-Endian*, alocados separadamente ou em conjunto, ou até 1 byte somente. Outras checagens ocorrem para se obter o mesmo comportamento dinâmico e eficiente na serialização, considerando se possui sinal ou não. A codificação de strings ocorre por meio de uma sequência ou *span* de leitura de bytes em UTF-8, e é acompanhada de um cabeçalho, responsável pela alocação do espaço de memória. A implementação da desserialização é relativamente curta e baseia-se majoritariamente na leitura em BE dos bytes contidos no leitor *SequenceReader*, que faz parte do sistema de *buffers* de C#.

ProtobufSerializer. A implementação de Protocol Buffers baseou-se no download do executável de compilação[9], construção de esquema em arquivo com formato *.proto*, e compilação por meio do seguinte comando:

```
protoc.exe -I=.\ --csharp_out=.\ .\protoObjects.proto
```

A execução da compilação resultou na criação dos objetos intermediários que implementam a interface *IMessage*. Nas classes principais do domínio, foi necessário fazer anotações *[ProtoContract]* na definição de classe, além da ordem de propriedades definidas na mesma (*[ProtoMember(1)]*, por exemplo).

A partir disso, a serialização baseou-se somente na instanciação da classe intermediária, cálculo do tamanho do *buffer* necessário para armazenar a mensagem, alocação do espaço e, por fim a escrita no *buffer*. O cálculo do tamanho do *buffer* e a escrita no mesmo estão disponíveis como métodos de cada classe intermediária *IMessage*, fazendo assim com que detalhes de serialização fossem abstraídos dentro de suas próprias implementações. A desserialização necessita somente do reconhecimento do tipo do objeto a ser serializado e, a partir de um *array* de bytes, realizar a desserialização em único comando por meio de seu *Parser* disponível em cada classe intermediária, que faz o reconhecimento dos contratos e anotações definidas anteriormente para uma desserialização rápida. A escrita serializada ocorre com a criação de *tags* representando o tamanho em bytes de cada propriedade, seguida da escrita de cada propriedade. A classe intermediária possui seu próprio contexto em *struct WriteContext* e cada propriedade possui seu contexto, também em *struct*, como *WriterInternalState*. Isto possibilita o reconhecimento de posição e realização

[9] `https://github.com/protocolbuffers/protobuf/releases`

de operações no *buffer*. Protocol Buffers interpreta codificações em *Little-Endian* e *Big-Endian* por meio da checagem no código. Estão disponíveis métodos de escrita de inteiros de tamanho fixo e variável, de 32 e 64 bits. Para inteiros de tamanho variável, tenta realizada menor alocação em bytes possível, dado o valor da propriedade. A escrita de strings reconhece o tamanho em bytes do texto, e faz a escrita no *buffer*. Se o comprimento da *string* é o mesmo da contagem de bytes, realiza a escrita em codificação ASCII. Se não, escreve em UTF-8. Para lista de strings, é utilizada como apoio a classe *FieldCodec<T>*, que realiza operações eficientes de cálculo e alocação de todo o espaço em memória em uma única operação, para em seguida realizar a escrita incremental, para os casos em que o tipo de fio do campo tem identificador 2 (veja a Tabela 1) , ou seja, é delimitado por comprimento. Como a lista de strings é deste tipo, ocorre a alocação e escrita eficiente. Para a lista de enumerações contida em *VideoInfo.Qualities*, ocorre a mesma operação. Para a escrita de tipos complexos e outros objetos derivados de *IMessage*, é chamado o próprio método interno de escrita do objeto intermediário. A leitura e desserialização ocorre por meio da atribuição das propriedades do objeto ao objeto principal e, para o caso de tipos complexos, é feita chamada interna que realiza as mesmas atribuições. Além disso, Protocol Buffers também consegue realizar a escrita e leitura de tipos não mapeados.

BinaryFormatterSerializer. O serializador depreciado *BinaryFormatter* faz uso de fluxos de memória para serialização e desserialização. Para esta implementação, foi realizada a conversão para *array* binário, por motivos de compatibilidade com a estrutura do sistema. Sua implementação não necessita de esquema compilável, ou classes intermediárias, apenas a sinalização via anotação em classe informando ao sistema que é uma classe serializável (*[Serializable]*). Na desserialização, no entanto, precisa de uma instância prévia da classe a ser decodificado, para injeção de dados no objeto. Seu código é simples e fácil de ser implementado. Devido ao fato deste serializador estar depreciado por graves motivos de segurança [10], não foi feita uma análise mais a fundo dos mecanismos de serialização e desserialização desta classe.

5.3.4 Gerador de Objetos/Mensagens

Para garantir a aleatoriedade dos dados gerados e evitar o enviesamento da análise de desempenho, foi utilizada a biblioteca AutoBogus[11], construída em C#, que tem o objetivo de prover a instanciação e preenchimento de objetos com valores semi-aleatórios para cada propriedade do mesmo, de forma parametrizável.

A tabela 7 descreve as propriedades e intervalo aproximado do tamanho de cada objeto de serialização, de modo a imaginar o volume de tráfego de dados a ser

[10] `https://learn.microsoft.com/en-gb/dotnet/standard/serialization/binaryformatter-security-guide`

[11] `https://github.com/nickdodd79/AutoBogus`

serializado.

Tabela 7 – Classes, propriedades e tamanho estimado de cada objeto de serialização

Classe	Propriedades	Tamanho médio
`Video`	`VideoId`, `Url`, `VideoInfo`, `SocialInfo`, `Channel`	2600 B
`VideoInfo`	`Description`, `Duration`, `Size`, `Qualities`	1350 B
`SocialInfo`	`Dislikes`, `Likes`, `Views`, `Comments`	590 B
`Channel`	`Name`, `ChannelId`, `Subscribers`	310B

Durante os testes de desempenho, a criação de um novo objeto é realizada usando as funções de construtor fornecidas pela biblioteca AutoBogus. Em alguns casos, foi criado um método que gera objetos de tipos aleatórios, para atender a necessidades específicas nas instrumentações das avaliações.

5.3.5 Comunicação Cliente/Servidor

Para possibilitar a comunicação em REST usando diferentes formatos de serialização, foi necessário implementar, para o envio de requisição HTTP, um método que analisa o tipo de objeto a ser trafegado para construção da requisição, e a dispara com o conteúdo mapeado pelo Serializador. No nó receptor, foi criado um *endpoint* que recebe qualquer objeto de serialização de determinado serializador.

Nó Servidor (*Serialization.Receiver*): É responsável por absorver o tráfego de requisições de serialização e desserialização em HTTP, para análise e extração de dados de vazão e tempo de resposta. A cada recebimento de requisição, ocorre o incremento de um contador que realiza a checagem de requisições atendidas por segundo. As rotas disponíveis estão expostas a seguir:

```
POST receiver/serializer
POST receiver/serializer/serialize/channel
POST receiver/serializer/serialize/socialinfo
POST receiver/serializer/serialize/videoinfo
POST receiver/serializer/serialize/video
POST receiver/serializer/deserialize
POST receiver/serializer/monitoring/start
POST receiver/serializer/monitoring/save-results
```

Para cada uma destas rotas, é disparada a desserialização ou serialização e envio da resposta ao requisitante. A primeira rota realiza a desserialização e serialização no mesmo método. Todos os métodos lidam com *arrays* de bytes, além de parâmetros na consulta como forma de apoio na serialização ou desserialização. O método *Serialize* aceita o objeto a ser serializado método oferece HTTP é responsável

por extrair e salvar as medições até o tempo presente, reinicia o contador e limpa as medições anteriores.

Nó cliente e gerador de carga ao servidor (*Serialization.Sender*): É responsável pelo disparo das requisições HTTP ao nodo receptor, gerando carga no lado servidor, durante a medição dos testes. As rotas disponíveis estão expostas a seguir:

```
POST sender/workload
POST sender/clear
```

A primeira rota é usada para disparar uma rotina contínua e concorrente de múltiplas requisições HTTP ao nó servidor, enquanto a segunda rota interrompe a geração contínua de carga. Estas rotas servem para simular o volume de tráfego local ou por meio de comunicação REST, que executam simultaneamente com os testes, explicados na sequência. Apesar deste nó ter sido projetado e construído para geração de carga distribuída, optou-se pelo uso do programa Grafana k6[12] para esta atividade, devido à facilidade na configuração e extração de resultados, além de possuir uma interface amigável.

Geração de carga de comunicação distribuída com Grafana k6. Conforme mencionado acima, foi utilizado a ferramenta Grafana k6, que disponibiliza a criação de cenários de carga personalizados. Os cenários de carga definidos para este experimento seguem a primitiva de aumento gradual nas instâncias de usuários virtuais, e estabilidade na quantidade de usuários simultâneos por certo período de tempo. Devido a complexidades não solucionadas e a não confiabilidade no comportamento do gerador, optou-se por não utilizar o nó *Serialization.Sender* e seu respectivo serviço de geração de carga *Workload* nas medições finais, servindo apenas como experiência e aprendizado.

5.3.5.1 Estrutura de testes

Para a realização deste trabalho, foi feita a divisão dos *benchmarks* em quatro agrupamentos principais: (i) Serialização Individual; (ii) Serialização Serial; (iii) Serialização Concorrente; (iv) Serialização em REST. Estes quatro agrupamentos formarão os tópicos de avaliação, a serem vistos posteriormente neste trabalho. A biblioteca BenchmarkDotNet, para cada conjunto de testes, segue um fluxo predefinido de etapas, seguido de um fluxo configurável. A construção dos casos de *benchmark* ocorre a partir da multiplicação da quantidade de todos os parâmetros configurados, dado que para um cenário com 2 propriedades, 2 itens em cada propriedade e 2 métodos a serem executados, haverá 8 casos no total. Para cada Benchmark, pode-se ter, em ordem de agrupamento decrescente, Lançamentos (*launches*), Iterações (*iterations*) e

[12] `https://k6.io/docs/`

Invocações (*invocations*), a forma mais atômica disponível. A biblioteca não disponibiliza as medições para invocações, somente informações simples no sumário. Sendo assim, cada iteração invoca x vezes o método observado, e o lançamento itera y vezes sobre o conjunto de invocações. Por fim, são executados z lançamentos, e os relatórios e medições são extraídos.

A estrutura de cada teste utilizando *BenchmarkDotNet* utiliza alguns dos seguintes parâmetros:

- ***Serializer*:** serializador a ser instanciado;
- ***Target*:** objeto a ser serializado;
- ***NumMessages*:** quantidade de mensagens (para serialização serial e concorrente);
- ***NumThreads*:** número de threads à serem executadas (para cenários com concorrência);

Quando um Benchmark é executado, todas as combinações definidas por meio dos parâmetros são executadas e exportadas para arquivos de medição e sumário. Para *Serializer*, é iniciada uma lista de Serializadores que será executada em todos os métodos. Portanto, todos os métodos serão executados usando cada um dos serializadores instanciados. O mesmo ocorre para *Target*, em que todos os métodos usarão todos os *Targets* instanciados. A biblioteca oferece relatórios sumarizados em arquivos *.html*, *.csv*, *.md*, entre outros, além de disponibilizar gráficos predefinidos na linguagem R. Os sumários disponibilizam customizações por meio de colunas, a partir do acesso aos dados dos casos de *benchmark*, e colunas predefinidas, como os tempos mínimo e máximo das medições. Para este trabalho, foi adicionado um diagnosticador de memória, para avaliar o consumo durante as operações.

Antes de cada iteração, se o caso de *benchmark* é de serialização, observa-se o tipo do objeto e verifica-se se o mesmo precisa da existência de uma classe intermediária. Foi necessário gerar objetos antes das medições para Apache Avro, Apache Thrift, Cap'n Proto e Protocol Buffers. Desta forma, a geração de cada objeto intermediário não interfere nas medições em nenhum dos cenários avaliados, exceto na análise de vazão pelo servidor. Já a preparação da desserialização envolve a execução da etapa anterior, seguida da serialização prévia do objeto. Desta forma, isola-se o código a ser instrumentado, dos códigos de preparação, sem interferir na medição final. Ao final de cada lançamento, todas as serializações e desserializações são limpas da memória.

Serialização Individual. A serialização individual contou com a adição de uma nova coluna que aponta o tamanho final médio da serialização, usado posteriormente na análise de taxa de compressão. Foram executadas 60 iterações de 1 invocação para cada caso de *benchmark*, e todas as iterações contaram com a geração de carga randomizada. Desta forma, pôde-se atingir uma normalidade estatística relativamente

segura quanto ao tamanho dos objetos após a serialização. Este tipo de serialização contém somente os casos de teste *Serialize()* e *Deserialize()*. A execução dos testes foi rápida, não demorando mais do que 2 minutos para que se pudesse ter os resultados em medições ou sumário.

Serialização Em Série. A serialização em série conta com a geração de múltiplos objetos, serializados ou desserializados para análise. Como medida de melhoria no desempenho, em cenários em que é preciso instanciar múltiplos objetos, foi percebido que a geração automática de múltiplos itens, todos com propriedades aleatórias, estava demorando mais do que os próprios testes em si. Para contornar a situação, foi definido que cada lista de objetos teria um tamanho múltiplo de 64, em que seriam gerados lotes de objetos de 64 itens aleatoriamente, repetidamente adicionados à lista principal de objetos até atingir a quantidade desejada para o teste. Com esta medida, espera-se obter um equilíbrio entre desempenho e aleatoriedade na geração de objetos. Aqui também foi necessário preparar cada iteração de caso de *benchmark* com a instanciação prévia dos objetos, geração de tipos intermediários quando necessário, e serialização prévia em caso de desserialização. Foram utilizadas as variações de carga de objetos, já apresentadas na definição de métricas: 128.000, 256.000, 384.000, 512.000, 640.000, 768.000, 896.000, 1.024.000, 1.152.000 e 1.280.000 mensagens. A execução deste conjunto de *benchmarks* teve uma demora considerável, dadas as preparações prévias, número de serializadores e quantidade de mensagens. Dadas 30 serializações para cada *benchmark*, foi necessário executar o processo em duas etapas, cada um com uma parte dos parâmetros avaliados, para que se pudesse haver o completamento das atividades em tempo adequado.

Serialização Concorrente. Para possibilitar a serialização concorrente, foi essencial realizar a instanciação, a preparação e a serialização como etapas preliminares à medição. Adicionalmente, foram conduzidas 30 iterações a fim de alcançar maior normalização de dados e minimizar qualquer viés decorrente de possíveis gerações aleatórias acima ou abaixo da média.

Serialização em REST. Durante o trabalho, observou-se que a biblioteca não oferece a instrumentação de execução de testes por tempo fixo, nem a coleta dos dados retornados por cada método durante os *benchmarks*, retornando somente o *makespan* de execução do método. Como a medição de cenários distribuídos envolvem o tempo e latência, foi necessário fazer a instrumentação e controle das observações em um componente codificado à parte, para que fosse possível coletar as informações e agrupá-las corretamente. A execução dos testes precisou estar alinhada em relação ao tempo com o gerador de carga, para cada cenário e número de clientes simultâneos. Desta forma, enquanto as cargas concorrentes eram executadas, os *benchmarks* foram coletados com *thinking time* de 500ms. Para que as medições estivessem alinhadas com a carga de requisições durante o teste, foi implementado um mecanismo que só

permitia a execução do próximo teste quando uma tecla fosse pressionada. Assim, foi possível extrair as medições para cada cenário proposto, com controle temporal. Tanto pata as coletas no servidor quanto para as coletas no cliente, foram os mesmos dois cenários de testes de estresse com cargas sequenciais. Separadamente, são executadas cargas de trabalho para serialização, seguida da desserialização. Cada cenário segue o seguinte plano de carga:

1. 200 VUs com *ramp-up* de 5s e duração de 60s;
2. 500 VUs com *ramp-up* de 5s e duração de 60s;
3. 1000 VUs com *ramp-up* de 5s e duração de 60s;
4. 2000 VUs com *ramp-up* de 10s e duração de 60s;
5. 3000 VUs com *ramp-up* de 15s e duração de 60s;
6. 3500 VUs com *ramp-up* de 15s e duração de 60s;
7. 500 VUs com *ramp-down* de 10s;
8. 0 VU com *ramp-down* de 10s;

Cada cenário tem duração de 7m15s. Esta configuração foi escolhida de forma empírica após experimentação dos cenários, de modo a representar a evolução do uso de recursos do sistema em relação à carga. Visa-se observar o sistema altas cargas e tempo médio de duração, com aumento rápido, porém não repentino, na demanda de operações. Os cenários são executados sequencialmente para que seja possível observar a vazão em cada demanda, de forma isolada. Para a geração das requisições ao servidor, foram criadas e salvas em arquivo JSON 64 mensagens distintas e aleatórias, dentro dos 4 objetos selecionados para serialização e desserialização. Cada mensagem contém uma *string* em *base64* que representa seu *array* binário, ou o objeto representado em JSON, para o caso da serialização. Cada VU utiliza uma das 64 mensagens disponíveis de forma randomizada a cada iteração, garantindo assim maior diversidade de dados trafegando pelo sistema durante a avaliação.

5.4 AVALIAÇÃO QUANTITATIVA

Nesta seção, os testes de desempenho nos cenários definidos serão executados, para medir o tempo de processamento, a vazão e latência na comunicação, compactação de dados e demais métricas em cada biblioteca de serialização, utilizando os protótipos desenvolvidos.

Em cada execução, são definidas as mesmas cargas e cenários, e modo a trazer a maior confiabilidade possível para os resultados. Os dados coletados durante os testes de desempenho e as características das implementações das bibliotecas de serialização, levando em consideração as métricas quantitativas e qualitativas estabelecidas, estão expostos em gráficos, quando possível, contendo considerações técnicas

relevantes para cada métrica avaliada. As métricas qualitativas estão descritas em formato de tópicos, apontando destaques positivos, negativos e uma conclusão geral. Ao final, são tecidas considerações gerais sobre cada biblioteca e seu desempenho de acordo com o contexto, em que cada implementação é comparada com as outras em vista do cenário avaliado. A apresentação de resultados em *microbenchmark* é produto da média de várias iterações, enquanto os resultados em cennário distribuído consideram múltiplas medições em um intervalo de tempo.

5.4.1 Serialização Individual

Ao serializar somente 1 objeto, visa-se obter informações quanto à velocidade e taxa de compressão de cada serializador, em cenário que exige pouco poder computacional. Foram executadas 60 iterações para obtennção de resultados, e suas médias estão expostas abaixo.

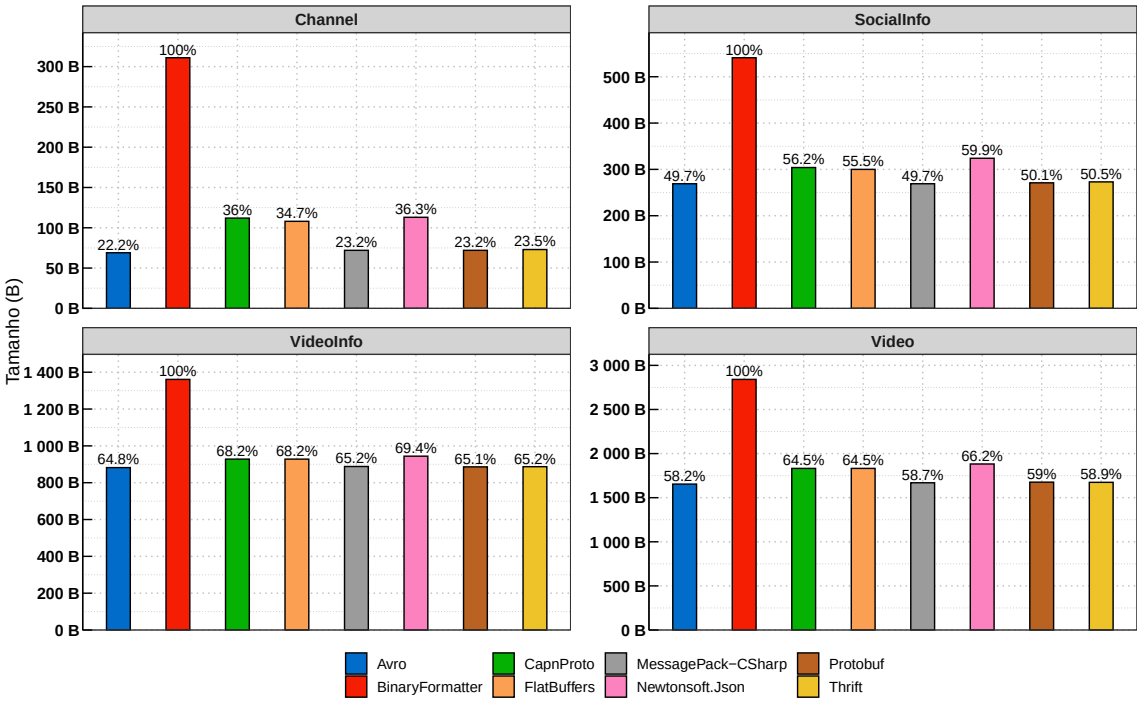
A Figura 5 ilustra o tamanho final da mensagem em bytes para cada objeto de serialização. Acima de cada coluna, está contida a informação do tamanho relativo do objeto, considerando como base o tamanho original do objeto, presente em *BinaryFormatter*. Aqui, é necessário destacar que o formato de serialização *BinaryFormatter* foi descontinuado devido a graves problemas de segurança[13], sendo usado neste trabalho somente para medir o tamanho do objeto e ser usado como base de comparação. Continuando, deve-se considerar que, quanto menor o percentual em relação ao tamanho original do objeto, mais eficiente é a serialização na dimensão de espaço. Para o objeto `Channel`, observa-se uma grande diferença no percentual de compressão em relação ao objeto original, para todos os serializadores observados. Isto ocorre pelos mecanismos de compressão de inteiros disponíveis, possivelmente sendo usados neste caso. Apache Avro, MessagePack, Protobuf e Apache Thrift apresentaram os menores tamanhos relativos, seguidos por Flatbuffers, Cap'n Proto e Newtonsoft.Json. Para os outros objetos, ainda se obteve compressão mais eficiente, porém em percentual menor e sem muitas discrepâncias. A compressão do objeto `SocialInfo` apresentou um intervalo percentual de aproximadamente 50% a 55%, em média, para todos os serializadores. `VideoInfo` teve um intervalo de taxa de compressão entre 65% a 70%. Ao considerar o objeto *Video*, a taxa de compressão passa a ter um intervalo de, em média, 60% a 65%.

No que diz respeito ao *Makespan*, a Figura 6 exibe o tempo médio de cada serializador para a serialização e desserialização de um único objeto, categorizado por método. Mais uma vez, *BinaryFormatter* servirá apenas como base de comparação para esta avaliação.

Para o objeto `Channel`, que contém duas pequenas *strings* e uma propriedade

[13] `https://learn.microsoft.com/pt-br/dotnet/standard/serialization/binaryformatter-security-guide`

Figura 5 – Tamanho final da mensagem



Fonte: Elaborado pelo autor (2023).

inteira de 32 bits, os serializadores FlatBuffers, MessagePack e Protobuf demonstraram eficiência na serialização em comparação com os demais. Ao contrário, o Cap'n Proto foi até mais lento do que a serialização textual presente em Newtonsoft.Json. Quanto à desserialização, Cap'n Proto, Flatbuffers e Protobuf foram os mais eficientes para este objeto, conseguindo o desserializar em menos de 10 microssegundos.

Para o `SocialInfo`, um objeto contendo uma pequena lista de *strings* de 5 a 300 caracteres, além de 3 inteiros de 32 bits, o Apache Avro teve um desempenho consideravelmente melhor na serialização em relação às demais observações. FlatBuffers, MessagePack e Protobuf também apresentaram rapidez na serialização, enquanto na desserialização, Cap'n Proto, FlatBuffers e MessagePack não tiveram dificuldades para desserializar com rapidez. O Apache Avro, ao desserializar o objeto, demorou aproximadamente três vezes mais tempo do que na serialização, sendo um resultado bastante negativo.

No que diz respeito ao objeto `VideoInfo`, que contém uma *string* de 50 a 1000 caracteres, um *array* de até 5 *enums* e 2 inteiros de 64 bits, FlatBuffers e Protobuf tiveram os melhores resultados tanto na serialização quanto na desserialização, seguidos por MessagePack. O Apache Thrift possui serialização eficiente, mas sua desserialização não é tão rápida quanto às outras bibliotecas. Cap'n Proto teve o mesmo comportamento de desserializar rapidamente e serializar lentamente, enquanto o Apache Avro teve desempenho semelhante ao Newtonsoft.Json.

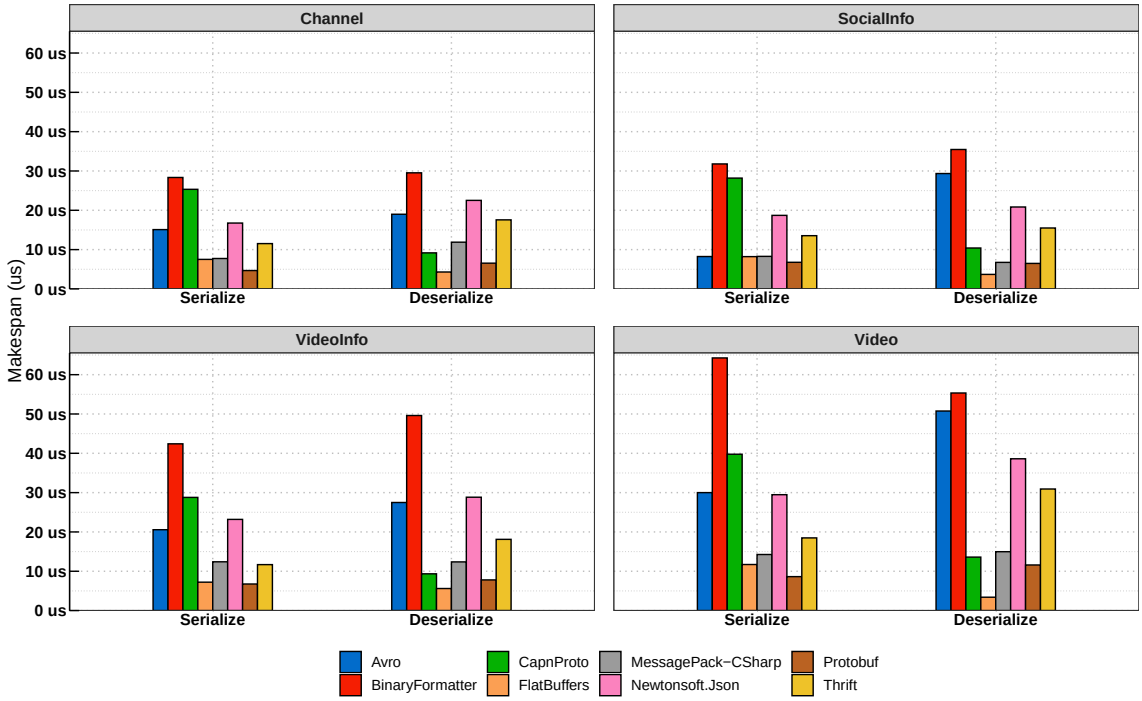
Para o objeto `Video`, que contém todos os anteriores, além de um UUID representado em *string* e uma URL de 80 a 100 caracteres, observa-se ineficiência na serialização por parte do Apache Avro, sendo tão ou mais lento que a serialização textual. Na desserialização, a ineficiência é acentuada. O Cap'n Proto apresentou a serialização mais lenta dentre as observadas, enquanto o Apache Thrift apresentou boa serialização e uma desserialização aproximadamente 50% mais lenta. MessagePack, FlatBuffers e Protobuf tiveram as melhores serializações e desserializações, com grande destaque para o FlatBuffers, que teve um *makespan* equivalente à da serialização de `Channel`.

Como conclusão, é possível elencar alguns pontos em relação a cada serializador:

1. Cap'n Proto apresentou maior eficiência na desserialização em relação à serialização em todos os cenários avaliados. O comportamento de desserialização ocorre devido ao seu mecanismo eficiente de *zero-copy*.
2. FlatBuffers teve desempenho constante e apresentou alguns dos menores *makespans* observados, tanto na serialização quanto desserialização. Na desserialização, também faz proveito do mecanismo de *zero-copy*.
3. O tamanho do objeto não afetou o *makespan* de serialização e desserialização de Protobuf, tornando-o um dos serializadores mais constantes nos testes observados;
4. MessagePack apresentou bom desempenho quanto aos tempos de serialização e desserialização, considerando qualquer objeto;
5. Apache Avro apresentou desempenhos negativos em relação aos concorrentes, sendo tão ou mais lento do que a serialização via Newtonsoft.Json;
6. Apache Thrift apresenta sempre uma desserialização mais lenta do que serialização, e não teve destaques em qualquer uma das observações. À medida em que o tamanho e complexidade do objeto aumentam seus tempos também aumentam.

A Figura 7 apresenta a quantidade de memória RAM alocada em cada serializador, obtidas por meio de mecanismo disponibilizado pela própria biblioteca BenchmarkDotNet, distinguidos por método e objeto de serialização. Na serialização, à medida em que o objeto aumenta de tamanho e complexidade, percebe-se o aumento do consumo de memória por Apache Avro, Cap'n Proto e Newtonsoft.Json, com destaque negativo para Cap'n Proto no objeto `Video`. Já para MessagePack, o consumo de memória é constante e baixo com grandes ou pequenos objetos, sendo quase zero para serializar o objeto `Channel`. Para Protobuf, ocorre o mesmo comportamento, porém desta vez o menor consumo apresentado reside na serialização de `SocialInfo`. FlatBuffers também apresenta baixo uso de memória, exceto para serializar `Video` e `Channel`, apesar

Figura 6 – *Makespan* observado com uma única serialização, distinguidos por serializador e objeto de serialização



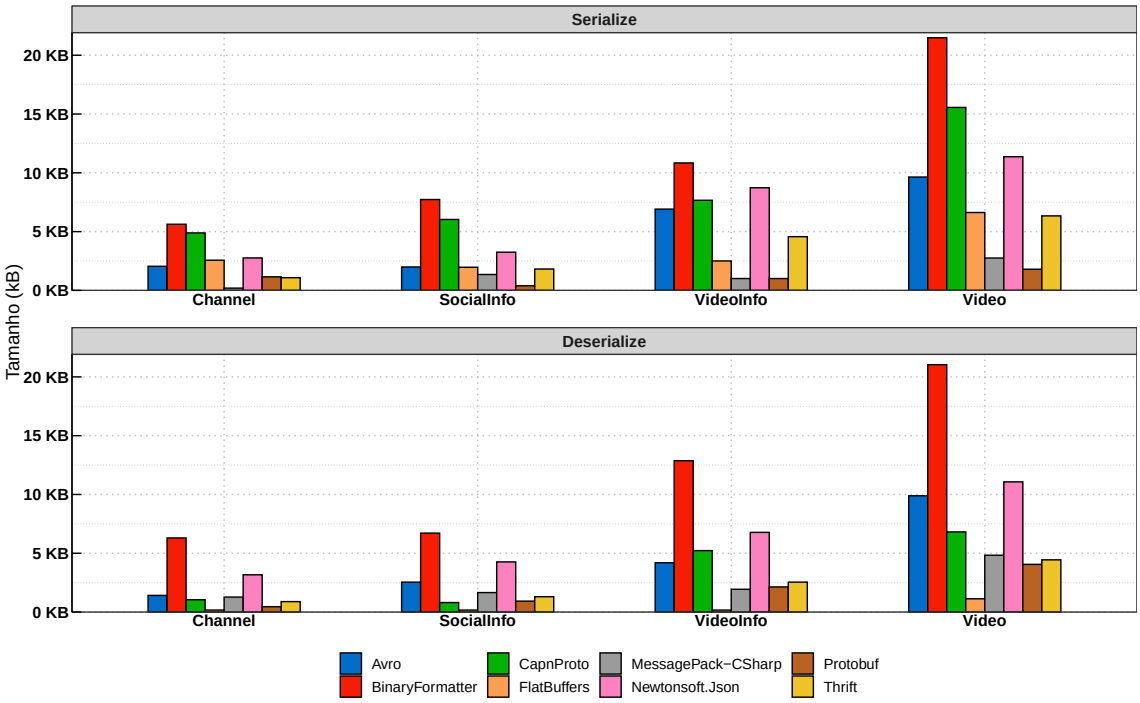
Fonte: Elaborado pelo autor (2023).

de se manter na média em comparação aos melhores concorrentes. Apache Thrift apresenta consumo proporcional ao tamanho do objeto.

Observando a desserialização, o consumo de memória é quase zero para FlatBuffers, em todas as observações, sendo o líder deste cenário. Cap'n Proto tem baixo consumo para `Channel` e `SocialInfo`, porém, o objeto *VideoInfo* o faz consumir mais memória. O mesmo acontece para `Video`, que abarca `VideoInfo`. Protobuf, MessagePack e Apache Thrift têm baixo consumo, de forma proporcional à complexidade e tamanho do objeto. Apache Avro e Newtonsoft.Json têm consumos em proporções parecida, com Apache Avro sendo relativamente melhor do que a serialização textual, e pior do que os outros concorrentes. O único cenário vantajoso observando as medições ocorre em `VideoInfo`, em que consome um pouco menos de memória RAM do que Cap'n Proto.

Conclui-se que, no viés de consumo de memória para uma única serialização, Protobuf e MessagePack consomem menos memória na serialização, enquanto que FlatBuffers é o mais leve na desserialização para qualquer tipo de objeto, junto de Protobuf, MessagePack e Apache Thrift, a depender da complexidade.

Figura 7 – Memória RAM alocada em cada serializador, distinguidos por método



Fonte: Elaborado pelo autor (2023).

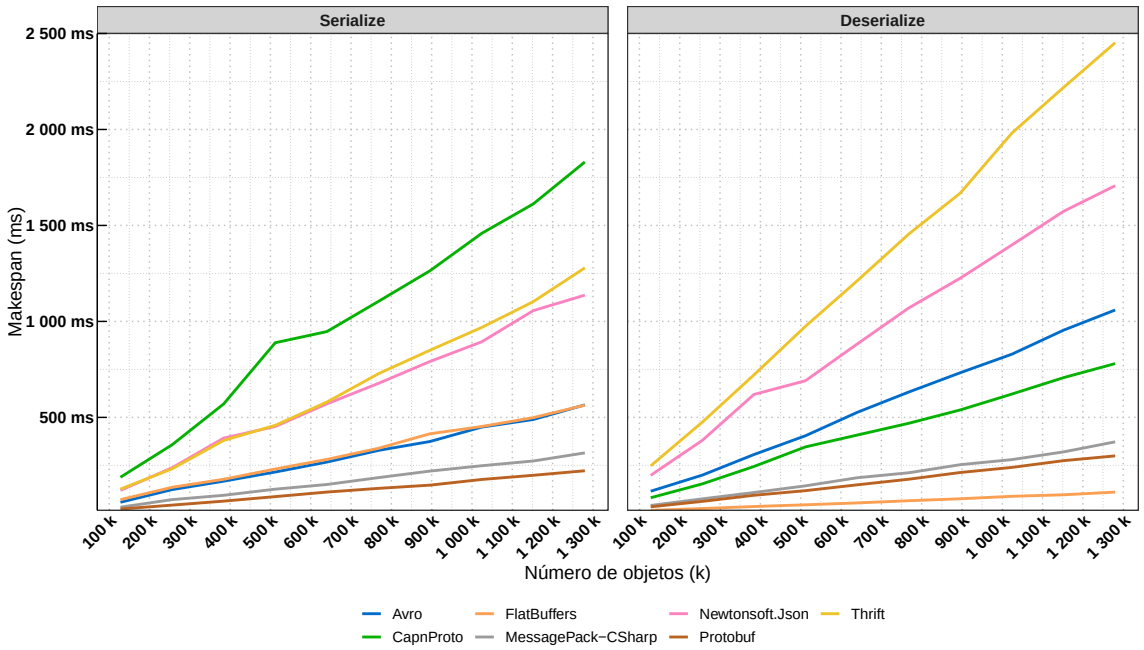
5.4.2 Serialização Serial

Aqui são trabalhados cenários em que o fator a ser observado é o comportamento do serializador com grande volume de objetos de níveis leve, médio e pesado para medir a eficiência de serialização em cenários de maior exigência de recurso. Parâmetros de análise: *número de mensagens*, *tamanho do objeto*.

A Figura 8 apresenta o *makespan* de cada serializador para múltiplas serializações e desserializações seriais do objeto `Channel`. Aqui observa-se que, para serialização, a vazão é maior entre MessagePack e Protobuf, que conseguiram serializar com constância e rapidez diversas cargas de objetos. Para a carga mázima, serializaram em aproximadamente 250ms. Observa-se um pequeno agrupamento entre FlatBuffers e Apache Avro, que aumento gradual constante no tempo de serialização, mesmo sendo rápidos. No entanto, apresentaram o dobro de *makespan* em relação à MessagePack e Protobuf nas cargas observadas. Apresentando maior tempo para serialização estão Newtonsoft.Json, Apache Thrift e Cap'n Proto, com destaque negativo para Cap'n Proto, que mesmo com um objeto simples, demorou consideravelmente mais do que seus concorrentes. Observa-se que, a partir de 512 mil objetos, houve um crescimento repentino em relação à carga anterior, de 384 mil. Apache Thrift e Newtonsoft.Json apresentaram serialização em tempos similares durante todo o experimento. Observando a desserialização, Flatbuffers desponta como o serializador com a melhor constância e velocidade na desserialização, independente da carga de

objetos, considerando o cenário observado. Mesmo com a carga máxima de 1280 mil (k) objetos, conseguiu desserializar em pouco mais de 100ms. Na segunda colocação está o agrupamento de MessagePack e Protobuf, que teve pouco aumento gradual à medida em que o número de objetos crescia. Observa-se um terceiro agrupamento entre Cap'n Proto e Apache Avro, com o segundo sendo ligeiramente mais lento com até 512k objetos, e com maior disparidade gradual a partir de 640k objetos. Por fim, os serializadores que apresentaram pior eficiência e velocidade na serialização de múltiplos objetos estão Newtonsoft.Json e Apache Thrift, sendo Thrift surpreendentemente muito mais lento que seus concorrentes, e com uma curva acentuada à medida em que aumenta-se o número de objetos.

Figura 8 – *Makespan* para serializações seriais do objeto `Channel`

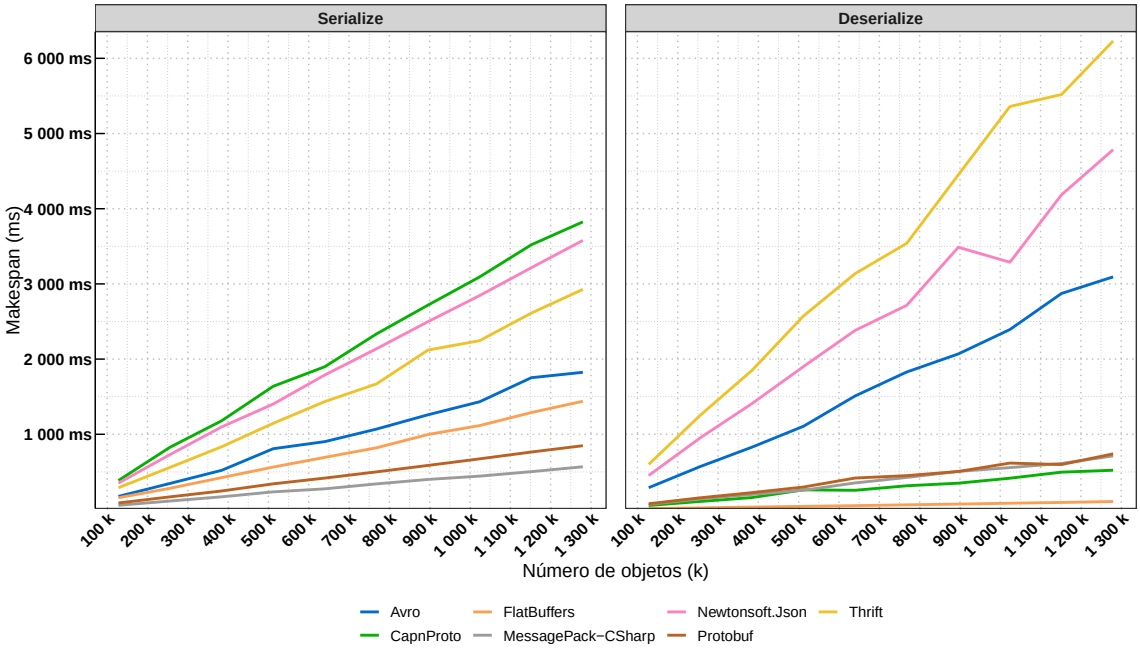


Fonte: Elaborado pelo autor (2023).

A Figura 9 apresenta o *makespan* de cada serializador para múltiplas serializações e desserializações seriais do objeto `SocialInfo`. Aqui, em comparação à Figura 8, observa-se uma maior distribuição nos tempos de serialização e nas curvas de *makespan*. Os mesmos agrupamentos na serialização de `Channel` ainda estão presentes, porém de forma mais discreta. O comportamento e eficiência da serialização de Cap'n Proto ainda é o pior das observações, porém consegue se aproximar da serialização textual de Newtonsoft.Json e da serialização de Apache Thrift. Mesmo com pouca variação, observa-se pequenos saltos nos tempos se serialização de Apache Avro para 512k e 1152k objetos, e um salto ligeiramente mais acentuado de apache Thrift para 896k objetos. MessagePack possui os melhores *makespans* observados em todas as medições de serialização, seguido de perto por Protobuf e, com maior distância, por

FlatBuffers. Quanto à desserialização, Há dois agrupamentos principais e bastante distantes no gráfico: desserializações muito rápidas e desserializações muito lentas. Cap'n Proto, MessagePack e Protobuf tiveram tempos rápidos, constantes e muito similares, enquanto FlatBuffers teve o melhor desempenho, sem sofrer com o aumento de carga. No outro extremo do gráfico estão Apache Thrift com o pior desempenho dentre todos, seguidos por Newtonsoft.Json e Apache Avro, que esteve na metade do gráfico. Com este tipo de objeto, Cap'n Proto teve melhor aproveitamento na desserialização, devendo-se relembrar que este possui três inteiros de 64 bits, além de uma lista de *strings* relativamente curtas. Surpreendentemente, Newtonsoft teve uma melhora no desempenho da desserialização de 896k objetos para 1024k objetos, porém voltou a aumentar seu *makespan* com maior carga.

Figura 9 – *Makespan* para serializações seriais do objeto `SocialInfo`

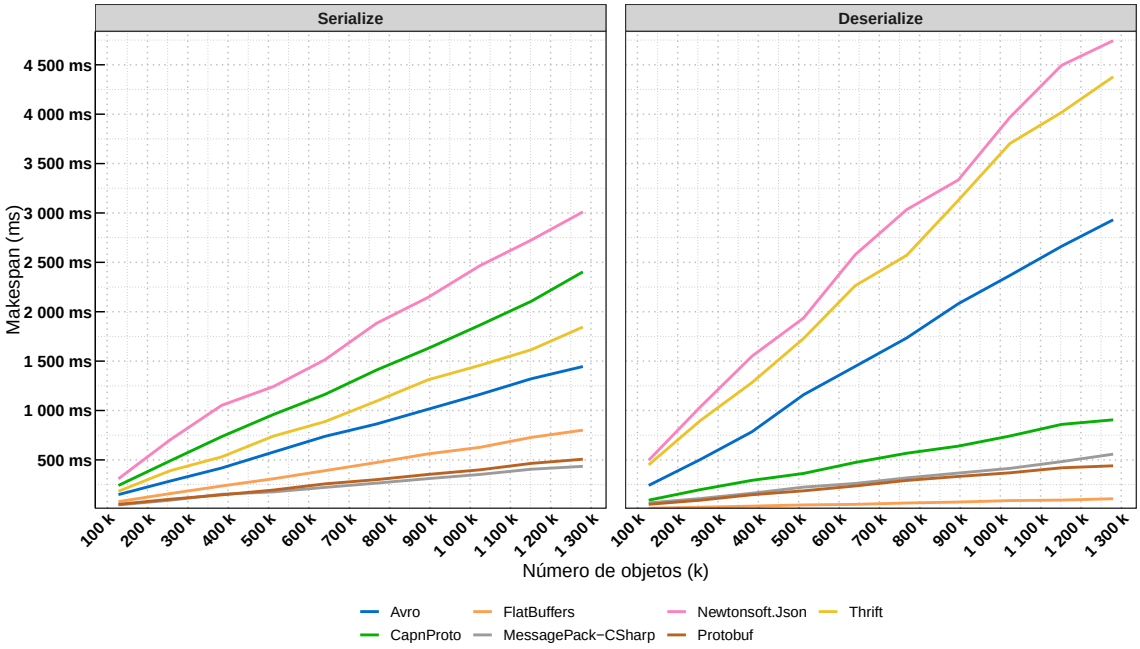


Fonte: Elaborado pelo autor (2023).

A Figura 10 apresenta o *makespan* de cada serializador para múltiplas serializações e desserializações seriais do objeto `VideoInfo`. Neste cenário, observa-se que o tipo e o tamanho dos dados mudou o comportamento de alguns dos serializadores. Para serialização, observa-se que, ao passo que Newtonsoft.Json e os demais mantiveram tempos parecidos em comparação à serialização de `SocialInfo`, Cap'n Proto se saiu melhor serializando este objeto em relação às observações anteriores. Não houve aumento repentino nos tempos de serialização dos serializadores, indicando que não há uma carga para este objeto com uma grande *string* e 3 inteiros de 64 bits que afete a constância e variância dos serializadores. MessagePack, Protocol Buffers e FlatBuffers continuam sendo os 3 melhores serializadores, seguidos por Apache

Avro, Apache Thrift, Cap'n Proto e, por fim, Newtonsoft.Json. Quanto à desserialização, neste caso, quem teve uma melhora no desempenho foi Apache Thrift, que teve *makespan* aproximadamente 1500ms menor em relação à serialização de `SocialInfo`. Cap'n Proto apresentou um tempo de desserialização maior para este tipo de objeto, porém ainda assim teve *makespan* máximo menor do que 1000ms e ficou na quarta colocação. FlatBuffers continuou sendo o líder na desserialização, seguido por Protobuf e MessagePack.

Figura 10 – *Makespan* para serializações seriais do objeto `VideoInfo`

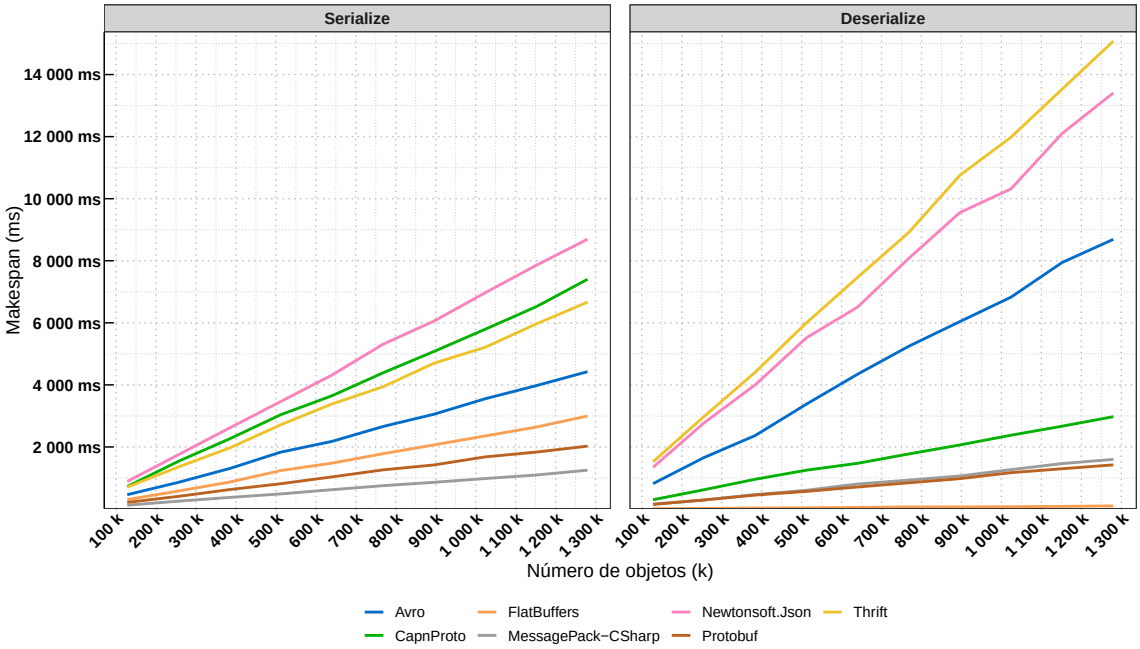


Fonte: Elaborado pelo autor (2023).

A Figura 11 apresenta o *makespan* de cada serializador para múltiplas serializações e desserializações seriais do objeto `Video`. No cenário de serialização, nenhum serializador demorou mais do que 1 segundo para serializar 128k objetos `Video`. Observa-se que MessagePack, Protobuf e FlatBuffers continuam sendo os mais rápidos para serialização, com *makespans* de aproximadamente 1, 2 e 3 segundos, respectivamente, para 1280k serializações de `Video`. No meio termo de serialização está Apache Avro com *makespan* de pouco mais de 4s. Por fim, Apache Thrift, Cap'n Proto e Newtonsoft.Json apresentam serialização em 6,5s ou mais. O comportamento de serialização de acordo com o m Quanto à desserialização, FlatBuffers manteve a liderança com baixíssimo *makespan* em relação aos concorrentes. Protobuf e MessagePack estiveram na faixa abaixo de 2s para serialização, enquanto Cap'n Proto desserializou a carga máxima em aproximadamente 3s. Apache Avro, Apache Thrift e Newtonsoft.Json sofreram com o aumento da carga, tendo os dois últimos uma curva acentuada linear, e com algumas variações no aumento gradual. Apache Avro tam-

bém teve maior curva, porém não tão acentuada quanto os demais, no entanto foi, ao menos, 2.5 vezes mais lento que os 4 primeiros colocados.

Figura 11 – *Makespan* para serializações seriais do objeto `Video`



Fonte: Elaborado pelo autor (2023).

A partir dos cenários observados, pôde-se perceber que alguns serializadores suportam maior carga de objetos na serialização ou desserialização do que outros. Cap'n Proto, nas medições observadas, teve maior eficiência na desserialização do que na serialização de múltiplos objetos, e foi o menos eficiente para serializar objetos simples em relação aos demais. Na desserialização, passou a ser, em média, o quarto mais eficiente na desserialização. O inverso ocorreu com Apache Avro, que teve maior eficiência na serialização do que na desserialização, em que apresentou curva acentuada dado o aumento na carga. MessagePack apresentou tempos rápidos e similares tanto para serialização quanto desserialização, de forma constante e curva suave à medida em que se aumentava a carga. Protocol Buffers teve comportamento similar, com curva suave e tempos rápidos, além de não sofrer variações dado o aumento da carga. FlatBuffers, apesar da serialização não tão rápida quanto Protobuf ou MessagePack, apresentou desserialização a uma velocidade impressionante. Apache Thrift apresentou uma serialização ineficiente e uma desserialização muito ineficiente, que fica ainda mais evidente com maior carga de objetos, sendo a pior na maioria dos cenários. Newtonsoft.Json apresenta o mesmo comportamento de serialização lenta, e desserialização ainda mais lenta, com comportamento não totalmente linear. No entanto, ao considerar alguns casos, a serialização competiu com alguns dos serializadores binários mais ineficientes.

Os serializadores apresentaram, em sua maioria, tendência linear crescente à medida em que o número de objetos aumentava, salvo durante a desserialização para FlatBuffers onde, independente do tipo de objeto, a desserialização era muito rápida e com crescimento quase imperceptível, considerando os outros serializadores.

5.4.3 Serialização Concorrente

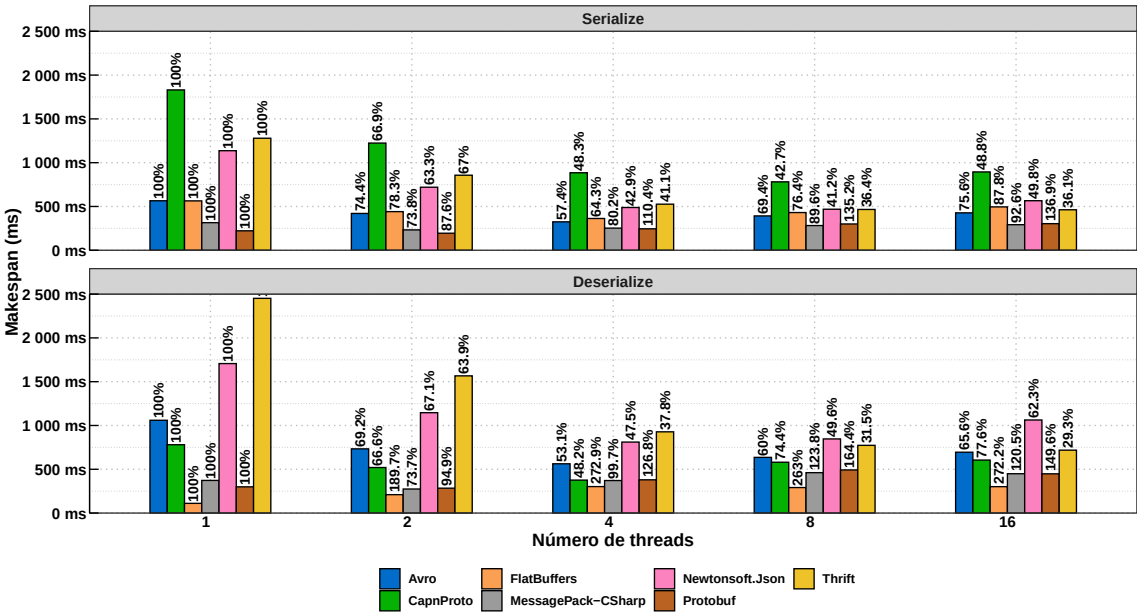
Neste conjunto de testes realizados, visa-se simular um ambiente com alta concorrência, em que múltiplas *threads*, localmente, disputam o mecanismo de serialização por meio da serialização e desserialização de múltiplas mensagens. Aqui é observado como o serializador trabalha em cenários de exclusão mútua quando múltiplos nodos estão disputando o serviço. São instanciadas *threads* antes de cada medição, para aumentar a quantidade de objetos sendo manipulados enquanto o teste ocorre. São alteradas a quantidade de *threads* a serem executadas concorrentemente para análise do comportamento dos serializadores.

Para observar o efeito do uso de variações de *threads* em cenário concorrente no tempo de execução, foram considerados somente a carga máxima de objetos, foram extraídos os gráficos para observar o *makespan* das funções de serialização e desserialização em cada objeto.

A Figura 12 apresenta o *makespan* de cada serializador para múltiplas serializações e desserializações concorrentes do objeto `Channel`, com variação de *threads*. Observando os dados, verifica-se que Protobuf somente consegue extrair desempenho com o uso de concorrência em 2 *threads*, conseguindo executar o trabalho em 87% do tempo. Já com uso de 4 *threads* em diante, deixou de ter ganho de desempenho, passando a demorar 10,4% mais tempo do que com a serialização serial. Na desserialização, ocorreu o mesmo comportamento, tendo percentuais relativos de 94,9% e 126,8%, com 2 e 4 *threads*, respectivamente. Para 8 e 16 *threads*, o cenário piora. Para Apache Avro, tanto para serialização quanto desserialização, atingem-se bons valores percentuais com o uso de concorrência, demonstrando que o serializador consegue fazer bom uso de cenários de exclusão mútua. Para 4 *threads*, foi possível completar o trabalho em apenas 57% do tempo na serialização e 53,1% do tempo na desserialização, um ganho bastante interessante e que começa a equiparar o desempenho deste com os serializadores mais rápidos em *single-thread*. Observando Cap'n Proto, também é feito bom uso de concorrência em todos os cenários avaliados, porém os picos de desempenhos residem em 42,7% na serialização com 8 *threads* e 48,2% na desserialização com 4 *threads*. Aqui, é possível completar o trabalho em menos da metade do tempo, fazendo com que seu desempenho, ao menos na desserialização, seja tão rápido quanto Protocol Buffers. Já para FlatBuffers, foi observado que somente é possível extrair mais desempenho na serialização concorrente. Foi possível atingir o pico de desempenho ao completar o trabalho em 64,3% usando 4 *threads*, e percentu-

ais mais discretos com 2, 8 e 16 *threads*. Na desserialização em *multi-thread*, não foi possível extrair mais desempenho, e FlatBuffers completou o trabalho mais lentamente em todos os cenários. Porém, é interessante ter em vista que a desserialização em *single-thread* já é extremamente rápida para este serializador. Apache Thrift foi o serializador que mais conseguiu extrair desempenho em cenário concorrente, completando a serialização em 36,1% a serialização em 29,3% do tempo, ambas com 16 *threads*. Destaca-se que, a partir de 4 *threads*, já é possível completar o trabalho em aproximadamente 41,1% e 37,8% do tempo, para serialização e desserialização. Portanto, deve-se refletir se a adição do dobro de *threads* compensa o ganho no desempenho. MessagePack foi mais discreto no ganho de performance em *multithreading*, porém teve percentual relativo de 73,8% e 73,7% já na serialização e desserialização em 2 *threads*. A partir de 4 *threads*, já não se torna mais vantajoso o uso de concorrência. Por fim, Newtonsoft.Json teve ganhos surpreendentes com o uso de concorrência, tanto na serialização quanto na desserialização. Foi possível atingir percentual de 41,2% na serialização com 8 *threads* e 47,5% na desserialização com 4 *threads*. Percebe-se aqui que a serialização concorrente foi tão ou mais eficiente que serializadores como Apache Avro, e FlatBuffers usando somente 1 *thread*.

Figura 12 – *Makespan* para serializações concorrentes do objeto `Channel`

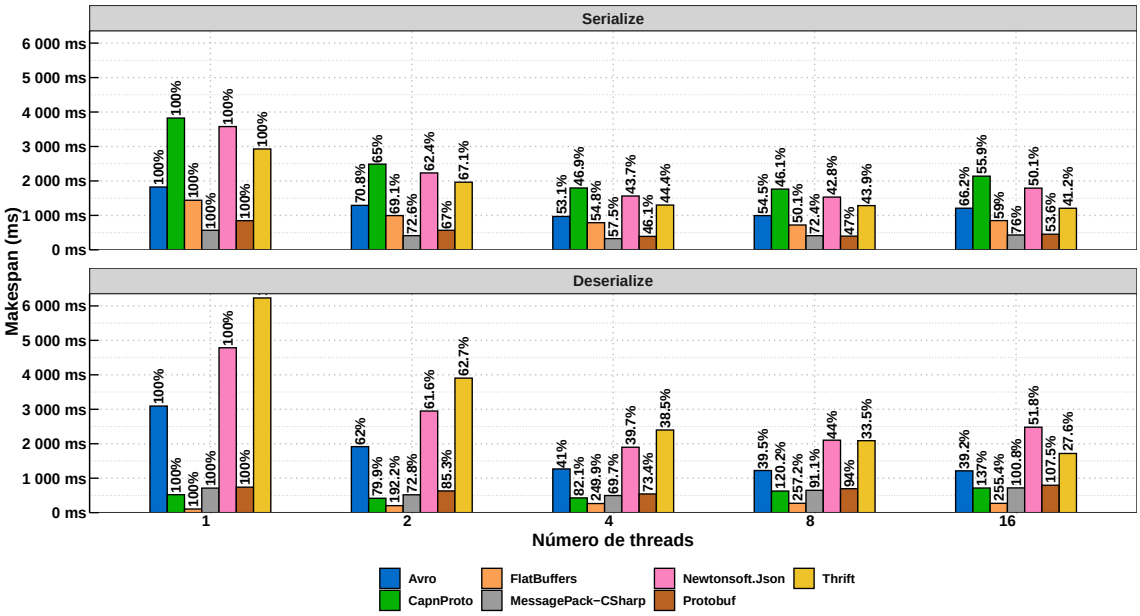


Fonte: Elaborado pelo autor (2023).

A Figura 13 apresentam o *makespan* de cada serializador para múltiplas serializações e desserializações concorrentes do objeto `SocialInfo`, com variação de *threads*. A concorrência na serialização deste tipo de objeto fez com que Apache Avro conseguisse completar o trabalho de serialização em 53,1% do tempo com 4 *threads*, e a desserialização com 39,2% em 16 *threads*. No entanto, já é possível ter percen-

tual relativo de 41% com somente 4 *threads* na desserialização. Aqui, a serialização em *multithread* conseguiu ser semelhante ou mais eficiente que a serialização serial de FlatBuffers e Protocol Buffers. Cap'n Proto consegue extrair mais desempenho na serialização com 8 *threads* e 46,1% do tempo serial e, diferentemente do cenário com o objeto *Channel*, conseguiu também ganho na desserialização com 2 *threads*, atingindo 79,9% do tempo de serialização serial. FlatBuffers conseguiu ter somente ganhos na serialização concorrente, completando a atividade em 50,1% do tempo usando 8 *threads*, ou 54,8% do tempo com 4 *threads*. MessagePack conseguiu extrair bastante desempenho em relação à observação da serialização de `Channel`, tendo completado em 57,5% do tempo a serialização com 4 *threads* e se tornando a serialização múltipla mais rápida para este objeto, com carga máxima. Para desserialização, foi possível atingir 69,7% do tempo com 4 *threads*, ou 72,8% do tempo com 2 *threads*. Newtonsoft.Json novamente conseguiu extrair bastante desempenho com o uso de concorrência, completando a serialização em 42,8% com 8 *threads*, ou 43,7% com 4 *threads*. Na desserialização destaca-se o completamento da atividade em, 39,7% do tempo com 4 *threads*. Quanto a Protocol Buffers, foi possível extrair mais desempenho do que no cenário anterior, completando em 46,1% do tempo a serialização concorrente em relação à serial, e 73,4% do tempo para desserialização concorrente, ambas com 4 *threads*. Por fim, Apache Thrift manteve bom comportamento em *multithreading*, extraindo mais desempenho com até 16 *threads*, e resultados de 41,2% na serialização e 27,6% na desserialização. O que antes era um serializador com desempenho ruim em cenário *single-threaded*, agora se torna um serializador eficiente no contexto concorrente.

Figura 13 – *Makespan* para serializações concorrentes do objeto `SocialInfo`

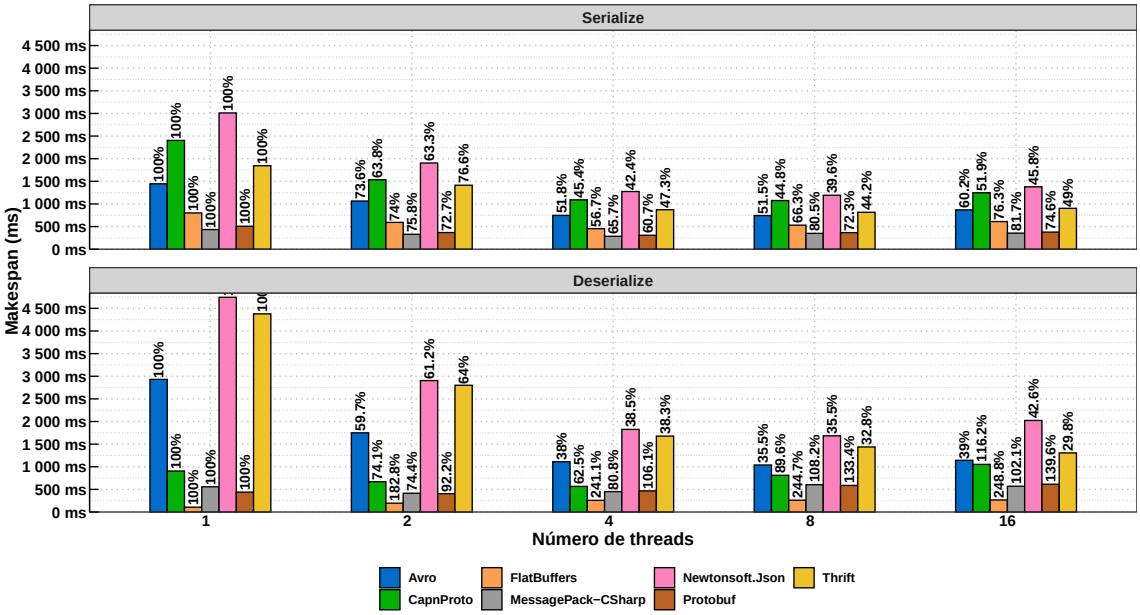


Fonte: Elaborado pelo autor (2023).

A Figura 14 apresentam o *makespan* de cada serializador para múltiplas serializações e desserializações seriais do objeto `VideoInfo`, com variação de *threads*. Para este objeto, Apache Avro manteve o bom comportamento na concorrência em serialização, serializando em 51,5% do tempo, e desserializando em 35,5% do tempo, ambos usando 8 *threads*. Cap'n Proto teve maior eficiência na serialização com 8 *threads* e 44,8% do tempo, porém já é possível atingir desempenho equivalente com 4 *threads*. Para desserialização, fez seu melhor uso de concorrência com 4 *threads* e 62,5% do tempo. Apesar da concorrência, não foi possível ter ganhos superiores a outros serializadores mais rápidos seriais, no entanto a desserialização foi quase tão eficiente quanto Protocol Buffers ou MessagePack. FlatBuffers novamente só conseguiu extrair desempenho na serialização concorrente, em 56,7% do tempo usando 4 *threads*. MessagePack conseguiu ter o pico de desempenho na serialização com 4 *threads* e 65,7% do tempo, enquanto que na desserialização o cenário mais vantajoso foi usando 2 *threads*, com 74,4% de percentual relativo. Para Newtonsoft.Json, novamente houve ganhos tanto na serialização quanto desserialização, completando o trabalho em 39,6% e 35,5% do tempo, ambas com 8 *threads*. No entanto, o cenário com 4 *threads* também é bastante vantajoso. A serialização via Protocol Buffers, neste cenário, também foi vantajosa, porém não tanto quanto na serialização de `SocialInfo`. Aqui, obtiveram-se ganhos somente usando 2 *threads* e 92,2% na desserialização. Para serialização, 4 *threads* e 60,7% foi o pico de desempenho atingido e a serialização mais rápida para este cenário, juntamente com MessagePack. Apache Thrift novamente conseguiu extrair ganhos máximos na serialização com 8 *threads* e 44,2% de percentual, e desserialização com 16 *threads* e 29,8% do tempo em relação à serialização serial.

A Figura 15 apresentam o *makespan* de cada serializador para múltiplas serializações e desserializações seriais do objeto `Video`, com variação de *threads*. Apache Avro teve picos de desempenho na serialização com 4 *threads* e 48,8% do tempo total, ao passo que na desserialização seu desempenho máximo foi apresentado usando 8 *threads* e 35,5% do tempo total. Cap'n Proto conseguiu fazer bom uso da concorrência, tendo picos de desempenho na serialização com 8 *threads* e 45,7% do tempo, e desserialização com 4 *threads* e 46,4% do tempo. FlatBuffers novamente teve ganho somente na serialização, tendo *makespan* relativo de 49% com 8 *threads*, ou 49,9% com 4 *threads*. MessagePack conseguiu completar a serialização em 45,7% do tempo e a desserialização em 52,8% do tempo, ambas com 4 *threads*. Para este caso, a serialização foi a mais eficiente das observadas com o menor número de *threads*. Newtonsoft.Json conseguiu percentuais relativos de 39,2% e 35,5% na serialização e desserialização, respectivamente, ambas com 8 *threads*. Protocol Buffers teve bom desempenho no uso de concorrência para a serialização e desserialização de um objeto complexo como é `Video`. Na serialização, foi possível extrair desempenho com

Figura 14 – *Makespan* para serializações concorrentes do objeto `VideoInfo`

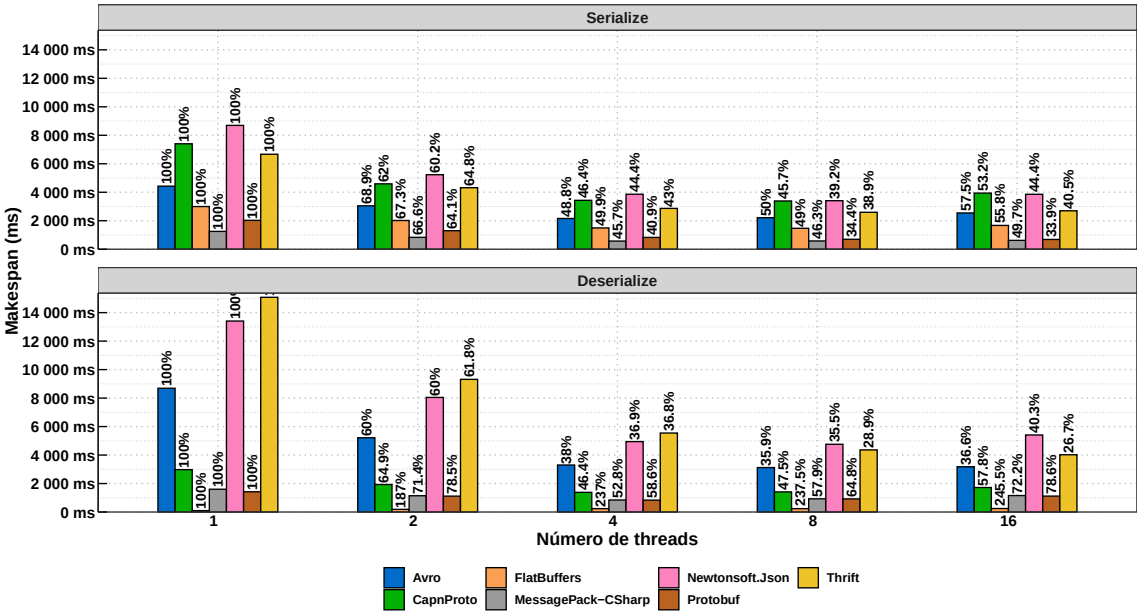


Fonte: Elaborado pelo autor (2023).

até 16 *threads*, completando o trabalho que já era rápido em *single-thread*, em apenas 33,9% do tempo. Ainda para a serialização, o desempenho foi similar ao usar 8 *threads*. Na desserialização o percentual relativo foi mais discreto, com 58,6% do tempo usando 4 *threads*. Apache Thrift continuou com ganhos interessantes por meio do uso de concorrência, e conseguiu atingir 38,9% do tempo com 8 *threads* e se tornando um dos quatro serializadores mais rápidos. Na desserialização, completou o trabalho em 26,7% do tempo usando todas as 16 *threads*.

Ao observar os resultados avaliados e analisados, pode-se concluir que o uso de concorrência traz ganhos a todos os serializadores em quase todos os cenários de serialização e desserialização. Para cada serializador, pode-se atingir o pico de desempenho em diferentes quantidades de *threads*. Apache Thrift, nas medições realizadas, se mostrou um serializador muito ineficiente em *single-thread*, porém com o uso de concorrência, foi possível completar atividades, em determinados cenários em até 1/4 do tempo. Foi possível extrair desempenho em quase todos os cenários com 16 *threads*, o que indica bom comportamento do formato em cenários de exclusão mútua. A implementação realizada fez uso de métodos assíncronos, o que pode ter trazido impacto positivo na concorrência. Apache Avro conseguiu reduzir sua ineficiência na desserialização usando concorrência, e desta forma conseguiu atingir tempos parecidos tanto na serialização quanto desserialização. Para este caso, o ganho maior foi na desserialização, que era um gargalo no cenário *single-thread*. Cap'n Proto conseguiu também reduzir pela metade o tempo de serialização e desserialização em quase todos os cenários, exceto para desserialização de `VideoInfo` e `SocialInfo`, indicando

Figura 15 – *Makespan* para serializações concorrentes do objeto `Video`



Fonte: Elaborado pelo autor (2023).

que *arrays* ou inteiros de 64 bits podem ter atrapalhado o uso de concorrência na recomposição do objeto. MessagePack, mesmo já sendo eficiente, rápido e constante nas medições em *single-thread*, conseguiu atingir tempos ainda mais rápidos tanto para serialização quanto desserialização, mostrando-se um formato versátil e robusto. Seus maiores ganhos foram obtidos com o uso de 2 ou 4 *threads*. Newtonsoft.Json conseguiu extrair muito bem o desempenho ao usar concorrência e múltiplas *threads*, o que levanta a hipótese da possibilidade de seu uso em cenários de alta vazão de dados, mesmo utilizando um serializador textual. Foi possível reduzir o tempo de serialização e desserialização para menos da metade em todas as observações. Sua maior eficiência reside no uso de 4 ou 8 *threads*. O único cenário em que não foi vantajoso, para qualquer objeto, usar concorrência, foi a desserialização com FlatBuffers, sendo mais interessante usar somente 1 *thread*. Isto pode indicar que o gargalo na desserialização para este formato não reside na capacidade de processamento, mas sim na velocidade de acesso à memória, por meio de *zero-copy*. No entanto, como a desserialização de FlatBuffers foi a mais rápida em todos os cenários, não há problemas no uso deste formato de forma serial.

5.4.4 Serialização Distribuída em comunicação HTTP

A realização deste conjunto de testes buscou simular um sistema que permite comunicação com diversos serviços ou clientes, e deve ser capaz de atender a múltiplas requisições simultaneamente, de forma rápida e eficiente. Neste cenário são avaliadas métricas em volta da vazão do servidor e a latência percebida no cliente, am-

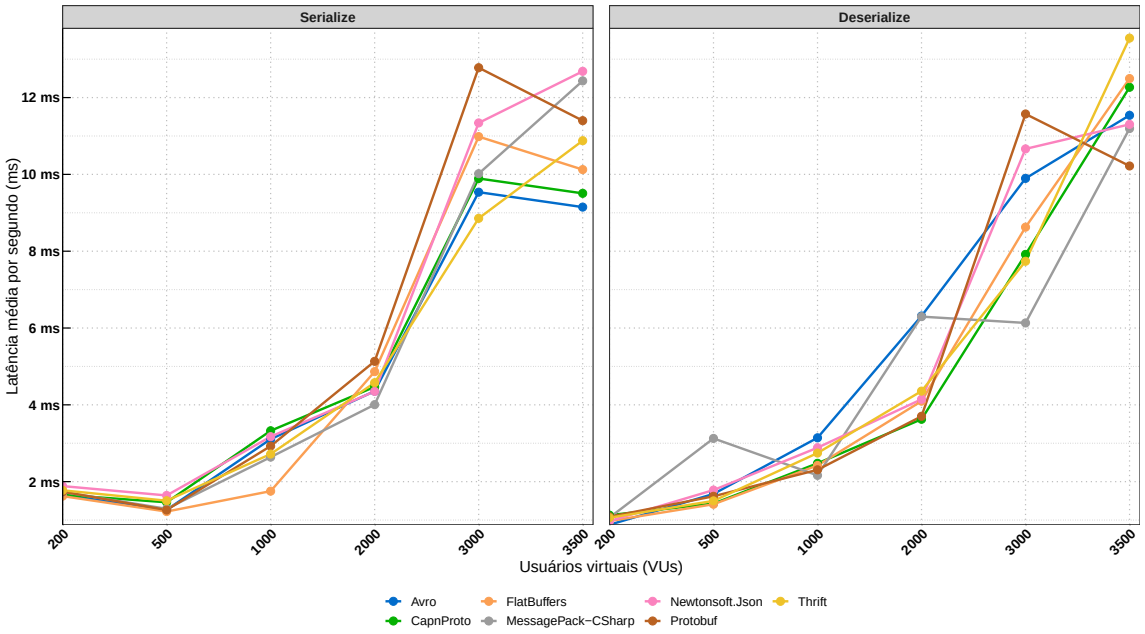
bas com variação de carga em número de clientes simultâneos enviando requisições de serialização ou desserialização. Cada cenário de avaliação foi agrupado por método, número de VUs e serializadores, tendo 60s de tempo de coleta de resultados. A instrumentação desta avaliação não buscou simular um sistema real, visto que, devido à execução no mesmo hospedeiro, não foi possível ativar todas as camadas de rede, e seus respectivos mecanismos, o que alteraria os resultados obtidos nesta subseção.

Percebeu-se, durante a execução dos testes, que, a partir de 3.000 VUs instanciadas pela ferramenta de carga, passou a ocorrer disputa de recursos do sistema, em especial a CPU, para o gerenciamento das requisições simultâneas, provavelmente gerando ruído nas medições. Como as medições foram realizadas em ambiente local e com um único *host*, é possível que o desempenho isolado dos serializadores tenha sido afetado, fazendo-os não atingir seu ponto de saturação máximo em cenário distribuído.

A Figura 16 apresenta a latência média observada para cada serializador, de acordo com o tipo de requisição e o número de clientes simultâneos. Percebeu-se que a serialização para todos os formatos, com exceção de FlatBuffers, que apresentou latência inferior para 1.000 VUs, apresentou uma curva de latência semelhante para até os 2.000 VUs. Para 3.000, foi observado aumento na latência média com uma curva acentuada de todos os serializadores, em especial para Protocol Buffers, Newtonsoft.Json e FlatBuffers. Com 3500 usuários virtuais simultâneos, a latência diminuiu para alguns, o que pode ser explicado processamento requisitado pelo gerador de carda. Na desserialização, foi possível perceber maior latência média por Apache Avro para até 2.000 clientes, o que corrobora com o baixo desempenho na desserialização percebido nas avaliações de *microbenchmark*. Protocol Buffers apresentou curva bem acentuada no salto de 2.000 para 3.000 usuários. MessagePack apresentou mudanças nas latências médias, e ficou com seu comportamento no gráfico em formato de escadas. Para 3.000 clientes simultâneos, executou com a menor latência percebida.

A Figura 18 apresentas a vazão em (milhar) k nos serializadores para desserialização de dados durante o período de observação. O formato do gráfico segue a estrutura parecida instrumentada nas configurações de carga simultânea. No entanto, percebe-se múltiplos pontos de *throttling* em diferentes intervalos de tempo, para diferentes serializadores. Também percebe-se que determinados serializadores conseguiram, manter carga constante, porém a partir de 275s, em que o número de VUs passou a ser de 3.000, não foi mais possível manter a vazão constante, ocorrendo vários picos e vales. Dada esta avaliação, é possível afirmar que, a partir da do cenário de implementação realizado e os serializadores envolvidos, o comportamento é semelhante para todos. Observou-se uso elevado de CPU durante as execuções, mostrando que a capacidade de processamento é um fator de maior peso em relação aos outros recursos da máquina. A vazão máxima percebida girou em torno de 20k a 25k, com

Figura 16 – Latência observada no cliente com variação na quantidade de clientes simultâneos

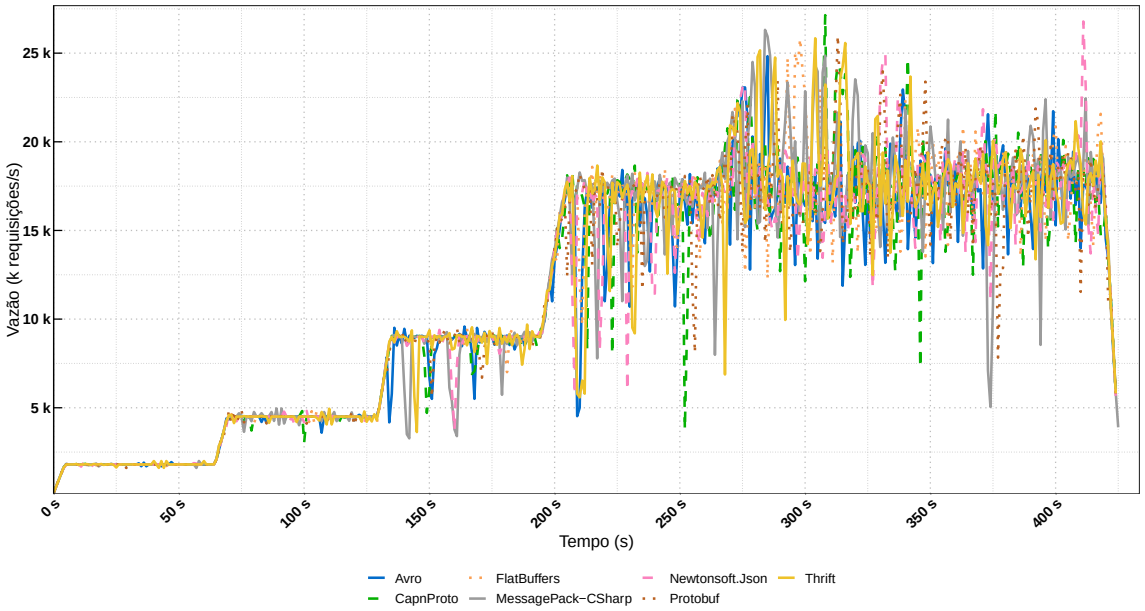


Fonte: Elaborado pelo autor (2023).

variações de acordo com o serializador.

Para receber dados em formato de *array* binário nos controladores HTTP, foi necessária a implementação de um *ByteArrayInputFormatter*. É possível que o recebimento dos dados em formato binário possa ter sido um gargalo neste cenário.

Figura 17 – Vazão na desserialização do servidor

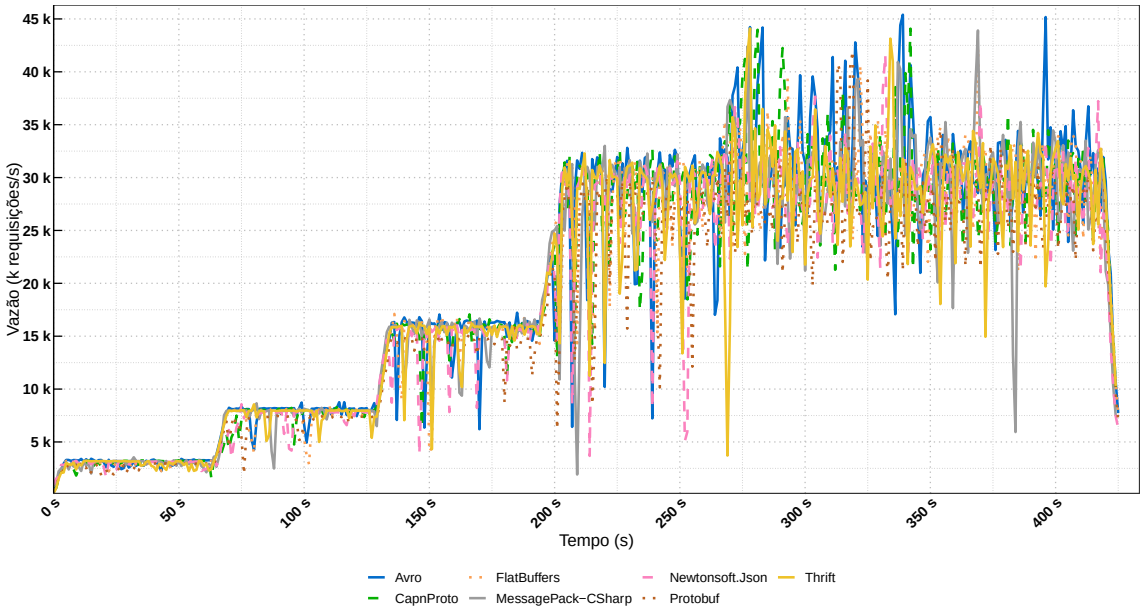


Fonte: Elaborado pelo autor (2023).

Já a Figura 18 apresenta a vazão na serialização dos componentes avaliados ao longo do tempo de observação. Percebe-se uma vazão média maior do que em relação à desserialização, variando de 32k a até 45k. Assim como na desserialização, observou-se *throttling*, sendo a CPU um dos ofensores percebidos durante os testes. Apache Avro apresentou bons picos de vazão com poucos vales no terceiro terço do experimento.

Para transmitir dados em formato de *array* binário nos controladores HTTP, foi necessária a implementação de um *ByteArrayOutputFormatter*. É possível que o envio dos dados em formato binário por meio desta classe possa ter sido um gargalo neste cenário. MessagePack possui suas próprias classes que disponibilizam a comunicação por meio deste formato, e é possível que se possa extrair mais desempenho

Figura 18 – Vazão na serialização do servidor

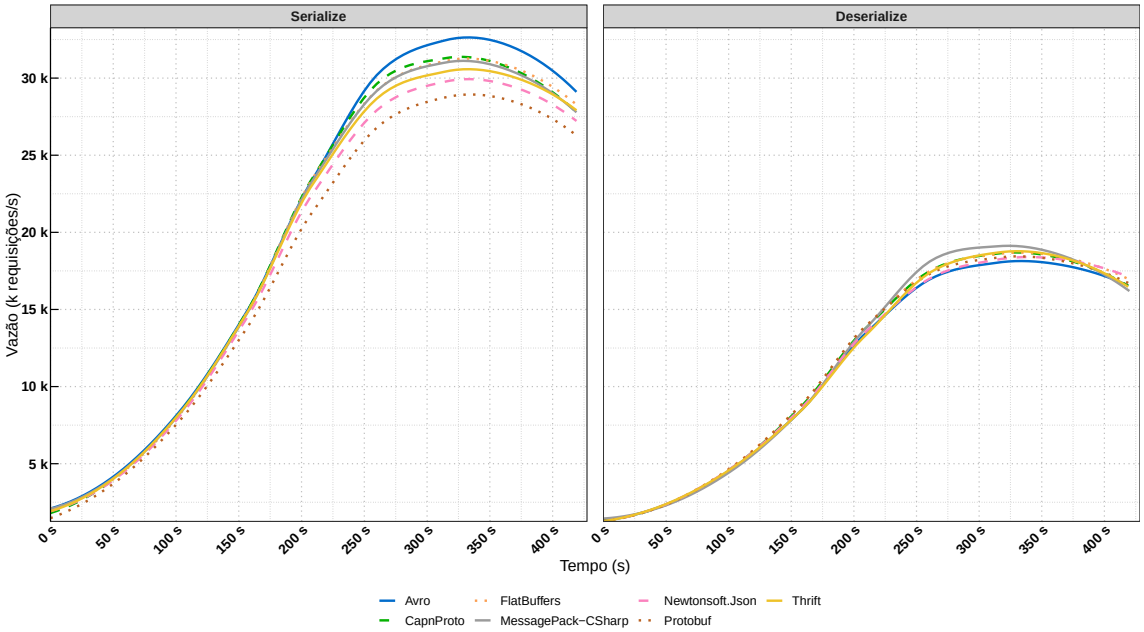


Fonte: Elaborado pelo autor (2023).

A Figura 19 apresenta curvas suavizadas que representam as tendências de vazão observadas ao longo do experimento, distinguidas por operação e serializador. Com este gráfico, é possível ver quais serializadores obtiveram as maiores e menores taxas de serialização durante o período monitorado. Com um número menor de usuários concorrentes na primeira metade do experimento, seja para serialização ou desserialização, a tendência permaneceu semelhante entre todos os serializadores. A partir da metade do tempo de análise, a serialização realizada por Apache Avro em cenário distribuído passou a apresentar maior taxa de vazão dentre os formatos analisados, em que se sobressaiu em cenário de alta concorrência, ao final do experimento. Por outro lado, Protocol Buffers obteve uma menor taxa de vazão ao longo do tempo, em que desde o início do experimento apresentou, mesmo que com pouca

diferença, menor curva de vazão. Em seu pico, observa-se que a diferença chega a ser de 3.000 serializações. Outros serializadores apresentaram comportamentos semelhantes, estando no intervalo entre as curvas de Apache Avro e Protocol Buffers. Na desserialização, Apache Avro teve decréscimo na taxa de vazão no pico, em relação aos concorrentes. MessagePack apresentou melhor taxa de desserialização no pico, mesmo que de forma sutil.

Figura 19 – Vazão média do servidor ao longo do tempo



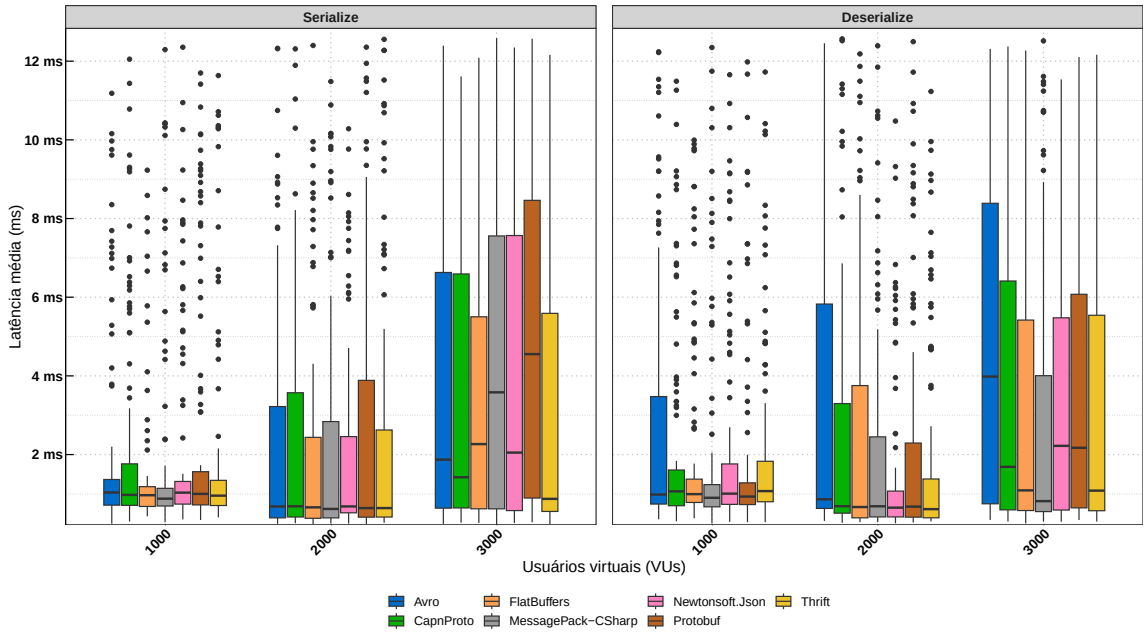
Fonte: Elaborado pelo autor (2023).

Por meio da visualização por diagramas de caixa, a Figura 20 apresenta a distribuição de latências para cada serializador, para os cenários de 1.000, 2.000 e 3.000 VUs simultâneos. Observou-se que, em alguns cenários, mesmo com o aumento da carga de clientes simultâneos, as medições apresentaram latências inferiores. Uma suspeita pode se dar ao fato de que a instrumentação de clientes simultâneos possa ter perdido desempenho, afetando a geração total de requisições. Dado isto, não serão analisados dados para 3.500 VUs. Da mesma forma, cargas como 200 e 500 VUs não apresentam diferenças significativas entre elas para serem analisadas neste gráfico.

Para 1.000 usuários, encontra-se um comportamento semelhante à serialização e desserialização em *microbenchmark*, para múltiplas serializações seriais. A desserialização dos formatos que já se mostraram eficientes nos casos anteriores permanece com desempenho semelhante.

A serialização considerando 1.000 clientes simultâneos apresentou baixos valores, com baixa variabilidade e quartis pouco espaçados, representando uma constância e velocidade na resposta. Na serialização, observou-se que os valores de mediana mantiveram-se estáveis, e até menores, considerando 1.000 e 2.000 VUs. Todos os se-

Figura 20 – Distribuição de valores para latências observadas, considerando 1.000, 2.000 e 3.000 VUs



Fonte: Elaborado pelo autor (2023).

rializadores tiveram baixa mediana de latência, porém começa-se a perceber um maior intervalo de valores na latência percebida. Para 3.000 clientes, todos apresentaram maior intervalo de latência percebida, com medianas maiores, indicando aumento no tempo médio de resposta, e possivelmente uma inconstância na gestão de múltiplos usuários concorrentes no servidor. Dentre estes, as medianas maiores apresentadas foram por MessagePack e Protocdol Buffers, enquanto Apache Thrift e Cap'n Proto tiveram baixas medianas, mesmo com maior range interquartil.

Observa-se que para desserialização com 1000 clientes concorrentes, a latência média e variabilidade de latência para todos os serializadores permanece pequena e com pequeno intervalo de valores, exceto para Apache Avro. Subindo para 2.000 clientes, FlatBuffers e Cap'n Proto passaram a ter maior variabilidade na latência, como demonstra os grandes tamanhos do terceiro quartil e linhas estendidas. No entanto, a mediana permanece em níveis baixos e aceitáveis, assim como o primeiro quartil. Com 3.000 clientes simultâneos, percebe-se aumento geral na variabilidade e o aumento da mediana, indicando que alguns serializadores demoraram, em média, mais tempo para responder. Além disso, o intervalo de valores aumentou consideravelmente. Apache Thrift, FlatBuffers, Cap'n Proto e MessagePack conseguiram manter valores de mediana baixos, mas ainda assim sofreram com alta variabilidade.

Observa-se um número razoável de *outliers* para as cargas menos pesadas nas observações, o que pode ser um ponto de atenção e indicar que o sistema já não possui alta disponibilidade, mesmo com um número médio de usuários simultâneos, seja por

detalhes de implementação do sistema, ou pelo comportamento das bibliotecas.

5.5 AVALIAÇÃO QUALITATIVA

Nesta seção, são elencadas informações quanto a fatores subjetivos de cada formato na linguagem C#, além de compartilhada a experiência de implementação das bibliotecas de serialização. Ao final, são tecidos comentários quanto à cada uma delas. Os formato de serialização *BinaryFormatter* e *Newtonsoft.Json* não serão avaliados nesta seção, por serem usados somente como bases de comparação.

5.5.1 Funcionalidades disponíveis

O propósito desta seção é apresentar e discutir as funcionalidades disponíveis para cada serializador, inclusive aquelas que não foram exploradas na implementação deste trabalho.

Ao analisar as capacidades do MessagePack, percebe-se que, embora não forneça mecanismos de comunicação próprios, permite a aplicação de compressão *LZ4*. Para facilitar seu uso em .NET, a biblioteca oferece *MessagePackInputFormatters* e *MessagePackOutputFormatters*, implementações que facilitam o tráfego do formato de dados *application/octet-stream* para comunicação em HTTP, melhorando o desempenho na comunicação distribuída. No entanto, não há suporte oficial para comunicação em RPC por meio desta biblioteca. É possível serializar um objeto para o formato JSON, e a biblioteca oferece várias ferramentas para garantir compatibilidade com implementações em C# em sistemas existentes. Além disso, permite a serialização de um fluxo de dados contendo múltiplas mensagens, sem a necessidade de construir um esquema. Ao anotar as propriedades de uma classe com chaves de inteiros, a serialização e desserialização se tornam mais rápidas. Em resumo, a biblioteca oferece amplo suporte para implementações em C# em diversos casos de uso.

Cap'n Proto, além de fornecer serialização binária para qualquer tipo em C#, oferece um mecanismo de comunicação em RPC assíncrono e é compatível com mecanismos de compressão para redução do tamanho final da serialização. Também oferece serialização e transmissão de mensagens em fluxo de bytes, embora esta capacidade não tenha sido explorada na construção da mensagem serializada neste contexto. A comunicação implementada não envolveu o uso de *streams*, portanto essa capacidade não foi explorada. Por fim, Cap'n Proto implementa mecanismos de validação e segurança na comunicação para prevenir ataques de amplificação e negação de serviço, entre outros.

Protocol Buffers não possui compressão ou mecanismos diversos de comunicação. Geralmente, é utilizado em conjunto com o arcabouço *gRPC* [14] para uso em

[14] `https://grpc.io/`

RPC. Além da serialização binária, fornece serialização em texto ou *JSON*. Sua implementação em C# não oferece mecanismos para uso de *streams*, e seu formato não é orientado a grandes conjuntos de dados. Embora haja classes de serialização em *JSON*, não foi possível confirmar se a serialização para este cenário é possível. Protocol Buffers destaca que seu mecanismo de serialização é *thread-safe*, indicando que não seria necessário implementar controle de acesso a dados e processamento *multithread*.

Apache Thrift disponibiliza, além dos mecanismos de serialização, maneiras de transporte da informação em distintos formatos de transporte e protocolo. Para C#, fornece tipos de transporte em baixo e alto nível, como memória, *Pipes*, Sockets, TLS, Framed e HTTP. Quanto aos protocolos de codificação, oferece opções como Binário, Compacto (*Thrift Compact Protocol*), JSON e Multiplexado. Em relação aos servidores, possibilita o uso de operações não bloqueantes. No entanto, até o momento, o suporte está disponível somente até a versão de arcabouço .NET 6, conforme informações do site oficial e matriz de funcionalidades.

FlatBuffers possui uma matriz de suporte a funcionalidades, distinguida por linguagem. A biblioteca oferece serialização em JSON se a configuração para este tipo de serialização for parametrizada no momento da compilação do esquema. Com um foco mais acentuado na serialização, especificamente binária, FlatBuffers não apresenta mecanismos próprios de comunicação e funcionalidades distintas. Não oferece suporte a mecanismos de compressão, e, de maneira mais restrita, fornece ferramentas para comunicação em RPC.

Apache Avro possui uma variedade maior de formatos de codificação e transporte. É possível realizar a serialização em *JSON* ou de forma binária, e utilizar tanto transporte em HTTP como por memória. O formato oferece suporte a fluxos de mensagens, sendo reconhecido por ter um desempenho superior nesse cenário. Avro também apresenta suporte a RPC, mecanismos de compressão como *bzip2*, *snappy*, *xz*, *zstandard* e *deflate*.

5.5.2 Qualidade na documentação

A documentação disponível para CapnProto em C# foi acessada por meio da *wiki* do repositório que a implementou, não fazendo parte da documentação oficial do site. A descoberta desta documentação demorou, no entanto, revelou-se bastante completa, possibilitando uma implementação rápida e bem-sucedida da serialização. A documentação inclui exemplos práticos e explicações detalhadas das diversas funcionalidades do serializador. A única dificuldade na compreensão da documentação surgiu ao perceber que a compilação do esquema ocorre em tempo de *build*.

Por outro lado, o Protocol Buffers disponibilizou uma documentação ampla e completa, contendo exemplos em C# referenciados no portal oficial. A compreensão

da base para implementação foi rápida devido à qualidade desta documentação.

MessagePack também possui uma boa documentação oficial e em seu repositório para C# [15]. Sua implementação e entendimento de funcionalidades foram simples, com exemplos claros de como extrair as funcionalidades da ferramenta, apresentando diversas abordagens de disposição de conteúdo.

A documentação do Apache Thrift fornece, em sua maioria, explicações em alto nível das funcionalidades existentes. Para C#, no entanto, não há documentação ou exemplos em seu site oficial[16]. Foi necessário mais pesquisa e esforço para compreender suas funcionalidades, sendo que os exemplos de uso foram encontrados somente após o desenvolvimento da aplicação, no repositório oficial[17], não explicitamente apresentados.

Quanto ao mecanismo FlatBuffers, observou-se durante a análise da documentação que sua matriz de funcionalidades está desatualizada. A apresentação de funcionalidades e exemplos de uso está contida tanto em seu repositório oficial[18] quanto na documentação oficial [19]. Sua documentação de API e especificação do formato é extensa e contém informações suficientes para a implementação da serialização e desserialização.

Ao analisar a documentação presente em Apache Avro, observou-se que seu repositório oficial contém informações suficientes para compreender a especificação do formato, mas não há exemplos de uso na linguagem usada neste trabalho[20].

Apache Avro possui pouca documentação no que se refere a exemplos de uso, o que limitou a exploração das funcionalidades do formato. Há apenas exemplos de uso em Java, e encontrar exemplos exigiu um esforço considerável. No site oficial, a documentação para C# está vazia. Em geral, sua documentação dificultou bastante o entendimento para a implementação de sua API, sendo assim o formato com menor qualidade de documentação.

5.5.3 Implementação e integração com o sistema existente

Entre os serializadores implementados, destaca-se a facilidade de implementação do MessagePack. Com poucas linhas e baixa interferência no sistema atual, revelou-se um formato simples de ser usado e testado. A integração não apresentou dificuldades, e não foram necessárias adaptações para comunicação distribuída, já que sua codificação transforma o objeto em um *array* binário. A serialização e desserialização consideraram as classes de domínio originais, eliminando a necessidade de criar

[15] `https://github.com/MessagePack-CSharp/MessagePack-CSharp`
[16] `https://thrift.apache.org/lib/csharp.html`
[17] `https://github.com/apache/thrift/blob/master/tutorial/netstd/Server/Program.cs`
[18] `https://github.com/google/flatbuffers/blob/master/docs/source/CsharpUsage.md`
[19] `https://flatbuffers.dev/flatbuffers_guide_use_c-sharp.html`
[20] `https://avro.apache.org/docs/1.11.1/api/csharp/html/`

classes intermediárias, o que agilizou bastante a implementação. Foi o serializador mais rápido para implementar e integrar.

A implementação da serialização em FlatBuffers exigiu a escrita de muitas linhas de código, especialmente para a serialização de objetos complexos, como `Video`, o que demandou refatoração e reutilização de funções estáticas de apoio. A adição de novas propriedades requer recompilação e ajustes na implementação da serialização. A desserialização foi transparente em termos de controle de propriedades, mas é necessário atentar para a conversão do formato intermediário para o final. Seus métodos para serialização de propriedades são de fácil compreensão, e não houve dificuldades na construção do serializador deste ponto de vista. Foi possível ter o controle das operações realizadas, e a facilidade nos testes e investigação de problemas na implementação por meio de *debug* foi um ponto positivo. Tanto na serialização quanto na desserialização, foi necessário realizar checagem de tipos.

Apache Avro foi o serializador mais difícil de implementar devido à escassa documentação, exigindo uma extensa pesquisa para compreender o fluxo de serialização em um sistema construído em C#. Durante a implementação, foram testadas duas bibliotecas principais: *AvroConvert* e *Apache.Avro*. A primeira fornece uma interface simples de uso e permite a serialização sem uso de esquema *JSON*, descobrindo o esquema durante a própria serialização. No entanto, a implementação apresentou problemas na serialização de `Video`, tornando a investigação de problemas bastante difícil. Por fim, foi testada a abordagem com uso de esquema. A geração de esquema estático contou com o apoio de uma ferramenta externa para a extração do formato em *JSON* das classes principais, para posterior geração de arquivo de esquema e compilação subsequente. Sua serialização exigiu a manipulação de uma classe de representação intermediária, mas a serialização foi genérica a partir disso. Cada classe intermediária possuía seu próprio esquema em *string JSON*, usado como parâmetro na serialização. Na desserialização, foi necessário realizar a checagem de tipos para que fosse possível utilizar o esquema e desserializar o *array* binário no objeto intermediário. Para a adaptação ao sistema, foi necessário converter o fluxo de bytes para um *array* binário ao final da serialização.

A implementação do serializador Cap'n Proto foi simples, após a descoberta da documentação com exemplos de uso. Sua compilação, por ser em tempo de *build*, permitiu ajustes rápidos no esquema, agilizando o desenvolvimento. Foi necessário realizar a checagem de tipos tanto na serialização quanto na desserialização, assim como a manipulação da representação intermediária. Em geral, observa-se que a manutenibilidade deste tipo de serializador pode ser custosa em sistemas com grande variabilidade nos contratos de dados. Apesar disso, o código de serialização funcionou de maneira genérica e apresenta pouca necessidade de manutenção. Para a adaptação ao sistema, foi necessário converter o fluxo de bytes para um *array* binário ao final

da serialização.

Quanto ao Protocol Buffers, o código gerado para a construção do serializador foi pequeno. Apesar da gestão de classes intermediárias e da checagem de tipos na desserialização, os métodos foram simples e não exigiram manipulação direta das propriedades de cada classe, sendo necessário usar o mesmo método disponível em cada classe intermediária para serialização e desserialização. Seu resultado final é um *array* de bytes, portanto não foi necessária adaptação para comunicação distribuída.

No caso do serializador Apache Thrift, a construção do código funcional foi relativamente complexa, devido às diversas opções de transporte, compactação e servidores. Os métodos de serialização e desserialização assíncronos eram os únicos disponíveis, e como a implementação, de forma geral, era síncrona, foi necessária adaptação. No entanto, em sistemas que fazem uso dessa capacidade, torna-se uma opção interessante. Foi necessária a gestão de classes intermediárias, além da checagem de tipos na desserialização. Contudo, o código de serialização passou a funcionar de maneira genérica e exigiu pouca manutenção. Também foi necessária a conversão de fluxo de bytes para *buffer* binário.

5.5.4 Interoperabilidade

Esta subseção destina-se a compilar os mecanismos disponíveis que favorecem a interoperabilidade entre sistemas. A lista a seguir compila as linguagens suportadas por cada formato de serialização:

- **Protocol Buffers:** C++, C#, Dart, Go, Java, Objective-C, Python, Ruby, Kotlin e PHP.
- **MessagePack:** ActionScript, C, C++, C#, Clojure, Crystal, D, Dart, Delphi, Elixir, Erlang, F#, Go, GNU Guile, Haskell, Haxe, HHVM, J, Java, JavaScript, Julia, Kotlin, Nim, MATLAB, OCaml, Objective-C, Pascal, PHP, Perl, Pony, Python, R, Racket, Ruby, Rust, Scala, Scheme, Smalltalk, SML e Swift.
- **FlatBuffers:** C++, C#, C, Go, Java, Kotlin, JavaScript, Lobster, Lua, TypeScript, PHP, Python, Rust e Swift.
- **Cap'n Proto:** C, C++, C#, D, Erlang, Go, Haskell, Java, JavaScript, Lua, Nim, OCaml, Python, Ruby, Rust e Scala.
- **Apache Thrift:** ActionScript, C, C++, C#, Common LISP, D, Dart, Erlang, Haskell, Haxe, Go, Java, JavaScript, Lua, OCaml, Perl, PHP, Python, Ruby, Rust, Smalltalk e Swift.
- **Apache Avro:** C, C++, C# e Java.

Dentre os distintos formatos e protocolos de comunicação, apenas Apache Avro, Apache Thrift e Cap'n Proto apresentam suporte, incluindo RPC. A serialização binária e JSON é fornecida por todos os serializadores avaliados, permitindo interoperabilidade

no formato de dados. Com exceção de FlatBuffers e Protocol Buffers, todos também apresentam suporte a fluxos de dados.

6 CONCLUSÃO

Este capítulo destaca conclusões e resultados obtidos durante o desenvolvimento, abordando as principais conclusões derivadas da análise de formatos de serialização eficiente. A serialização binária, notadamente crucial para otimizar o desempenho e a vazão em sistemas que lidam com grandes volumes de dados, é um ponto focal.

O objetivo principal deste trabalho foi conduzir um estudo abrangente sobre bibliotecas de serialização comumente empregadas em sistemas distribuídos de alta vazão na linguagem C#. As bibliotecas analisadas incluíram MessagePack, Apache Avro, Apache Thrift, Protocol Buffers, FlatBuffers e Cap'n Proto. Como base de comparação, foram utilizados os serializadores Newtonsoft.Json, formato de serialização textual em JSON usado amplamente em projetos .NET, e *BinaryFormatter*, serialização binária depreciada, utilizado para extração dos tamanhos em memória dos objetos em formato não serializado.

Como principais contribuições deste trabalho, foi possível agregar diversas informações sobre bibliotecas de serialização binárias disponíveis para a linguagem C#, elencando suas funcionalidades e características disponíveis. Quanto ao âmbito quantitativo, realizou-se a agregação dos resultados das bibliotecas analisadas em comparação à biblioteca de serialização textual Newtonsoft.Json, tanto com a serialização individual quanto múltipla. Além disso, foram extraídas métricas de desempenho de cada biblioteca com o uso de concorrência em múltiplas *threads*, buscando apresentar o ganho proporcional de desempenho ao usar uma abordagem que busque melhor utilização das *threads* disponíveis. Ao mesmo tempo, foram apresentadas características empíricas que visam enriquecer a análise e escolha de um formato de serialização de acordo com a necessidade observada, levando em consideração as funcionalidades disponíveis, assim como possíveis dificuldades de manutenção e implementação. Dado um ambiente com comunicação em HTTP, foi possível observar a vazão e latência média na serialização e desserialização de um sistema com recursos computacionais das características apresentadas, implementado em C#.

Dentre as bibliotecas avaliadas, algumas conseguiram extrair bom desempenho em todos os cenários avaliados em *microbenchmark*. MessagePack, além de ser uma biblioteca de fácil implementação e baixa manutenibilidade, conseguiu ter boa taxa de compressão, velocidade de serialização e desserialização para carga individual e múltipla de objetos, e conseguiu extrair ainda mais desempenho em cenário concorrente MessagePack não exige esquema compilado, e por meio de poucas anotações nas classes, conseguiu serializar propriedades de forma dinâmica e sem representação intermediária. Protocol Buffers teve atuação semelhante, porém não conseguiu extrair tanto desempenho em cenário concorrente em todos os cenários, como quanto a bi-

blioteca anterior. Ambos serializadores apresentaram pouca variabilidade de acordo com o formato de dados e tamanho da carga, e desempenharam seu papel de forma eficiente.

FlatBuffers foi protagonista nos tempos de desserialização, e foi o serializador com a maior velocidade e menor uso de memória durante a desserialização entre todos os concorrentes, por uma margem proporcional grande. No entanto, sua implementação exige controle refinado da serialização das propriedades em cada objeto. Isto faz com que seja necessário acessar o código de serialização e fazer alterações sempre que o modelo de dados é alterado, e para determinados cenários isto pode ser um fator negativo para a escolha deste formato.

A partir dos resultados obtidos por um serializador textual, como no caso de NewtonSoft.Json, foi observado que seu desempenho é bem menor que os concorrentes na serialização serial, principalmente com uma grande carga de objetos. No entanto, esta disparidade foi bastante reduzida com o uso da estratégia de concorrência. Desta forma, o uso de serialização textual em cenário concorrente mostrou-se possível em cenários de alta carga de dados.

Apache Thrift Apresentou o pior desempenho dentre todos os serializadores para trabalho com múltiplas cargas de forma serial. Em muitos casos, foi até menos eficiente do que a serialização textual. Em cenário concorrente, no entanto, conseguiu extrair o máximo de desempenho na concorrência, chegando a completar o trabalho em até um quarto do tempo serial, com o máximo de *threads* disponíveis nos experimentos, e isto pode ter relação direta com o uso de métodos assíncronos de serialização e desserialização. Com o uso de 16 *threads*, passou das últimas colocações para ter desempenho parecido ou melhor que os formatos mais rápidos de serialização observados.

Apache Avro, devido à pouquíssima documentação, foi o serializador com a maior dificuldade de implementação, e seus resultados não foram positivos. Para os casos avaliados, demonstrou ter maior eficiência na serialização do que na desserialização. Este serializador possui mais funcionalidades, e pode ser que ele desempenhe melhor em outros cenários, como em serviços de troca de mensagens e processamento de *streams*.

Cap'n Proto, por sua vez, possui desserialização eficiente e serialização ineficiente. No cenário de múltiplas serializações, apresentou o maior *makespan* para serialização de um objeto simples como `Channel`, demonstrando variabilidade de acordo com o formato de dados. Para se obter maiores detalhes e encontrar o motivo disto, é necessário analisar detalhadamente sua implementação na biblioteca em C#.

De forma geral, em *microbenchmark*, todos os serializadores desempenharam seus papéis, porém observa-se uma clara distinção entre serializadores constantemente eficientes, e serializadores eficientes de acordo com o cenário.

Observando os resultados deste trabalho, é possível destacar que FlatBuffers, Protocol Buffers e MessagePack tiveram as melhores avaliações em cenário local. Ao escolher um serializador binário de propósito geral para sistema distribuído de alta vazão, independente da necessidade, é recomendado o uso de um destes três. Para casos em que a desserialização é o enfoque, além dos formatos já destacados, Cap'n Proto é uma opção viável. Quando se trata da serialização, Apache Avro também pode ser utilizado. Apache Thrift não teve bom desempenho nas avaliações em *microbenchmark*, e seu uso só é recomendado em cenário concorrente com múltiplas *threads*.

Em resumo, em se tratando de serialização no cenário de *microbenchmark* para C#, o formato com melhor desempenho foi MessagePack. Este formato é muito simples de ser implementado e também apresentou os melhores resultados da avaliação.

Dentre os resultados distribuídos, somente observaram-se leves destaques, positivo na vazão da serialização para Apache Avro, e negativo para Protocol Buffers. Apache Thrift apresentou boas médias de latência até mesmo para uma grande número de cliente simultâneos, o que demonstra boa capacidade para lidar com múltiplas requisições. Na desserialização, a baixa latência para múltiplos clientes pôde ser observada também para FlatBuffers, MessagePack e Cap'n Proto.

Os resultados dos experimentos distribuídos, apesar de terem sido afetados por conta do mesmo ambiente compartilhado entre o gerador de carga e o sistema de serialização, mostraram uma visão parcial da vazão e latência de um sistema de tráfego intensivo de dados que implementa serialização eficiente. Foi possível observar os limites de serialização em que o sistema não apresenta *throttling*, e o número de requisições atendidas por segundo na serialização, que foi aproximadamente duas vezes maior do que na desserialização. É possível que outros fatores da implementação tenham impactado a análise, visto que o comportamento de todos os serializadores foi, de certa forma, parecido, indicando que o limite do sistema não foi atingido. De qualquer forma, mesmo com a implementação atual, a implementação de estratégias de replicação de servidores em conjunto com a adoção de uma biblioteca de serialização eficiente pode aumentar bastante a capacidade geral de um sistema, o que é um bom sinal.

É necessário reiterar que os resultados de cada serializador estão diretamente relacionados com a forma como foi implementada a serialização, e é possível que a implementação realizada neste trabalho é não seja a mais eficiente possível e que faz melhor uso das ferramentas disponíveis para cada formato.

6.1 TRABALHOS FUTUROS

A análise do comportamento de bibliotecas de serialização em ambiente distribuído pode ser expandido para muito mais esferas. O tema de serialização eficiente

engloba muitos aspectos da programação distribuída e problemas reais de software para as mais diversas linguagens e cenários. Para assegurar uma maior confiabilidade na avaliação do comportamento das bibliotecas em ambientes distribuídos, propõe-se a realização dos testes em um ambiente de *testbed*. Este ambiente deve replicar as condições reais de comunicação em rede, ativando os mecanismos de controle de fluxo e ajustando os parâmetros de qualidade de serviço. Idealmente, seria em um ambiente especializado projetado para otimizar o desempenho dos sistemas, permitindo a obtenção de métricas mais precisas e refinadas, tais como análise do ponto de saturação e da vazão da aplicação. Além disso, executar testes de carga por maior tempo pode permitir maior controle das observações e uma melhor visualização dos comportamentos de serialização. Perfis de carga como *spike*, *breakpoint* e *soak* também se mostram candidatos interessantes para avaliar como cada serializador reage a diferentes características de carga.

A comunicação eficiente com serialização binária também pode ser testada em conjunto com outros protocolos de comunicação, tanto na camada de aplicação quanto na camada de transporte, e as capacidades de tráfego de uma grande quantidade de informação, assim como em *streams* de dados, deve ser avaliada.

As estratégias de serialização eficientes podem ser aplicadas em soluções existentes, de modo que seja possível mensurar o ganho de desempenho real em uma aplicação executando em produção, em larga escala. Pode-se levar as implementações realizadas para uma aplicação concreta, como sistemas de alta vazão, protocolos de fusão, replicação de dados ou um sistema que implemente a comunicação por meio de microsserviços.

Trata-se de um campo bastante amplo de pesquisa e que pode ser explorado junto de outros temas da área de sistemas distribuídos.

REFERÊNCIAS

ABERNETHY, William. **Programmer's Guide to Apache Thrift**. [*S.l.*]: Simon e Schuster, 2019.

ADIBFAR, Alireza; COSTIN, Aaron M. Evaluation of IFC for the Augmentation of Intelligent Transportation Systems (ITS) into Bridge Information Models (BrIM). American Society of Civil Engineers Reston, VA, p. 177–184, 2019.

ADIBFAR, Alireza; COSTIN, Aaron M. Review of Data Serialization Challenges and Validation Methods for Improving Interoperability, p. 522–529, 2021.

ANTIREZ. **Redis: An open-source, in-memory data structure store**. Software. 2009. Disponível em: `https://redis.io/`.

BORMANN, C.; SHELBY, Z. **The Constrained Application Protocol (CoAP)**. Am Fallturm 1, 28359 Bremen, Germany, jun. 2014. Disponível em: `https://datatracker.ietf.org/doc/html/rfc7252`.

BRAY, T. **The JavaScript Object Notation (JSON) Data Interchange Format**. Internet, mar. 2014. `http://www.rfc-editor.org/rfc/rfc7159.txt`. Disponível em: `http://www.rfc-editor.org/rfc/rfc7159.txt`.

BRAY, Tim; PAOLI, Jean; SPERBERG-MCQUEEN, C Michael; MALER, Eve; YERGEAU, François; COWAN, John. Extensible markup language (XML) 1.0. W3C recommendation October, Internet, 2000.

CASEY, Allen *et al.* **Performance of Serialization Libraries in a High Performance Computing Environment**. 2022. Tese (Doutorado).

CHANG, Bao Rong; TSAI, Hsiu-Fen; WANG, Yo-Ai; KUO, Chin-Fu. Intelligent adaptation to in-cloud NoSQL database remote backup between data centers, p. 1–6, 2015.

COMBS, Gerald. **Wireshark**. `https://www.wireshark.org/`, 1998. Accessed on 2023-06-15.

COSTIN, Aaron; EASTMAN, Charles. Need for interoperability to enable seamless information exchanges in smart and sustainable urban systems. **Journal of**

Computing in Civil Engineering, American Society of Civil Engineers, v. 33, n. 3, p. 04019008, 2019.

DOTNET/BENCHMARKDOTNET. **BenchmarkDotNet**. 2013. Disponível em: https://github.com/dotnet/BenchmarkDotNet.

FACEBOOK. **Zstandard - Fast real-time compression algorithm**. Accessed: 2023-06-15. 2016. Disponível em: https://github.com/facebook/zstd.

FIELDING, Roy Thomas. **Architectural Styles and the Design of Network-based Software Architectures**. Jun. 2000. Tese (Doutorado) – University of California, Irvine.

FIRESTONE, Joseph M; MCELROY, Mark W. Doing knowledge management. **The learning organization**, Emerald Group Publishing Limited, 2005.

FOUNDATION, Apache Software. **Apache**. Accessed on 2023-06-15. 2023. Disponível em: https://www.apache.org/.

FOUNDATION, Apache Software. **Apache Cassandra**. Accessed on 2023-06-15. 2023. Disponível em: https://cassandra.apache.org/.

FOUNDATION, Apache Software. **Apache Hadoop**. Accessed on 2023-06-15. 2023. Disponível em: https://hadoop.apache.org/.

FOUNDATION, Apache Software. **Apache HBase**. Accessed on 2023-06-15. 2023.

FOUNDATION, Apache Software. **Apache Kafka**. Accessed on 2023-06-15. 2023. Disponível em: https://kafka.apache.org/.

FOUNDATION, Apache Software. **Apache Spark**. Accessed on 2023-06-15. 2023. Disponível em: https://spark.apache.org/.

FOUNDATION, The Apache Software. **Apache Avro**. 2022. Disponível em: https://avro.apache.org/.

FOUNDATION, The Apache Software. **Apache License 2.0**. Accessed on 2023-06-15. 2004. Disponível em: https://www.apache.org/licenses/LICENSE-2.0.

FOUNDATION, The Apache Software. **Apache Thrift**. 2022. Disponível em: https://thrift.apache.org/.

FROZZA, Angelo Augusto; SANTOS MELLO, Ronaldo dos; COSTA, Felipe de Souza da. An approach for schema extraction of JSON and extended JSON document collections, p. 356–363, 2018.

FURUHASHI, Sadayuki. **MessagePack: It's like JSON. but fast and small.** 2008. Disponível em: https://msgpack.org/.

GABRIEL, Edgar *et al.* Open MPI: Goals, concept, and design of a next generation MPI implementation, p. 97–104, 2004.

GAN, Yu; DELIMITROU, Christina. The architectural implications of cloud microservices. **IEEE Computer Architecture Letters**, IEEE, v. 17, n. 2, p. 155–158, 2018.

GIMENEZ, Jose Miguel; FERNANDEZ, Javier David; MARTINEZ, Miguel Angel. A MapReduce-based Approach to Scale Big Semantic Data Compression with HDT. **IEEE Latin America Transactions**, IEEE, v. 15, n. 7, p. 1270–1277, 2017.

GOOGLE. **Cap'n Proto: Cerealization protocol**. 2022. Disponível em: https://google.github.io/flatbuffers/.

GOOGLE. **Protocol buffers | google developers**. 2022. Disponível em: https://developers.google.com/protocol-buffers/.

GOOGLE. **Snappy - A fast compressor/decompressor**. Accessed: 2023-06-15. 2011. Disponível em: https://google.github.io/snappy/.

IBM. **MQTT**. Accessed: 2023-06-15. 1999. Disponível em: https://mqtt.org/.

IETF, Internet Engineering Task Force. **RFC 1951 - DEFLATE Compressed Data Format Specification version 1.3**. Internet, mar. 1996. Accessed: 2023-06-15.

ITU TELECOMMUNICATION STANDARDIZATION SECTOR. Message handling systems: presentation transfer syntax and notation. Malaga-Torremolinos, Spain, 1984.

JIA, Yangqing; SHELHAMER, Evan; DONAHUE, Jeff; KARAYEV, Sergey; LONG, Jonathan; GIRSHICK, Ross; GUADARRAMA, Sergio; DARRELL, Trevor. Caffe: Convolutional architecture for fast feature embedding, p. 675–678, 2014.

KRIJNEN, Thomas; BEETZ, Jakob. An IFC schema extension and binary serialization format to efficiently integrate point cloud data into building models. **Advanced Engineering Informatics**, Elsevier, v. 33, p. 473–490, 2017.

MARTIN, Robert C. **Clean code: a handbook of agile software craftsmanship**. [*S.l.*]: Pearson Education, 2009.

MARTÍNEZ-PRIETO, Miguel A; ARIAS GALLEGO, Mario; FERNÁNDEZ, Javier D. Exchange and consumption of huge RDF data, p. 437–452, 2012.

MICROSOFT. **Bond Compiler 0.12.0.1**. 2018. Disponível em: `https://microsoft.github.io/bond/manual/compiler.html`.

MIZRAHI, T; FABINI, J; MORTON, A. **Guidelines for Defining Packet Timestamps**. Internet, 2020.

MIZRAHI, T; FABINI, J; MORTON, A. RFC 8877 Guidelines for Defining Packet Timestamps, 2020.

MONGODB. **BSON Specification**. Accessed: 2023-06-15. 2009. Disponível em: `https://bsonspec.org`.

MOONEY, P; MINGHINI, M. Geospatial Data Exchange Using Binary Data Serialization Approaches. **International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences**, v. 48, p. 4, 2022.

O'HARA, John. **Advanced Message Queuing Protocol (AMQP)**. Accessed: 2023-06-15. 2013. Disponível em: `https://www.amqp.org/`.

OLEG, Iakushkin; SEVOSTYANOV, Ruslan; DEGTYAREV, Alexander; KARPIY, PE; KUZEVANOVA, EG; KITAEVA, AA; SERGIEV, SA. Position tracking in 3D space based on a data of a single camera, p. 772–781, 2019.

OLINER, Adam J; IYER, Anand P; STOICA, Ion; LAGERSPETZ, Eemil; TARKOMA, Sasu. Carat: Collaborative energy diagnosis for mobile devices, p. 1–14, 2013.

OORTMERSSEN, Wouter van. **FlexBuffers**. Documentation. 2017. Disponível em: `https://google.github.io/flatbuffers/flexbuffers.html`.

PENG, Shangfu; SANKARANARAYANAN, Jagan; SAMET, Hanan. DOS: A spatial system offering extremely high-throughput road distance computations, p. 199–208, 2018.

POPIĆ, Srđan; PEZER, Dražen; MRAZOVAC, Bojan; TESLIĆ, Nikola. Performance evaluation of using Protocol Buffers in the Internet of Things communication, p. 261–265, 2016.

PROOS, Daniel Persson; CARLSSON, Niklas. Performance comparison of messaging protocols and serialization formats for digital twins in IoV, p. 10–18, 2020.

RABBITMQ. **RabbitMQ: Messaging that just works**. 2007. Disponível em: `https://www.rabbitmq.com/`.

RAJLICH, Václav. Software evolution and maintenance, p. 133–144, 2014.

RODDICK, John F. A survey of schema versioning issues for database systems. **Information and Software Technology**, Elsevier, v. 37, n. 7, p. 383–393, 1995.

SAYAR, Imen; BARTEL, Alexandre; BODDEN, Eric; LE TRAON, Yves. An In-depth Study of Java Deserialization Remote-Code Execution Exploits and Vulnerabilities. **ACM Transactions on Software Engineering and Methodology**, ACM New York, NY, 2022.

SEWARD, Julian. **bzip2 - a block-sorting file compressor**. Accessed: 2023-06-15. 1996. Disponível em: `https://sourceware.org/bzip2/`.

TECHNOLOGY, Massachusetts Institute of. **MIT License**. Accessed on 2023-06-15. 1989. Disponível em: `https://opensource.org/licenses/MIT`.

THE LINUX KERNEL. **netem**. Software. 2003. Disponível em: `https://wiki.linuxfoundation.org/networking/netem`.

THE REGENTS OF THE UNIVERSITY OF CALIFORNIA. **BSD 3-Clause License**. License. 1999. Disponível em: `https://opensource.org/licenses/BSD-3-Clause`.

TUKAANI PROJECT. **XZ Utils - Compression Utilities**. Accessed: 2023-06-15. 2009. Disponível em: `https://tukaani.org/xz/`.

UK, Borey; KONAM, David; PASSOT, Clément; ERDELJ, Milan; NATALIZIO, Enrico. Implementing a system architecture for data and multimedia transmission in a multi-UAV system, p. 246–257, 2018.

USHAKOV, Yury A; POLEZHAEV, Petr N; SHUKHMAN, Aleksandr E; USHAKOVA, Margarita V; NADEZHDA, MV. Split neural networks for mobile devices, p. 420–425, 2018.

VANURA, Jan; KRIZ, Pavel. Perfomance evaluation of Java, JavaScript and PHP serialization libraries for XML, JSON and binary formats, p. 166–175, 2018.

VARDA, Kenton. **Cap'n Proto: Cerealization protocol**. 2022. Disponível em: `https://capnproto.org/`.

VERNON, Vaughn. **Implementing domain-driven design**. [*S.l.*]: Addison-Wesley, 2013.

VIOTTI, Juan Cruz; KINDERKHEDIA, Mital. A Benchmark of JSON-compatible Binary Serialization Specifications. **arXiv preprint arXiv:2201.03051**, 2022.

VIOTTI, Juan Cruz; KINDERKHEDIA, Mital. A survey of JSON-compatible binary serialization specifications. **arXiv preprint arXiv:2201.02089**, 2022.

WEERASINGHE, LDSB; PERERA, Indika. An exploratory evaluation of replacing ESB with microservices in service-oriented architecture. v. 4, p. 137–144, 2021.

WEICHHART, Georg; PANETTO, Hervé; MOLINA, Arturo. Interoperability in the cyber-physical manufacturing enterprise. **Annual Reviews in Control**, Elsevier, v. 51, p. 346–356, 2021.

WOLNIKOWSKI, Adam; IBANEZ, Stephen; STONE, Jonathan; KIM, Changhoon; MANOHAR, Rajit; SOULÉ, Robert. Zerializer: Towards zero-copy serialization, p. 206–212, 2021.

XIE, Tao; QIN, Xiao. Performance evaluation of a new scheduling algorithm for distributed systems with security heterogeneity. **Journal of Parallel and Distributed Computing**, Elsevier, v. 67, n. 10, p. 1067–1081, 2007.

APÊNDICE A – CÓDIGO DO PROJETO

O código fonte deste projeto pode ser obtido em:

- `https://github.com/mudouasenha/serialization-benchmark.git`

ANEXO A – ARTIGO DO PROJETO

Avaliação de estratégias de serialização para comunicação em sistemas distribuídos

Matheus Miranda Torres Gomes[1] , Odorico Machado Mendizabal[1]

[1]Universidade Federal de Santa Catarina

[1]Departamento de Informática e Estatística
Universidade Federal de Santa Catarina (UFSC) – Florianópolis, SC – Brazil

matheusgomesmtg1@gmail.com, odorico.mendizabal@ufsc.br

Abstract. *Distributed systems are essential for processing and distributing information in a technologically advanced society. Designing a distributed system requires complex considerations, especially in relation to communication efficiency. Choosing the right communication format is crucial, as it affects performance, costs and service availability. This work aims to carry out an exploratory analysis of the serialization libraries Newtonsoft.Json, MessagePack, Apache Avro, Apache Thrift, Protocol Buffers, FlatBuffers, and Cap'n Proto, for data transport in distributed systems in the scope of \textit{microbenchmark} and communication between services in the client-server model, building a prototype in the C# language that represents a data-intensive traffic system. Performance tests explore their behaviour with large volumes of serialization, sequentially and concurrently, to determine their efficiency in different scenarios, as well as an analysis of the implementation aspects and qualitative analysis of the functionalities of each one. The metrics collected included processing time, compression rate and system throughput, as well as qualitative metrics such as language coverage and interoperability. Among the libraries evaluated, FlatBuffers, MessagePack and Protocol Buffers performed efficiently in all scenarios, while Apache Avro and Thrift had slower times. FlatBuffers showed deserialization times and very low RAM consumption, even when working with large volumes of data. Newtonsoft.Json, despite being a textual serialization, showed a significant performance gain by using concurrency, as did other libraries. In the distributed environment, Newtonsoft.Json, with a lower load of concurrent clients, performed well, Apache Avro had the highest serialization throughput, while Protocol Buffers had the lowest average serialization throughput observed.*

Resumo. *Sistemas distribuídos são essenciais para o processamento e distribuição de informações em uma sociedade tecnologicamente avançada. Projetar um sistema distribuído requer considerações complexas, especialmente em relação à eficiência da comunicação. Escolher o formato de comunicação correto é crucial, pois afeta o desempenho, custos e disponibilidade do serviço. Este trabalho se propõe a realizar uma análise exploratória das bibliotecas de serialização Newtonsoft.Json, MessagePack, Apache Avro, Apache Thrift, Protocol Buffers, FlatBuffers, e Cap'n Proto, para transporte de dados em sistemas distribuídos no escopo de \textit{microbenchmark} e comunicação entre serviços no modelo cliente-servidor, construindo um protótipo na linguagem C# que represente um sistema de tráfego intensivo de dados. Com testes de desempenho, é explorado o comportamento das mesmas com grandes volume de serialização, de forma sequencial e concorrente, para determinar a eficiência em diversos cenários, além de uma análise quanto aspectos de implementação e de forma qualitativa sobre as funcionalidades de cada uma. As métricas coletadas envolveram, tempo de processamento, taxa de compressão e vazão do sistema, assim como*

métricas qualitativas, como abrangência em linguagens e interoperabilidade. Dentre as bibliotecas avaliadas, FlatBuffers, MessagePack e Protocol Buffers desempenharam de forma eficiente em todos os cenários, enquanto Apache Avro e Thrift obtiveram tempos mais lentos. FlatBuffers apresentou tempos de desserialização e um consumo muito baixo de memória RAM, mesmo trabalhando com grandes volumes de dados. Newtonsoft.Json, apesar de ser uma serialização textual, apresentou um importante ganho de desempenho fazendo uso da concorrência, assim como outras bibliotecas. No âmbito distribuído, Newtonsoft.Json, com carga menor de clientes simultâneos, apresentou bom desempenho, Apache Avro teve maior vazão na serialização, enquanto Protocol Buffers teve menor vazão média na serialização observada.

1. Introdução

No contexto da acelerada evolução tecnológica impulsionada pela globalização, os sistemas atuais enfrentam crescente pressão por maior velocidade no processamento e na transferência de dados para atender às demandas da sociedade. Sistemas distribuídos desempenham um papel crucial ao possibilitar a distribuição de dados e processamento, especialmente em cenários de grande volume de dados, como nas redes sociais. A serialização e desserialização de dados impactam diretamente a eficiência e a velocidade dos sistemas. A interoperabilidade também é uma característica essencial em sistemas distribuídos, os quais necessitam de formatos de dados agnósticos e bibliotecas de serialização abrangentes para lidar com a heterogeneidade inerente a esses ambientes.

2. Motivação

No intuito de avaliar o desempenho das bibliotecas de serialização e comunicação distribuída, foram conduzidos testes de *microbenchmark* e implementado um protótipo de comunicação cliente-servidor usando a linguagem C#. A análise abrangeu características de implementação de cada serializador no contexto do protótipo, proporcionando uma avaliação qualitativa e quantitativa das bibliotecas de serialização. Este estudo destaca a importância da eficiência na serialização e desserialização de dados para otimizar a vazão do sistema distribuído, especialmente diante da diversidade de linguagens e arquiteturas presentes nesse cenário complexo.

3. Objetivos

- Elencar, por meio de estudo comparativo, as bibliotecas de serialização com maior destaque para os cenários avaliados;
- Gerar uma avaliação comparativa de desempenho das bibliotecas selecionadas por meio de testes em ambiente de *microbenchmark* e em protótipo local com comunicação distribuída no modelo cliente-servidor;
- Analisar aspectos qualitativos sobre cada experiência de implementação;
- Identificar e destacar as estratégias e bibliotecas de serialização mais eficientes e adequadas para o desenvolvimento de sistemas distribuídos na arquitetura cliente-servidor, com base nas conclusões e métricas obtidas durante a análise comparativa.

4. Fundamentação Teórica

A serialização e desserialização são processos essenciais para converter dados em um formato legível por máquinas, tornando-os transferíveis e utilizáveis por diferentes softwares. Essas técnicas permitem codificar ou traduzir dados em formatos de arquivo neutros, que podem ser recuperados ou utilizados por outras aplicações [Adibfar *et. al.*, 2021]. Viotti *et al.* (2022) descreve a serialização como o processo de tradução de uma estrutura de dados em uma sequência de bits para fins de armazenamento ou transmissão, com a possibilidade de reconstruir a estrutura original por meio do processo de desserialização.

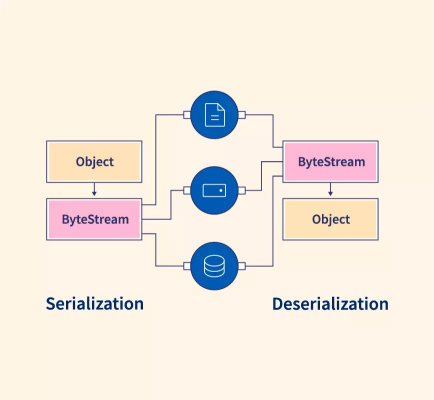


Figura 1. Processo de serialização e desserialização. Disponível em: https://www.scaler.com/topics/java/serialization-and-deserialization/ (Acesso em: 05 de Dezembro de 2023)

As bibliotecas de serialização fornecem a tradução bidirecional entre estruturas de dados e sequências de bits, sendo categorizadas de acordo com a forma como a informação é representada nesta sequência. Há uma grande variedade de formatos de serialização, diferenciando-se em termos de serem ou não orientados a esquema, textual ou binária, e no suporte a coleções e outros tipos de dados.

Os formatos *Extensible Markup Language* (XML) [Bray *et. al.*, 2000] e *Javascript Object Notation* (JSON) [Bray, 2014] são exemplos de formatos de serialização textual amplamente utilizados para a transferência de informações na Web. JSON, em particular, tornou-se uma alternativa popular ao XML devido à sua legibilidade e facilidade de uso. Ambos os formatos desempenham um papel significativo na representação de dados atualmente. No entanto, em detrimento da legibilidade, eles podem sacrificar a eficiência espacial e a rapidez na serialização.

4.1. Serialização Orientada e não Orientada a Esquema

A serialização de dados pode ser categorizada em: (i) orientada a esquema; (ii) não orientada a esquema, dependendo do uso ou não de uma definição formal de estrutura de dados, denominada esquema. Assim, bibliotecas de serialização podem adotar ou não uma definição de esquema para facilitar codificação e decodificação, conforme desejado. A serialização não orientada a esquema possibilita codificar e decodificar dados sem conhecimento prévio da estrutura, sendo autodescritiva, pois a sequência de bits contém

informações suficientes para interpretação. Exemplos incluem JSON [Bray, 2014] e MessagePack [Furuhashi, 2008]. Já a serialização orientada a esquema evita inserir informações sobre a estrutura nos bits resultantes, permitindo eficiência de espaço. Para aplicações que exigem eficiência na transmissão de dados pela rede, formatos orientados a esquema, como mencionado por [Popič *et. al.*, 2016], são preferíveis.

4.2. Serialização binária e baseada em texto

A serialização pode ser binária ou textual, dependendo se a cadeia de bits resultante corresponde a caracteres em codificação textual, como *American Standard Code for Information Interchange* (ASCII) ou *8-bit Unicode Transformation Format* (UTF-8). A serialização textual é legível e amigável ao humano, respeitando limites de codificação e sendo suportada por várias ferramentas. Por outro lado, a serialização binária permite estruturas de codificação mais eficientes, mas requer ferramentas especializadas para decodificação, limitando as opções de leitura e representação.

5. Trabalhos Correlatos

Os estudos de Popič *et. al.* (2016) e Proos *et. al.* (2020) investigam o desempenho da comunicação em ambientes de *Internet of Vehicles* (IoV). [Popič *et. al.*, 2016] analisa o uso de *Binary JSON* (BSON) e Protocol Buffers como alternativas para a serialização em comparação com um formato proprietário, observando que o formato proprietário supera JSON em compressão, com Protocol Buffers sendo a segunda opção mais eficiente. BSON não se destacou significativamente. No estudo de [Proos *et. al.*, 2020], os protocolos de troca de mensagens *Message Queuing Telemetry Transport* (MQTT), *Advanced Message Queuing Protocol* (AMQP) e *Constrained Application Protocol* (CoAP), e os formatos de serialização Protocol Buffers e FlatBuffers são avaliados na comunicação veículo-para-nuvem via *Wireless Fidelity* (Wi-Fi) e redes móveis. Protocol Buffers mostra melhor tempo de serialização e desserialização conjuntos, e distribuição de tamanho de mensagem menor, enquanto FlatBuffers tem vantagem em consumo de memória mais baixo e estável. Ambos os estudos contribuem para a compreensão do desempenho dessas tecnologias em contextos específicos da IoV.

A dissertação de Casey *et. al.* (2022) investiga a comunicação eficiente em um ambiente de *High Processing Computing* (HPC), específico para cenários de alto processamento. O autor avalia o desempenho das bibliotecas de serialização Protocol Buffers, FlatBuffers e MessagePack em comparação com a serialização padrão, no contexto de HPC, para a troca de mensagens entre nós em um sistema distribuído de análise de interação de proteínas em organismos. O *Message Passing Interface* (MPI) é usado para a troca de mensagens e coordenação de nós no sistema, implementado em C++. Os aspectos avaliados incluem tempo de execução, pegada de memória, utilização de hardware, tamanho do código gerado e métricas qualitativas de utilização das bibliotecas.

De modo geral, o autor conclui que o MessagePack teve um desempenho satisfatório em todas as métricas nos cenários analisados. Quanto à desserialização, o FlatBuffers mostrou eficiência, apesar de ser mais lento na serialização.

6. Protótipos

O sistema contou com um conjunto de objetos que visa simular dados de um site de transmissão de vídeos, com os objetos *Channel*, *VideoInfo*, *SocialInfo* e *Video*. Para a serialização e desserialização por cada formato, foi criada uma interface genérica *ISerializer*, que permite a serialização e desserialização de forma transparente, dentro do possível, à classe que a for utilizar.

6.1. Microbenchmark

O protótipo utilizado contou com o apoio da biblioteca BenchmarkDotNet, uma ferramenta popular para testes de desempenho em .NET. Foram criados perfis de teste para serializações individuais, seriais e concorrentes, de modo a identificar o comportamento de cada serializador em cenários distintos.

6.2. Comunicação em HTTP

Para comunicação em *Hypertext Transfer Protocol* (HTTP), foram utilizadas rotas de controlador para acionamento dos endpoints, estes que contabilizam a vazão por segundo. Para a geração de carga distribuída, foi utilizada a biblioteca Grafana k6, que permite a instanciação de usuários virtuais para geração de carga distribuída através de perfis de carga. Para o cliente, foi instanciado um sistema que mede requisições individuais, em determinado intervalo e período de tempo.

7. Avaliação Quantitativa

7.1. Serialização Individual

Visa-se obter informações quanto à rapidez e taxa de compressão de cada serializador considerando somente 1 objeto por vez, em cenário que exige pouco poder computacional. Utilizou-se como base de comparação o serializador BinaryFormatter.

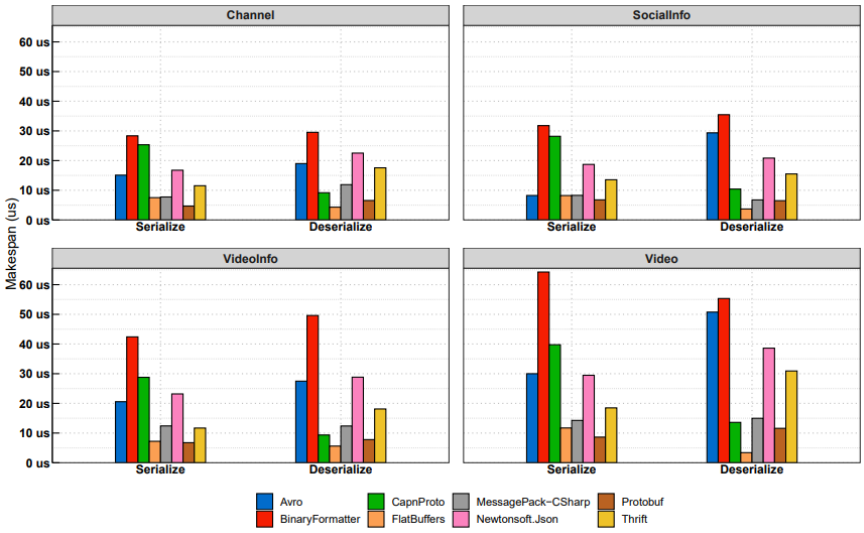


Figure 2. *Makespan* de serialização e desserialização dos objetos analisados.

Dado o *makespan* de serialização e desserialização, observou-se que FlatBuffers, Cap'n Proto e Protocol Buffers apresentaram serialização e desserialização eficientes. Enquanto isso, Apache Avro apresentou serialização mais eficiente do que desserialização, e o inverso para Cap'n Proto. Newtonsoft.Json, Apache Thrift, Apache Avro e Cap'n Proto apresentaram baixo desempenho para um ou mais cenários.

7.2. Serialização Serial

Nesta seção de teste são trabalhados cenários em que o fator a ser observado é o comportamento do serializador com maior volume de objetos, para medir a eficiência de serialização em cenários com maior exigência de recursos. **Número de mensagens:** 128.000, 256.000, 384.000, 512.000, 640.000, 768.000, 896.000, 1.024.000, 1.152.000 e 1.280.000.

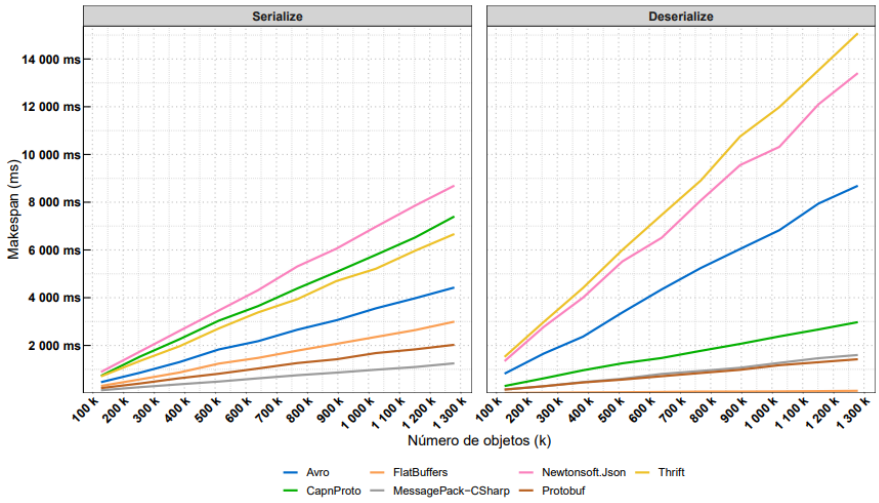


Figure 3. *Makespan* para serializações seriais do objeto *Video*.

Observa-se que MessagePack, FlatBuffers e Protocol Buffers possuem bons tempos de serialização e desserialização, com destaques em MessagePack na serialização e FlatBuffers na desserialização. Como destaques negativos, estão Newtonsoft.Json e Apache Thrift, além de Apache Avro na desserialização e Cap'n Proto na serialização.

7.3. Serialização Concorrente

Alternam-se os parâmetros de quantidade de objetos e *threads* que disputarão a serialização. Aqui é observado como o serializador trabalha em cenários de exclusão mútua quando múltiplos nós estão disputando o serviço. Serão instanciadas *threads* para a serialização e desserialização do conjunto de mensagens, com o uso de concorrência. **Número de mensagens:** 1.280.000. **Número de *threads*:** 2, 4, 8 e 16. As porcentagens se referem ao makespan proporcional em relação à execução em uma única *thread*.

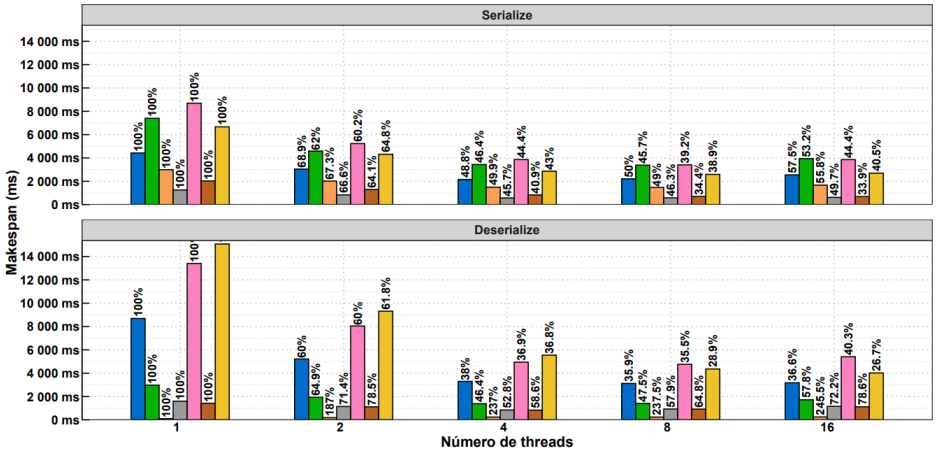


Figure 4. *Makespan* para serializações concorrentes do objeto *Video*.

Observou-se que todos os serializadores fizeram bom uso de concorrência em múltiplas threads, tendo variações em seu número de threads ótimo. Newtonsoft.Json e Apache Thrift obtiveram destaque na extração de desempenho em múltiplas *threads*, e outros serializadores já eficientes, como MessagePack e Protocol Buffers, conseguiram também diminuir seu *makespan*.

7.4. Serialização em HTTP

A realização deste conjunto de testes buscou simular um sistema que permite comunicação com diversos serviços ou clientes, e deve ser capaz de atender a múltiplas requisições simultaneamente, de forma rápida e eficiente. Neste cenário são avaliadas métricas em volta da vazão do servidor e a latência percebida no cliente, ambas com variação de carga em número de clientes simultâneos enviando requisições de serialização ou desserialização. Cada cenário de avaliação foi agrupado por método, número de *Virtual Users* (VUs) e serializadores, tendo 60s de tempo de coleta de resultados, executados em um único host.

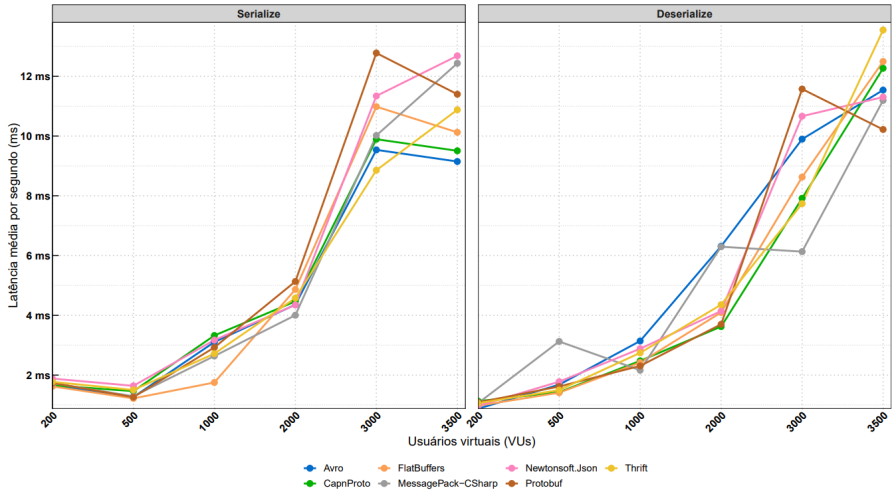


Figure 5. Latência observada no cliente para o objeto Video.

Percebeu-se que a serialização para todos os formatos apresentou uma curva de latência semelhante para até os 2.000 VUs. Para 3.000, foi observado aumento na latência média com uma curva acentuada de todos os serializadores, em especial para Protocol Buffers, Newtonsoft.Json e FlatBuffers. Na desserialização, foi possível perceber maior latência média por Apache Avro para até 2.000 clientes, o que corrobora com o baixo desempenho na desserialização percebido nas avaliações de *microbenchmark*.

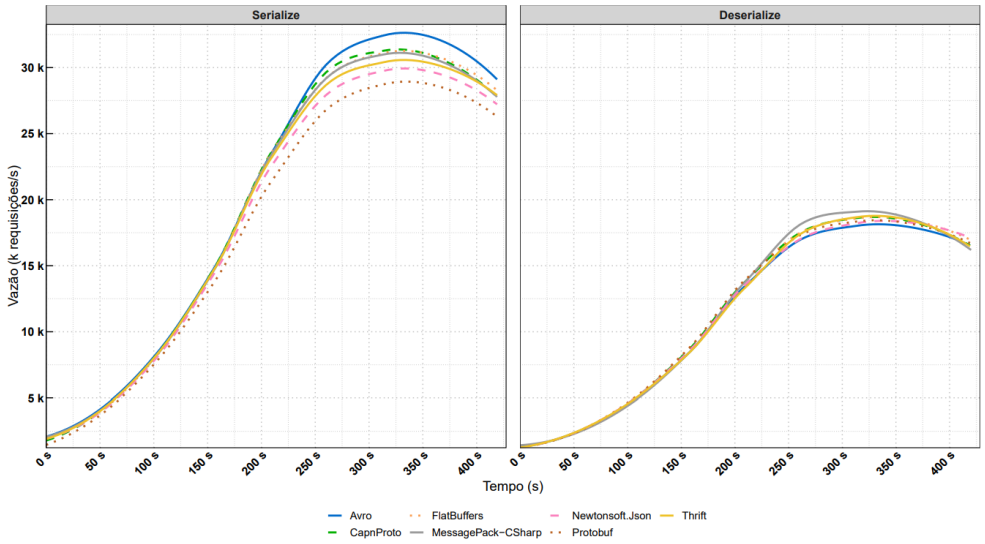


Figure 6. Vazão média no servidor ao longo do tempo, para o objeto *Video*.

A Figura 6 demonstra a vazão observada no cliente durante a execução do perfil de carga distribuída, indicando que a implementação atual tem vazão com uma média de aproximadamente 30.000 requisições por segundo, enquanto a desserialização apresenta uma média de 18.000 requisições por segundo.

8. Conclusão

O objetivo principal deste trabalho foi conduzir um estudo abrangente sobre bibliotecas de serialização comumente empregadas em sistemas distribuídos de alta vazão na linguagem C#. As bibliotecas analisadas incluíram MessagePack, Apache Avro, Apache Thrift, Protocol Buffers, FlatBuffers e Cap'n Proto. Como base de comparação, foram utilizados os serializadores Newtonsoft.Json, formato de serialização textual em JSON usado amplamente em projetos .NET, e BinaryFormatter, serialização binária depreciada, utilizado para extração dos tamanhos em memória dos objetos em formato não serializado.

Como principais contribuições deste trabalho, foi possível agregar diversas informações sobre bibliotecas de serialização binárias disponíveis para a linguagem C#, elencando suas funcionalidades e características disponíveis. Quanto ao âmbito quantitativo, realizou-se a agregação dos resultados das bibliotecas analisadas em comparação à biblioteca de serialização textual Newtonsoft.Json, tanto com a serialização individual quanto múltipla. Além disso, foram extraídas métricas de desempenho de cada biblioteca com o uso de concorrência em múltiplas *threads*,

buscando apresentar o ganho proporcional de desempenho ao usar uma abordagem que busque melhor utilização das *threads* disponíveis. Ao mesmo tempo, foram apresentadas características empíricas que visam enriquecer a análise e escolha de um formato de serialização de acordo com a necessidade observada, levando em consideração as funcionalidades disponíveis, assim como possíveis dificuldades de manutenção e implementação. Dado um ambiente com comunicação em HTTP, foi possível observar a vazão e latência média na serialização e desserialização de um sistema com recursos computacionais das características apresentadas, implementado em C#.

Dentre os resultados obtidos, foi possível observar um ganho de desempenho considerável por grande parte dos serializadores com o uso de concorrência, em relação à serialização em série. MessagePack, FlatBuffers e Protocol Buffers apresentaram bom desempenho na maior parte dos cenários avaliados, em microbenchmark, com destaque para FlatBuffers na desserialização, e MessagePack pela simplicidade no desenvolvimento e desempenho geral. Nos testes distribuídos, a realização da análise em um único hospedeiro fez com que os resultados dos testes não representassem um ambiente real de comunicação via rede, passando por todas as camadas e ativando seus respectivos mecanismos, diminuindo a sua confiabilidade geral. Reitera-se que a implementação realizada pode ter influenciado os testes distribuídos, e sugere-se realizar testes mais robustos e em ambiente mais adequado para analisar o desempenho neste cenário.

9. Trabalhos Futuros

Propõe-se realizar testes em um *testbed* para garantir confiabilidade na avaliação do comportamento de bibliotecas em ambientes distribuídos. Este ambiente irá replicar condições reais de comunicação, ativando controle de fluxo e ajustando parâmetros de qualidade de serviço. Testes com diferentes perfis de carga, como *spike*, *breakpoint* e *soak*, são indicados para avaliar a resposta de cada serializador a diferentes características de carga. A eficiência da comunicação com serialização binária deve ser testada com outros protocolos, tanto na camada de aplicação quanto na de transporte, incluindo avaliação das capacidades de tráfego e *streams* de dados.

Estratégias eficientes de serialização podem ser aplicadas em soluções existentes, permitindo medir o ganho de desempenho real em aplicações de larga escala em produção. As implementações podem ser adaptadas para cenários específicos, como sistemas de alta vazão, protocolos de fusão, replicação de dados ou sistemas que utilizam comunicação por microsserviços.

References

ADIBFAR, Alireza; COSTIN, Aaron M. Review of Data Serialization Challenges and Validation Methods for Improving Interoperability, p. 522–529, 2021.

VIOTTI, Juan Cruz; KINDERKHEDIA, Mital. A survey of JSON-compatible binary serialization specifications. arXiv preprint arXiv:2201.02089, 2022.

BRAY, Tim; PAOLI, Jean; SPERBERG-MCQUEEN, C Michael; MALER, Eve; YERGEAU, François; COWAN, John. Extensible markup language (XML) 1.0. W3C recommendation October, Internet, 2000.

BRAY, T. The JavaScript Object Notation (JSON) Data Interchange Format. Internet, mar. 2014. http://www.rfc-editor.org/rfc/rfc7159.txt. Disponível em: http://www.rfc-editor.org/rfc/rfc7159.txt

FURUHASHI, Sadayuki. MessagePack: It's like JSON. but fast and small. 2008. Disponível em: https://msgpack.org/.

POPIČ, Srđan; PEZER, Dražen; MRAZOVAC, Bojan; TESLIČ, Nikola. Performance evaluation of using Protocol Buffers in the Internet of Things communication, p. 261–265, 2016.

CASEY, Allen et al. Performance of Serialization Libraries in a High Performance Computing Environment. 2022. Tese (Doutorado).

PROOS, Daniel Persson; CARLSSON, Niklas. Performance comparison of messaging protocols and serialization formats for digital twins in IoV, p. 10–18, 2020.