INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Avaliação da Framework gRPC

JOSÉ ANTÓNIO MOREIRA RODRIGUES
Junho de 2022

POLITÉCNICO
DO PORTO

Instituto Superior de Engenharia do Porto

Avaliação da Framework gRPC

José António Moreira Rodrigues

Dissertação para obtenção do Grau de Mestre em Engenharia Informática, Área de Especialização em Engenharia de Software

Orientador: Alexandre Bragança

Júri:

Presidente:

Vogais:

Porto, junho 2022

Resumo

A arquitetura e construção de software pode ser descrita como um processo evolutivo que, através de constantes iterações, conduz ao desenvolvimento de melhores tecnologias, abordagens, processos, entre outros. Esta evolução nasce devido à necessidade de mudança – necessidade esta que pode surgir devido a novas tecnologias, métodos, objetivos ou outra qualquer alteração no ambiente.

A arquitetura baseada em microsserviços pode ser vista como um dos resultados deste processo evolutivo – trata-se de uma abordagem à construção de aplicações complexas, onde a aplicação é dividida num conjunto de elementos de menor tamanho, autónomos, e que comunicam entre si. Embora a adoção desta abordagem traga consigo uma série de vantagens que lhe são associadas, – agilidade de desenvolvimento, eficiência de recursos e manutenibilidade, para nomear algumas – a comunicação entre os vários serviços continua a mostrar-se relativamente desafiante.

gRPC, desenvolvido pela Google em 2015, é uma *framework* RPC que pode servir de alternativa a outras abordagens à comunicação entre microsserviços (como é o caso do estilo REST) que constituem uma aplicação, e que foi introduzida com o objetivo de promover o desenvolvimento de soluções baseadas em microsserviços e simplificar a troca de mensagens entre sistemas. Dada a juventude desta *framework* existe alguma dificuldade em compreender as ideias por trás da mesma e como esta pode ser adotada.

Este documento apresenta os conceitos que compõe esta *framework* e descreve, resumidamente, a abordagem REST – a principal abordagem ao desenvolvimento de comunicação (síncrona) entre microsserviços. Para além disso, é também proposto e apresentado o desenvolvimento de aplicações protótipo com o objetivo estudar a adoção destas abordagens.

O documento termina com uma avaliação ao desempenho dos protótipos desenvolvidos e com uma conclusão que tem como objetivo ajudar a perceber como gRPC se compara com a adoção de REST quando aplicada a diversos cenários de comunicação entre microsserviços.

Palavras-chave: gRPC, REST, Microsserviços, Comunicação

Abstract

Software architecture and construction can be described as an evolutionary process that, through constant iterations, leads to the development of better technologies, approaches, processes, among others. This evolution arises due to the need for change - a need that can arise due to new technologies, methods, goals, or any other change in the environment.

The microservices-based architecture can be seen as one of the results of this evolutionary process - it's an approach to building complex applications, where the application is divided into a set of smaller elements, that are autonomous, and that communicate with each other. While adopting this approach brings with it several advantages associated with it - development agility, resource efficiency, and maintainability, to name a few - communication between the various services still proves to be relatively challenging.

gRPC, developed by Google in 2015, is an RPC framework that can serve as an alternative to implementing communication between microservices that constitute an application, and that was introduced with the goal of promoting the development of microservices-based solutions and simplifying the exchange of messages between systems. Given the youth of this framework there is some difficulty in understanding the ideas behind it and how it can be adopted.

This paper presents the concepts that make up this framework, and briefly describes the REST approach - the main approach to the development of (synchronous) communication between microservices. In addition, the development of prototype applications is also proposed and presented in order to study the adoption of these approaches.

The document ends with an evaluation of the performance of the prototypes developed and with a conclusion that aims to help understand how gRPC compares to the adoption of REST when applied to several scenarios of communication between microservices.

Keywords: gRPC, REST, Microservices, Communication

Agradecimentos

Agradeço aos amigos, família e todos aqueles que de alguma forma contribuíram ou possibilitaram o trabalho desenvolvido ao longo deste documento.

Agradeço também ao professor Alexandre Bragança por toda a sua orientação e paciência ao longo do desenvolvimento da presente dissertação.

Índice

Lista de Figuras

Lista de Tabelas

Acrónimos e Símbolos

Lista de Acrónimos

AHP	Analytic Hierarchy Process
AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
CPU	Central Processing Unit
CRUD	Create, Read, Update, Delete
CSS	Cascading Style Sheets
DSRM	Design Science Research Methodology
gRPC	g[1] Remote Procedure Call
HATEOAS	Hypermedia as the Engine of Application State
HPACK	Algoritmo de compressão do cabeçalho
HTTP	Hypertext Transfer Protocol
HTTP/2	Segunda versão do protocolo HTTP
HTTPS	Hypertext Transfer Protocol Secure
IC	Índice de Consistência
IDL	Interface Definition Language
IoT	Internet of Things
IR	Índice Aleatório
JSON	JavaScript Object Notation
KB	Kilobyte
NCD	New Concept Development
NPD	New Product Development

[1] A letra 'g' tem um significado diferente em cada lançamento (por exemplo na versão 1.45 o 'g' significa *gravity*)

RC	Razão de Consistência
REST	REpresentational State Transfer
RPC	Remote Procedure Call
STOMP	Simple (or Streaming) Text Orientated Messaging Protocol
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UDP	User Datagram Protocol
URI	Uniform Resource Identifier
XML	Extensible Markup Language

1 Introdução

Este capítulo tem como objetivo contextualizar o problema abordado no decorrer do documento e apresentar o mesmo. Para isso é fornecida uma breve descrição do problema abordado, são sucintamente abordados os objetivos deste trabalho e apresentada a metodologia seguida. O capítulo termina expondo a estrutura do documento.

1.1 Contexto

O atual panorama de desenvolvimento de tecnologia está carregado de *buzzwords* que competem entre si com o objetivo de se tornarem a próxima grande tendência que irá dominar o futuro deste mercado. Termos como *Internet of Things (IoT)*, *multi-cloud*, análise preditiva, hiperautomação, *big data*, entre outros, representam alguma das áreas consideradas como de investimento essencial por parte de organizações que operem no mundo do desenvolvimento tecnológico.

Um dos termos que a cada dia apresenta maior presença no desenvolvimento de soluções, e que se vem tornando cada vez mais importante para o desenho de aplicações de software é o termo "microsserviços". Muito resumidamente, o termo microsserviços (ou arquitetura baseada em microsserviços) pode ser definido como uma abordagem à construção de uma aplicação como um conjunto de vários serviços mais pequenos, onde cada serviço implementa funcionalidades sobre um domínio ou capacidade empresarial específica, onde cada um é desenvolvido de forma autónoma e distribuído de forma independente, e onde existe comunicação entre os mesmos através de protocolos como HTTP/HTTPS, *WebSockets,* ou AMQP (Wenzel, Parente and Anil, 2021).

Apesar das várias vantagens associadas ao desenvolvimento de aplicações em forma de microsserviços (vantagens essas que serão exploradas neste documento), a adoção desta abordagem apresenta também algumas complicações, nomeadamente a comunicação entre os mesmos.

A tecnologia gRPC, desenvolvida em 2015 pela Google, apresenta-se como sendo uma "*framework* de *Remote Procedure Call* (RPC) de alto desempenho, *open source*, que funciona em qualquer ambiente", e que pode ser usada na implementação de comunicação entre serviços (Google, 2021). Com o desenvolvimento desta tecnologia, a Google procurou promover o desenvolvimento de soluções baseadas em microsserviços, simplificando a troca de mensagens entre sistemas, oferecendo interoperabilidade entre tecnologias, garantindo a sua viabilidade em CPUs e dispositivos com memória limitada, entre outros.

Embora o recurso à *framework* gRPC se apresente como uma possível solução para a comunicação entre microsserviços, existem outras opções que, atualmente, apresentam maior popularidade. De entre as alternativas, destaca-se o estilo REST que lidera, com uma margem significativa, na área do desenvolvimento de APIs ("State of APIs Developer Survey 2021 Report," 2021). A escolha da abordagem à comunicação entre microsserviços é um passo critico para qualquer projeto que adote este tipo de arquitetura.

1.2 Problema

Num sistema baseado em microsserviços, serviços relativamente pequenos que funcionam de forma independente e que oferecem funções distintas, são combinados de forma a apresentar funcionalidades e capacidades normalmente associadas a aplicações maiores (De *et al.*, 2016). Isto implica a existência de troca de informação entre os vários serviços de maneira a garantir que operações que envolvam mais do que um serviço possam ser executadas.

gRPC identifica-se como uma ferramenta que pode ser usada em cenários onde existe uma necessidade de transacionar informação entre serviços, suportando funcionalidades como transmissão bidirecional de eventos, bloqueio de chamadas assimétricas, interoperabilidade entre linguagens, entre outras (Indrasiri and Kuruppu, 2020). Apesar de gRPC apresentar funcionalidades relevantes para um desenvolvimento baseado em microsserviços, devido à sua juventude e ao relativamente reduzido nível de material sobre a mesma, é atualmente difícil perceber como é que a ferramenta realmente funciona, em que cenários esta pode ser útil, e como é que esta se compara com outras abordagens à comunicação de maior popularidade, nomeadamente o estilo arquitetural REST.

1.3 Objetivos

Este trabalho tem como foco principal explorar a *framework* gRPC, assim como estudar a aplicação desta tecnologia e de outras abordagens alternativas (nomeadamente o estilo REST) quando aplicadas à construção de uma aplicação baseada em microsserviços. Pretende-se também observar o desenvolvimento das soluções tendo em consideração alguns cenários de comunicação que acontecem com frequência dentro de um sistema deste género:

- Comunicação síncrona entre serviços

- Comunicação em ambiente poliglota
- Comunicação de baixa latência e alto rendimento
- Comunicação em ambientes com restrições de rede (banda larga disponível)
- Comunicação direta entre *frontend* e *backend*

Tanto as abordagens consideradas como os cenários de comunicação serão explorados com maior detalhe no capítulo 2.

De forma a obter uma compreensão de como as diferentes abordagens podem ser usadas para a implementação de comunicação entre microsserviços, e como estas de comparam entre si, este trabalho tem como intenção apresentar um paralelo entre a implementação recorrendo ao estilo REST e a implementação recorrendo a gRPC – este paralelo deve permitir ao leitor facilmente perceber o que as diferentes abordagens implicam.

Importa realçar que a adoção do estilo REST foi selecionado como principal alternativa a gRPC devido a dois fatores:

- A sua popularidade no desenvolvimento de software ("State of APIs Developer Survey 2021 Report," 2021)
- O facto de que possibilita a sua utilização em cenários de comunicação semelhantes (este ponto será explorado com maior profundidade no capítulo Contexto e Estado da Arte)

O trabalho não tem como objetivo definir, de entre as ferramentas comparadas, qual é indiscutivelmente a melhor ou qual deve ser definitivamente usada na construção de uma solução baseada em microsserviços.

Em função das diferentes particularidades que cada cenário pode apresentar, e devido à quase infinita possibilidade de cenários, é importante realçar que nem todos os cenários possíveis serão considerados na realização deste trabalho.

É também essencial destacar o facto de que nem todas as ferramentas/tecnologias/estilos que poderiam ser relevantes, serão consideradas durante o estudo. O número de possibilidades nesta área cresce a cada dia, tornando inviável a consideração de todas.

Finalmente, pretende-se confirmar a seguinte hipótese:

- **Hipótese 1 (H1):** A aplicação que recorre a gRPC para comunicação entre os serviços que a compõe apresenta desempenho superior à aplicação que adota o estilo REST

1.4 Metodologia de Trabalho

Dada a complexidade do trabalho, a quantidade e importância dos detalhes do mesmo, e a necessidade de cumprir os prazos estabelecidos para a unidade curricular, torna-se importante

adotar uma metodologia de trabalho que não só permita cumprir os objetivos delineados, como também acrescente alguma validade ao mesmo.

Uma metodologia de trabalho tem como principal objetivo auxiliar na definição do que se pretende fazer, definir que passos devem ser seguidos, e expor a estratégia geral do trabalho.

Este trabalho adota a *Design Science Research Methodology* (DSRM), uma metodologia normalmente aplicada para o "desenvolvimento e avaliação de artefactos destinados a resolver problemas identificados" (Peffers *et al.*, 2007).

Esta metodologia é composta pelas seguintes fases (Peffers *et al.*, 2007):

- **Identificação do problema e motivações:** Esta fase consiste na definição do problema a ser investigado e a justificação do valor da solução. Neste documento esta fase encontra-se na secção 1.2.
- **Definição dos objetivos da solução:** Devem ser definidos os objetivos da solução tendo por base o problema estabelecido e o conhecimento do que é possível e viável realizar. É possível observar esta fase na secção 1.3.
- **Desenho e desenvolvimento:** Aqui, a solução deve ser criada e desenvolvida. O desenho pode ser observado no capítulo 4.
- **Demonstração:** Esta fase envolve a demonstração da solução para resolver um ou mais exemplos do problema. Isto pode envolver a sua utilização em experiências, simulações, casos de estudo, ou qualquer outra atividade que se considere apropriada. A demonstração da solução é visível no capítulo 5, onde são apresentados os principais pontos relacionados com o desenvolvimento da solução.
- **Avaliação:** Durante esta fase deve ser observado e medido o quão bem a solução resolve o problema. Para isso foi estabelecida uma hipótese a estudar e foram definidos casos de teste de modo a confirmar (ou desmentir) a mesma. O resultado desta fase pode ser observado no capítulo 6. São também apresentadas algumas conclusões sobre os objetivos propostos nas secções 7.1 e 7.2.
- **Comunicação:** O problema deve ser comunicado juntamente com a sua importância, utilidade, rigor e eficácia ao público relevante. No caso deste trabalho, a fase de comunicação consiste no presente documento.

1.5 Contribuições

Dada a relativamente baixa popularidade que a *framework* gRPC apresenta atualmente no mercado de desenvolvimento de software ("State of APIs Developer Survey 2021 Report," 2021), existe alguma dificuldade em compreender como adotar esta tecnologia e onde a mesma pode ser usada.

Os resultados do trabalho ajudarão os leitores do mesmo a adquirir conhecimento sobre a *framework* gRPC, mais concretamente a sua história, os conceitos por trás da mesma e os potenciais benefícios da sua utilização para a comunicação em sistemas distribuídos. Com a

estudo de diferentes abordagens à comunicação quando aplicadas a cenários comuns em sistemas baseados em microsserviços, o trabalho pretende auxiliar na tomada de decisão de desenvolvedores que se deparem com cenários semelhantes aos estudados e que procuram suporte na decisão sobre que tecnologia/abordagem/estilo usar na construção da solução.

1.6 Estrutura do documento

O documento é composto por 7 capítulos.

O atual capítulo (1) tem como principal objetivo auxiliar o leitor na compreensão do trabalho realizado. Para isso é apresentado um pequeno enquadramento do trabalho, o problema, os objetivos definidos, o âmbito do trabalho e o plano definido para a realização do mesmo.

O capítulo 2 expõe o estado da arte, introduzindo conceitos considerados importantes para este trabalho, bem como outros trabalhos onde o tema abordado seja relevante na área em estudo.

De seguida, no capítulo 3, é realizada uma análise de valor ao projeto proposto. Este capítulo contém também a aplicação do método *Analytic Hierarchy Process* (AHP) que foi usado com o objetivo de selecionar a ideia que serve como base à construção da solução proposta.

Posteriormente, no capítulo 4, é apresentado o desenho proposto para a solução, assim como justificação das escolhas realizadas.

O capítulo 5 demonstra e analisa os detalhes do desenvolvimento da solução considerados mais relevantes.

O capítulo 6 contém o plano de avaliação da solução, onde são definidos os indicadores a usar para realizar a avaliação e o procedimento a seguir para a realização da mesma. Neste capítulo são também apresentados e analisados os resultados da execução do plano de avaliação.

Finalmente, no capítulo 7, são apresentadas as conclusões do trabalho através de uma breve discussão sobre os resultados e da exposição das observações realizadas em diferentes cenários relacionados ao problema. Este capítulo expõe também limitações do trabalho, possíveis desenvolvimentos futuros e uma reflexão final sobre gRPC.

2 Contexto e Estado da Arte

O presente capítulo tem como finalidade explorar os principais conceitos associados ao problema apresentado previamente. Para isso, o capítulo começa por explorar em que consiste uma arquitetura baseada em monólitos, e a sua evolução para uma arquitetura baseada em microsserviços apresentando algumas das inconveniências e vantagens de cada abordagem. De seguida, são apresentados e discutidos alguns dos principais desafios associados ao desenvolvimento de um sistema sustentado por microsserviços, explorando resumidamente algumas das alternativas comumente usadas no desenvolvimento da componente de comunicação em sistemas semelhantes ao discutido.

2.1 Dos monólitos aos microsserviços

Nos dias que correm, grande parte das organizações depende do funcionamento dos seus sistemas empresariais – sistemas complexos que tendem a lidar com um grande volume de dados e que permitem às organizações integrar e coordenar os processos necessários às suas operações diárias.

Devido à grande complexidade que estes sistemas normalmente apresentam, desenvolvedores que começam a lidar com um sistema semelhante ao descrito tendem a necessitar de meses para perceber a base de código existente antes mesmo de poderem começar a trabalhar no mesmo. Para além disso, alterações a um sistema destes ou novas funcionalidades tendem a ser longamente discutidas devido ao receio que a equipa de desenvolvimento tem de que possíveis consequências imprevistas e indesejadas surjam, tornando assim a evolução do sistema mais lenta. O ato de gerir um sistema destes é algo que requere um equilíbrio delicado entre perceber as necessidades do sistema, e como pequenas alterações podem afetar outros componentes do sistema.

De forma a lidar com este problema e com o objetivo de reduzir a complexidade de um sistema com as caraterísticas descritas, desenvolvedores têm optado por construir (ou migrar) os seus sistemas tendo como base uma arquitetura consistente em microsserviços.

Esta secção começa por discutir os desafios caraterísticos à adoção de uma arquitetura baseada em microsserviços. De seguida são a é apresentada a ideia por trás da arquitetura monolítica e da arquitetura baseada em microsserviços, explorando as vantagens e desvantagens inerentes à adoção de uma ou de outra.

2.1.1 Desafios inerentes à adoção de uma arquitetura baseada em monólitos

Um monólito pode ser brevemente definido como uma "aplicação de software cujos módulos não podem ser executados independentemente (Dragoni Nicola and Giallorenzo, 2017).

O estilo arquitetural baseado em monólitos é considerado um tipo de arquitetura tradicional, usado amplamente pela indústria para o desenho de soluções. Embora exista a possibilidade de construir uma aplicação monolítica compreendida de várias partes, tais como apresentação, lógica de negócio e acesso à base de dados, esta continuar a ser construída e implantada como uma única unidade (De *et al.*, 2016; Aroraa, Kale and Manish, 2017).

Apesar desta arquitetura ser uma opção viável para o desenvolvimento de aplicações mais pequenas, permitindo apresentar rapidamente uma solução de qualidade ao cliente final (Westeinde, 2019), a adoção desta arquitetura para a construção de soluções maiores e mais complexas obriga a que exista um maior esforço de orquestração por parte das equipas de desenvolvimento para concretizar qualquer alteração, por mais pequena que esta seja (De *et al.*, 2016).

De seguida são apresentados alguns dos problemas que se tornam mais evidentes à medida que uma aplicação que adote esta arquitetura cresce, especialmente no contexto de mudanças rápidas em que vivemos hoje.

Base de código muito extensa

Como já brevemente descrito anteriormente, uma grande base de código pode criar dificuldades a um programador que esteja no processo de se familiarizar com o código, especialmente quando o programador é um novo membro da equipa de desenvolvimento. Para além disso, pode existir maior dificuldade ao lidar com o código dada a possibilidade de o mesmo sobrecarregar as ferramentas usadas para desenvolvimento. Tudo isto contribui para uma menor produção por parte dos desenvolvedores e para o desencorajamento de tentativas de fazer alterações (De *et al.*, 2016; Aroraa, Kale and Manish, 2017).

Complexidade da base de código

Com o aumento da base de código existente, a complexidade do mesmo tende também a aumentar, enquanto a sua qualidade tende a diminuir. Decisões irreversíveis tomadas no passado que promovem o desenvolvimento de soluções menos ideais, dependências entre

secções de código e escrita de código menos eficiente por parte dos desenvolvedores são algumas ações que levam ao aumento da complexidade.

O aumento da complexidade tende a amplificar a quantidade de *bugs* e vulnerabilidades existentes (Aroraa et al., 2017; Dragoni Nicola and Giallorenzo, 2017).

Implantação complexa

Numa aplicação monolítica, mesmo a mais pequena mudança obriga a que toda a aplicação seja reconstruída, testada e implantada. Isto torna complicado para a equipa responsável pela implantação da aplicação manter uma entrega contínua, resultando numa menor frequência de implantações e numa resposta mais lenta às mudanças necessárias (De et al., 2016; Dragoni Nicola and Giallorenzo, 2017).

Alta dependência entre os vários módulos do sistema

Uma vez que as várias partes que compõe um monólito estão interligadas e apresentam dependências entre si, uma única mudança num monólito pode iniciar uma sequência de falhas em testes que não apresentam relações diretas entre si (Westeinde, 2019). A correção destas falhas exige muitas vezes que alterações adicionais sejam realizadas em diferentes partes do monólito. Para além disso, uma única mudança numa parte do monólito pode causar um erro ou vulnerabilidade noutra parte do monólito, que pode não ser identificado pelos testes existentes (Dragoni Nicola and Giallorenzo, 2017).

Escalabilidade

A escalabilidade de uma solução monolítica é um dos maiores desafios deste tipo de arquitetura. Habitualmente, a estratégia usada para responder ao incremento do número de pedidos realizados à aplicação é a de criar novas instâncias da mesma aplicação, dividindo a carga entre as mesmas. Contudo, por vezes o aumento do tráfego corresponde apenas à procura de funcionalidades de parte dos módulos do monólito. Estes casos são inconvenientes, visto que existe uma atribuição de recursos a um subconjunto de módulos que não viram aumento na sua utilização (Aroraa et al., 2017; De et al., 2016; Dragoni Nicola and Giallorenzo, 2017).

Incapacidade/Dificuldade de adoção de uma nova tecnologia

Ao seguir o modelo monolítico, devido aos desafios inerentes ao desenvolvimento de alterações, a adaptação a novas tecnologias ou a novas versões das tecnologias usadas para desenvolvimento torna-se difícil. Como resultado, uma aplicação que siga esta arquitetura usa, normalmente, durante o seu ciclo de vida as mesmas tecnologias, o que constitui um obstáculo para que a aplicação acompanhe as novas tendências de desenvolvimento, e o que representa um bloqueio tecnológico para os desenvolvedores, que se vêm obrigados a usar as mesmas linguagens e *frameworks* que foram usadas na conceção original da aplicação (De et al., 2016; Dragoni Nicola and Giallorenzo, 2017).

2.1.2 A adoção de uma arquitetura baseada em microsserviços

Ao longo dos anos, novas soluções para o desenvolvimento de software têm surgido na indústria. Como nenhuma solução é perfeita (cada uma apresenta as suas próprias vantagens e desvantagens), a capacidade de responder rapidamente a exigências inerentes ao desenvolvimento de software, tais como a escalabilidade, desempenho, e fácil implantação é cada vez mais importante para as empresas, principalmente tendo em conta o impacto que cada um destes aspetos pode ter no funcionamento do negócio (Aroraa, Kale and Manish, 2017).

A arquitetura baseada em microsserviços vem sendo usada pelos arquitetos dos grandes sistemas empresariais para auxiliar na proteção de um sistema contra os problemas mencionados anteriormente.

Microsserviços são pequenas aplicações autônomas, que têm apenas uma responsabilidade, e que podem ser implantadas, escalonadas e testadas de forma independente. Num sistema onde as funcionalidades foram decompostas em vários microsserviços, existe uma interação entre os mesmos que é feita usando protocolos como HTTP, AMQP e STOMP baseados em mensagens, ou TCP e UDP (De *et al.*, 2016; Larrucea *et al.*, 2018).

De seguida são apresentadas algumas das caraterísticas associadas à adoção desta arquitetura.

Independência entre serviços

Um sistema baseado nesta arquitetura, quando bem desenhado, apresenta microsserviços com limites claros, que podem ser implantados de forma independente e que são capazes de operar de maneira autónoma. Esta independência minimiza o risco de ocorrência de efeitos secundários devido a alterações, e aumenta a capacidade de testar um microsserviço isoladamente do resto do sistema (Dragoni Nicola and Giallorenzo, 2017).

Porém, para que a independência entre sistemas se mantenha, cada microsserviço deve cumprir e manter o seu contrato de API (Aroraa, Kale and Manish, 2017).

Escalonamento eficaz

Dada a natureza da arquitetura, e ao contrário de uma aplicação monolítica onde é necessário adicionar/remover instâncias da aplicação inteira, é possível aumentar apenas o número de instâncias dos microsserviços que sofram de maior carga, reduzindo assim a quantidade de recursos necessários (Aroraa et al., 2017; Dragoni Nicola and Giallorenzo, 2017).

Desenvolvimento tende a ser mais fácil

Como cada microsserviço é focado em apenas uma responsabilidade, a base de código tende a manter-se relativamente pequena, facilitando assim a sua aprendizagem e o ato de *debugging*, o que por sua vez contribui para a integração de um novo elemento na equipa de desenvolvimento. Para além disso, como o código de um microsserviço não depende do código presente noutros microsserviços, alterações tendem a ser mais facilmente implementadas,

dada a baixa probabilidade das mesmas provocarem erros em outras partes da aplicação (De et al., 2016; Dragoni Nicola and Giallorenzo, 2017).

Fácil implantação

Num sistema composto por microsserviços, cada microsserviço "vive" de maneira independente e isolada dos outros. Assim, é possível realizar a implantação de cada microsserviço um de cada vez facilitando o processo. Para além disso, a falha de um microsserviço não implica a falha de todo o sistema (Aroraa, Kale and Manish, 2017).

Outro ponto a favor desta abordagem, é que a alteração de um microsserviço não obriga a uma reimplantação de todo o sistema (Dragoni Nicola and Giallorenzo, 2017).

Maior liberdade de tecnologias

Seguindo esta abordagem, existe uma liberdade, por parte dos desenvolvedores, de escolher as ferramentas, *frameworks,* e pilha de tecnologias que sejam identificadas como as melhores opções para um determinado microsserviço, não existindo a necessidade de escolher tecnologias que englobem todas as funcionalidades esperadas do sistema (Aroraa et al., 2017; Dragoni Nicola and Giallorenzo, 2017).

Resposta ao mercado mais rápida

Como mencionado anteriormente, esta arquitetura tende a permitir desenvolver, atualizar e distribuir aplicações de forma mais rápida. Esta facto leva a que a organização apresente uma maior capacidade de responder às rápidas mudanças do mercado, garantindo que o seu produto segue as novas tendências e que o mesmo não perde espaço para os seus principais concorrentes (Aroraa, Kale and Manish, 2017).

2.2 Comunicação numa arquitetura baseada em microsserviços

Numa aplicação baseada numa arquitetura monolítica, a comunicação tende a ser simples - os componentes da aplicação invocam-se entre si através de métodos ao nível da linguagem ou através de chamadas a funções. Apesar desta abordagem apresentar vantagens em termos de desempenho, é comum, como já exposto anteriormente, resultar em código que está fortemente acoplado e que se torna difícil de manter, evoluir e escalar (de la Torre, Wagner and Rousos, 2021; Vettor and Smith, 2021).

Apesar do número de benefícios que a adoção de uma arquitetura baseada em microsserviços pode trazer, nem tudo é perfeito. Uma aplicação que siga esta arquitetura consiste num sistema distribuído em múltiplos processos ou serviços, geralmente espalhados por vários servidores – o que antes era realizado através de chamadas locais efetuadas entre componentes, é agora conseguido por meio de chamadas a outros serviços através da rede. Esta necessidade que cada

microsserviço tem de comunicar através da rede acrescenta alguma complexidade ao sistema (Vettor and Smith, 2021).

De seguida serão apresentados os tipos de comunicação normalmente observados numa solução baseada em microsserviços.

2.2.1 Tipos de comunicação

Num sistema composto por microsserviços, existem vários tipos de comunicação. Cada um destes tipos é empregue conforme o cenário e objetivos da comunicação. É possível dividir os tipos de comunicação em dois grupos: o primeiro grupo define se a comunicação é síncrona ou assíncrona; o segundo grupo define se a comunicação tem um ou vários recetores (de la Torre, Wagner and Rousos, 2021).

No primeiro grupo temos (Aroraa, Kale and Manish, 2017; de la Torre, Wagner and Rousos, 2021):

- **Comunicação síncrona -** A comunicação síncrona acontece quando um cliente envia um pedido a um serviço e aguarda pela resposta ao mesmo, apenas prosseguindo com o processo após a receber
- **Comunicação assíncrona -** Neste tipo de comunicação, o sistema que inicia a comunicação (normalmente através do envio de uma mensagem) não espera por resposta por parte do recetor da mesma, continuando assim com a tarefa que estava a processar. Várias mensagens podem ser enviadas antes do emissor obter resposta

No segundo grupo (de la Torre, Wagner and Rousos, 2021):

- **Único recetor -** Com um único recetor espera-se que cada pedido seja processado por um (e apenas um) recetor
- **Múltiplos recetores -** Como o nome indica, cada pedido pode ser processado por zero ou mais recetores. Este tipo de comunicação deve ser assíncrono. Uma escolha comum para a sua implementação é o uso de uma interface *event-bus*, onde podem ser enviadas atualizações de volta pelo recetor

Numa aplicação baseada em microsserviços, uma combinação dos tipos de comunicação apresentados é usada para possibilitar a troca de informação entre os mesmos. A comunicação síncrona que implica um único recetor é o tipo mais comum de comunicação e é normalmente realizada através de HTTP/HTTPS. Para a comunicação assíncrona entre microsserviços são geralmente usados protocolos de mensagens (de la Torre, Wagner and Rousos, 2021).

Apesar dos tipos de comunicação ajudarem na compreensão sobre os possíveis mecanismos de comunicação entre microsserviços, estes não são o mais importante na construção de uma solução baseada em microsserviços. Um dos aspetos mais importantes na construção da solução deve ser a integração dos microsserviços de forma assíncrona, mantendo a

independência dos mesmos (de la Torre, Wagner and Rousos, 2021). Mesmo assim, é importante aceitar a realidade de que, durante o desenvolvimento de uma aplicação composta por microsserviços, tanto comunicação síncrona como comunicação assíncrona serão necessárias para as várias interações entre os mesmos, devendo a principal preocupação ser quando e que protocolos aplicar (McLarty, 2019).

2.2.2 Tipos de interação na comunicação entre microsserviços

Dada a influência que a interação entre microsserviços tem para o funcionamento de uma aplicação composta pelos mesmos, é importante definir que tipos de interação existem e o que cada um destes tipos implica. Dito isto, é possível classificar as interações entre microsserviços de acordo com os seguintes tipos (McLarty, 2019; Vettor and Smith, 2021):

- ***Query* (Consulta) -** É habitual, um microsserviço precisar de obter alguma informação que um outro microsserviço detém, requisitando resposta imediata para que possa concluir a sua operação. O facto de um microsserviço responsável pela gestão de um cesto de compras precisar de informação (que é da responsabilidade de outro microsserviço) sobre um certo produto para o adicionar a um determinado cesto é um exemplo deste tipo de interação
- ***Command* (Comando) -** Este tipo de interação acontece quando um microsserviço requisita a execução de determinada tarefa por parte de outro microsserviço, sem requerer uma resposta por parte do mesmo. Um microsserviço de gestão de encomendas que pede a um microsserviço de gestão de transporte que faça algo, é um exemplo deste tipo de interação
- ***Event* (Evento) -** Este tipo de interação acontece quando um microsserviço, normalmente designado de *publisher* (aquele que publica), partilha um evento, anunciando a ocorrência de alguma ação. Outros microsserviços, denominados de *subscribers,* que estejam interessados no evento podem reagir ao mesmo de forma apropriada. Um exemplo desta interação: um microsserviço responsável pela gestão do cesto de compras publica um evento de finalização da compra para que outros microsserviços (gestão de encomendas, gestão de inventário, gestão de produtos, etc) reajam de acordo e atualizem as informações relevantes à operação

Tal como nos tipos de comunicação, sistemas baseados em microsserviços utilizam uma combinação de interações na execução de operações que requerem interação entre vários microsserviços.

2.2.3 Cenários comuns na comunicação entre microsserviços

Tendo em conta os tipos de comunicação observados num sistema de microsserviços, e os tipos de interações definidos, torna-se relevante apresentar alguns cenários de comunicação comumente verificados no funcionamento normal de um sistema deste género.

Comunicação síncrona entre microsserviços, onde é necessária uma resposta imediata para continuar o processamento da operação

Como o tipo de comunicação implica, neste cenário, um microsserviço realiza um pedido a outro microsserviço e aguarda pela resposta ao mesmo. O processamento da tarefa pelo microsserviço emissor do pedido fica bloqueado até receber alguma comunicação de volta.

A Figura 1 apresenta um cenário exemplo de comunicação síncrona entre dois microsserviços.

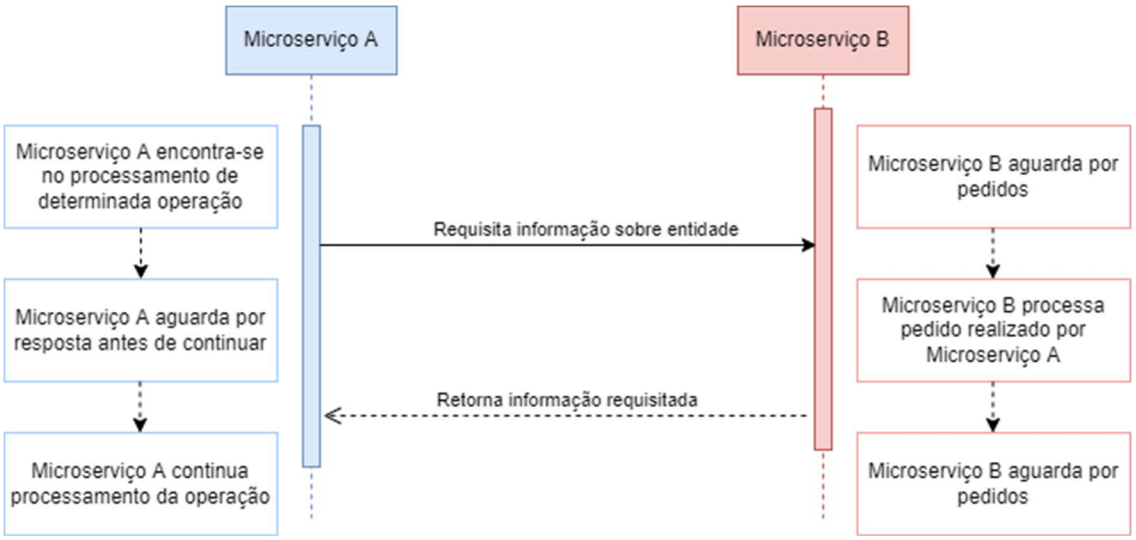

Figura 1 – Exemplo de cenário de comunicação síncrona entre microsserviços

Comunicação assíncrona entre microsserviços, onde um pedido é enviado, mas não se espera uma resposta imediata ao mesmo

Este cenário, tal como o tipo de comunicação envolvido, implica que um microsserviço notifique um outro microsserviço (ou um grupo de microsserviços) sobre alguma operação que realizou e que pode ser relevante para os recetores da notificação. Neste cenário, o emissor da notificação não aguarda por resposta à mesma.

A Figura 2 apresenta um cenário exemplo de comunicação assíncrona com múltiplos recetores. Neste exemplo é possível observar o padrão *publisher/subscriber* (Buck, 2021), onde um microsserviço (*publisher*) publica um evento, e outros microsserviços interessados nesse evento (*subscribers*) são notificados, podendo atuar sobre o mesmo.

Figura 2 – Exemplo de cenário de comunicação assíncrona entre microsserviços

Comunicação em ambiente poliglota, onde existe a necessidade de suportar tecnologias de programação mistas

Num ambiente poliglota (Figura 3), são usadas múltiplas linguagens e tecnologias de programação de acordo com a natureza da aplicação, os requisitos comerciais e as prioridades estabelecidas para o desenvolvimento do sistema, mantendo a comunicação entre os mesmos (de la Torre, Wagner and Rousos, 2021).

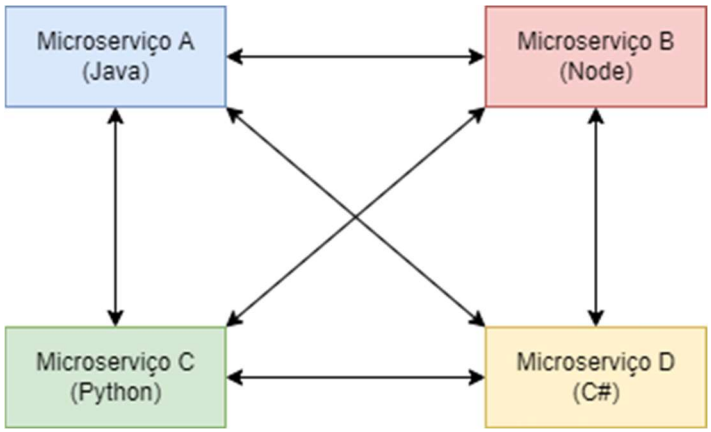

Figura 3 – Exemplo de cenário de comunicação em ambiente poliglota

Comunicação de baixa latência e alto rendimento onde o desempenho é crítico

Como mencionado anteriormente, apesar da arquitetura baseada em microsserviços oferecer um conjunto de vantagens significativo, o facto de obrigar à comunicação entre serviços através da rede acrescenta latência (intervalo de tempo entre o início de uma consulta ou transmissão e a receção da resposta) ao funcionamento da aplicação (de la Torre, Wagner and Rousos, 2021).

A Figura 4 exemplifica uma possível abordagem que apresentaria problemas de latência, devido ao número de chamadas síncronas que são obrigatórias entre os microsserviços que compõem a solução.

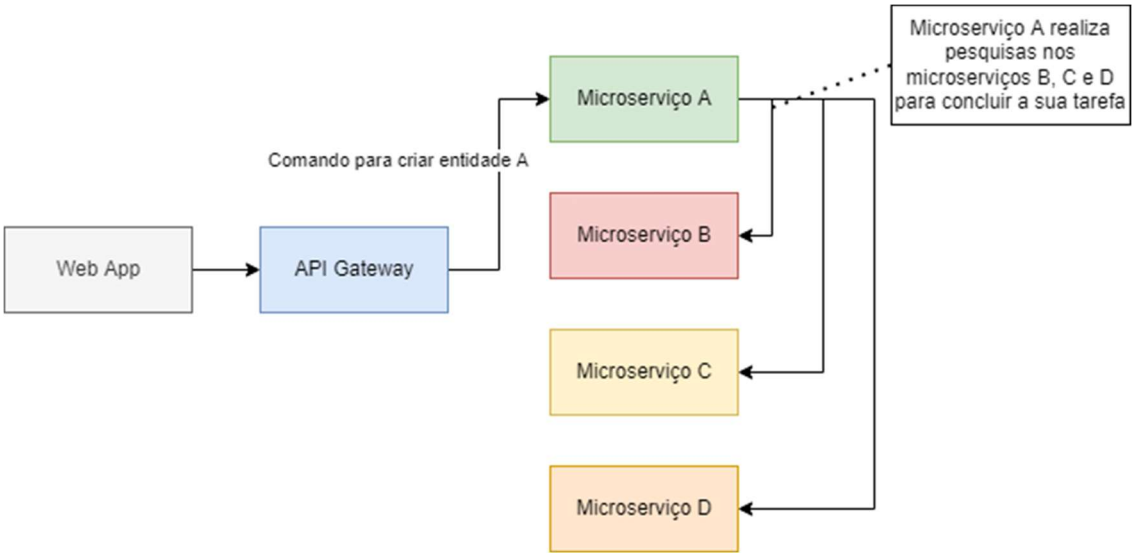

Figura 4 – Exemplo de aplicação que realiza vários pedidos durante a execução de uma tarefa

Dada a importância que a latência apresenta para a atividade normal das grandes empresas (Einav, 2019), este cenário implica que o tempo que a aplicação demora a dar resposta a um pedido seja o menor possível.

Comunicação em ambientes com restrições de rede, onde o tamanho da mensagem tem maior importância

Com o aumentar do número de microsserviços que compõem uma aplicação (podem chegar a centenas ou até milhares) a quantidade de dados enviados entre os mesmo tende a aumentar significativamente. Este cenário implica que as comunicações realizadas entre microsserviços ocupem a menor quantidade de largura de banda possível.

Comunicação direta entre o componente *frontend* e os microsserviços que constituem o *backend* do sistema

Quer seja com chamadas diretas aos microsserviços que compõe o *backend* da aplicação, quer seja através de um *API Gateway* que gere estas chamadas e que as reencaminha para o microsserviço apropriado (Figura 5), é habitual os componentes *frontend* comunicarem com o *backend* da aplicação de modo a cumprir com as suas funcionalidades (de la Torre, Wagner and Rousos, 2021; Vettor and Smith, 2021).

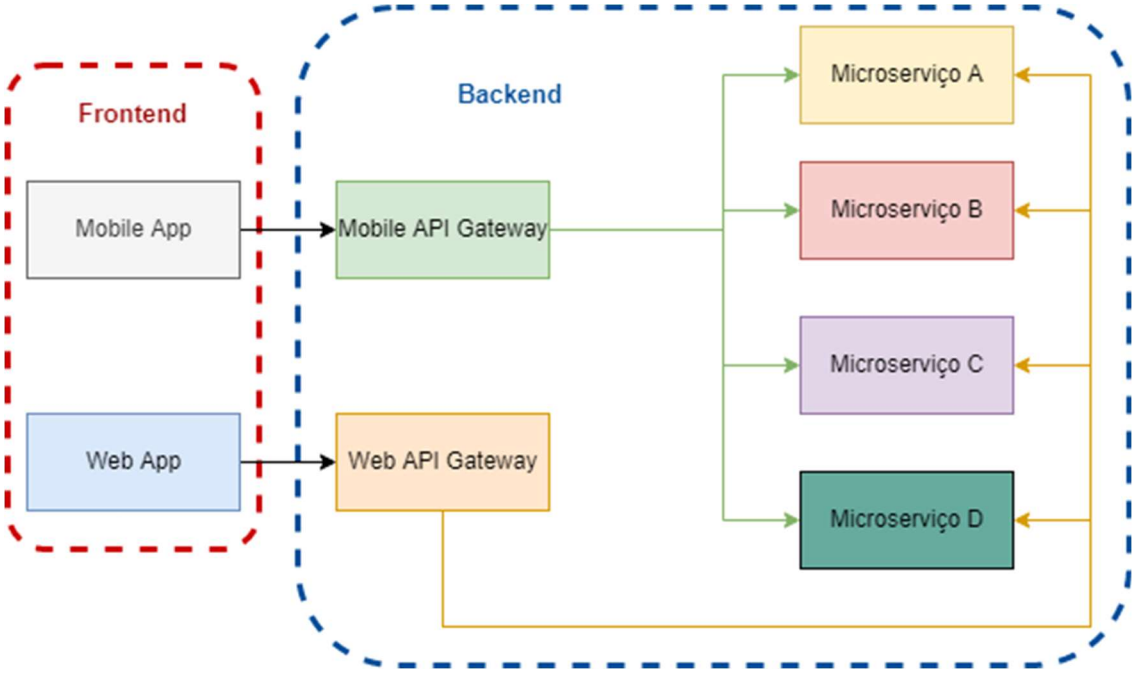

Figura 5 – Exemplo de comunicação entre *frontend* e *backend*

2.3 Protocolos comuns na comunicação entre microsserviços

Atualmente, existem vários protocolos e tecnologias de que desenvolvedores de uma aplicação formada por microsserviços podem tirar partido de maneira a estabelecer comunicação entre os mesmos. O uso de protocolos como HTTP e REST é uma abordagem comum para responder à necessidade de comunicação síncrona baseada em pedido/resposta entre serviços, enquanto que para comunicação assíncrona baseada em mensagens é comum o uso de AMQP. O protocolo a usar na comunicação entre serviços deve ser selecionado tendo em conta o tipo de comunicação e interação que se espera estabelecer entre os mesmos.

Nesta secção são explorados ao detalhe os protocolos considerados de maior relevância para a compreensão deste trabalho – HTTP e REST, e gRPC (o foco deste documento). De seguida, são também resumidamente analisadas algumas abordagens que, apesar de não serem as principais para este trabalho, têm alguma relevância para a contextualização do mesmo.

2.3.1 HTTP e REST

Introdução

REST, um acrónimo de *Representational State Transfer* e inicialmente concebido por Roy Fielding (Fielding, 2000), é um estilo arquitetural que define um conjunto de princípios para auxiliar a construção e organização de sistemas distribuídos (Varanasi and Belida, 2015). Algo importante a retirar sobre esta definição, é que REST é estabelecido como um "estilo": REST

não é uma diretriz, nem uma norma, nem nada que implique um conjunto muito restrito de regras difíceis de seguir para o desenho de uma arquitetura *RESTful* (Doglio, Doglio and Corrigan, 2018).

Serviços web que seguem o estilo arquitetural REST são chamados de serviços web *RESTful*, enquanto que as suas interfaces programáticas são designadas de *REST APIs* (Rodríguez Carlos and Baez, 2016).

A principal motivação por trás do estilo arquitetural REST, é que uma aplicação que siga as restrições definidas pelo mesmo verá melhoramentos nas seguintes áreas: *performance*, escalabilidade, simplicidade da interface, modificabilidade, portabilidade, fiabilidade e visibilidade (Feng, Shen and Fan, 2009; Doglio, Doglio and Corrigan, 2018)

Restrições

De modo a atingir os benefícios expostos anteriormente (*performance*, escalabilidade, etc), REST contém um conjunto de restrições chave:

- **Cliente-Servidor (*Client-Server*):** O principal desta restrição é a separação de responsabilidades e preocupações entre clientes e servidores, o que permite desagregar código *frontend* (responsável pela representação de informação e possível processamento de dados relacionados com a interface com o utilizador) do código *backend* (responsável pelo armazenamento e processamento de dados). Esta restrição possibilita a evolução independente de ambos os componentes, oferecendo flexibilidade ao proporcionar a possibilidade de alterar aplicações cliente sem afetar as aplicações servidor e vice-versa (Varanasi and Belida, 2015; Doglio, Doglio and Corrigan, 2018).
- **Sem estado (*Stateless*):** Esta restrição complementa a restrição anterior, e define que a comunicação entre o cliente e o servidor deve ser "sem estado" – cada pedido realizado pelo cliente deve conter toda a informação necessária para que o servidor seja capaz de o compreender e processar, sem necessitar de tirar partido de dados armazenados ou de informação referente a pedidos anteriores (Feng et al., 2009; Rodríguez Carlos and Baez, 2016; Varanasi & Belida, 2015). Esta restrição tende a melhorar a visibilidade, escalabilidade, fiabilidade e facilidade de implementação de um sistema (Feng, Shen and Fan, 2009; Doglio, Doglio and Corrigan, 2018). Contudo, nem tudo são benefícios: o tráfego de rede pode ser potencialmente prejudicado, pois cada pedido contém informação sobre o seu estado, informação essa que é idêntica entre vários pedidos (Doglio, Doglio and Corrigan, 2018).
- ***Cacheable*:** Aqui, é proposto que cada resposta por parte do servidor a um pedido deve ser explícita ou implicitamente definida como *cacheable* (ou não-*cacheable*). Isto permite ao cliente ou aos seus componentes intermediários armazenar respostas e utilizá-las posteriormente para outros pedidos, reduzindo a carga no servidor e melhorando o desempenho (Feng, Shen and Fan, 2009; Varanasi and Belida, 2015). O compromisso com esta restrição é a possibilidade de os dados armazenados em *cache*

ficarem obsoletos, potencialmente devido a regras de armazenamento em *cache* questionáveis (Doglio, Doglio and Corrigan, 2018).

- **Interface uniforme:** Todas as interações entre clientes, servidores e componentes intermediários, devem ser baseadas na uniformidade das suas interfaces. Isto acaba por simplificar a arquitetura global de um sistema, dado que os componentes que constituem o mesmo podem evoluir de forma independente, desde que cumpram e mantenham o contrato estabelecido com outros componentes. Esta restrição pode ainda ser dividida em quatro sub-restrições: identificação de recursos, representação de recursos, mensagens auto-descritivas e HATEOAS (*Hypermedia as the Engine of Application State*) (Doglio et al., 2018; Rodríguez Carlos and Baez, 2016; Varanasi & Belida, 2015).
- **Sistema em camadas:** A separação dos componentes por camadas, onde cada camada apenas pode utilizar a camada abaixo e comunicar o resultado à camada acima, simplifica a complexidade do sistema e mantém o acoplamento entre os componentes sob controlo (Doglio, Doglio and Corrigan, 2018). As camadas podem ser adicionadas, modificadas, reordenadas ou até removidas de forma a melhorar a escalabilidade (Varanasi and Belida, 2015). A principal desvantagem desta restrição é que, para sistemas pequenos, pode ser introduzida latência indesejada no fluxo global de dados, devido às diferentes interações entre as camadas (Doglio, Doglio and Corrigan, 2018).
- **Código a pedido (*Code-on-demand*):** A única restrição opcional, o que significa que um arquiteto que siga o estilo arquitetural REST pode escolher se pretende ou não recorrer a esta restrição. Com esta restrição, clientes podem alargar a sua funcionalidade através do *download* e execução de código a pedido. Servidores web ou clientes (como um *browser*) podem beneficiar desta restrição (Varanasi and Belida, 2015; Doglio, Doglio and Corrigan, 2018).

Recursos

Um recurso, no contexto de REST, é qualquer coisa que pode ser nomeada, acedida ou manipulada. Um recurso pode ser uma página web, um perfil de utilizador, uma imagem, uma pessoa, etc. No fundo, os recursos definem aquilo em que consiste um serviço, o tipo de informação que o mesmo gere e as ações que o mesmo suporta (Doglio, Doglio and Corrigan, 2018).

Para que seja possível interagir com um recurso, o mesmo deve ser identificável. A Web fornece o URI - *Uniform Resource Identifier* (ou Identificador Universal de Recurso) - que, tal como o nome indica, permite identificar recursos de forma única (Varanasi and Belida, 2015). Um URI segue a seguinte sintaxe presente em Código 1.

```
scheme:scheme-specific-part
```

Código 1 – Sintaxe URI

Exemplos de *scheme* incluem “http” ou “ftp” e são usados para definir a semântica e interpretação do restante do URI. A Tabela 1 contém alguns exemplos de URIs.

Tabela 1 – Exemplo de URIs

URI	Descrição
http://www.example.com/entities	Representa uma coleção de recursos "entity"
http://www.example.com/entities/1	Representa um recurso “entity” com identificador “1”
http://www.example.com/entities/1/children	Representa uma coleção de recursos “child” que se encontram associados à "entity" com identificador "1"
http://www.example.com/entities/1/children/123	Representa um recurso “child” com identificador “123”

Quando um recurso *RESTful* é requisitado por um cliente, os dados e metadados que representam o mesmo precisam de ser serializados numa representação, que é basicamente uma descrição atual do estado do recurso, antes do envio da resposta. Esta representação pode ser em diferentes formatos, – entre os mais comuns estão JSON e XML – onde todos representam um recurso de forma correta, e onde cabe ao cliente ler e analisar a informação recebida (Varanasi and Belida, 2015). A representação a usar deve ser negociada em tempo de execução, com o cliente a expressar a sua preferência quanto à representação do recurso através de *Content Negotiation* - este método faz parte do padrão HTTP e tende a ser o método preferencial de definição da representação a usar; para além disso fomenta a reutilização, a interoperabilidade e o desacoplamento, devendo ser usado em detrimento da especificação das extensões do ficheiro (outro método de definição da representação, onde a extensão do ficheiro é adicionada ao URL do recurso) (Rodríguez Carlos and Baez, 2016).

Operações

Para possibilitar o acesso e modificação de recursos, APIs REST devem seguir um conjunto uniforme de operações ou verbos que tenham por base um certo padrão (Interface Uniforme). O padrão HTTP fornece um conjunto de métodos (ou verbos) que permitem definir o tipo de operação a realizar sobre um determinado recurso (Varanasi and Belida, 2015):

- **GET:** deve ser usado quando se pretende obter a representação de um recurso
- **HEAD:** deve ser usado quando se pretende obter apenas os metadados sobre o estado de um recurso
- **DELETE:** deve ser usado quando se pretende eliminar um recurso
- **PUT:** deve ser usado quando se pretende alterar o estado de um recurso

- **POST:** deve ser usado quando se pretende criar um recurso
- **PATCH:** deve ser usado quanto se pretende realizar uma modificação parcial de um recurso

***Hypermedia as the Engine of Application State* (HATEOAS)**

Hypermedia as the Engine of Application State ou HATEOAS é uma das restrições necessárias para que uma interface REST seja uniforme, cujo objetivo é auxiliar os clientes no consumo do serviço sem necessidade de conhecimento prévio da API. O principal conceito por detrás de HATEOAS é simples – uma resposta deve incluir ligações a outros recursos, permitindo assim aos clientes recorrer destas ligações para comunicar com o servidor e modificar o estado do mesmo (Varanasi and Belida, 2015; Doglio, Doglio and Corrigan, 2018).

Códigos de estado HTTP

Outro padrão interessante do qual o REST pode beneficiar quando baseado em HTTP é a utilização de códigos de estado HTTP – um código de estado é um número que permite ao servidor comunicar o resultado do processamento de um pedido. É possível agrupar os códigos de estado de acordo com as seguintes categorias (Varanasi and Belida, 2015):

- **Códigos informacionais:** indicam que o servidor recebeu o pedido, mas que não terminou o seu processamento. Encontram-se na série de códigos 1xx.
- **Códigos de sucesso:** indicam que o pedido foi recebido e processado com sucesso. Encontram-se na série de códigos 2xx.
- **Códigos de redirecção:** indicam que o pedido foi processado, mas que uma ação (normalmente a redirecção para outra localização) adicional deve ser realizada pelo cliente para que o mesmo seja completado. Encontram-se na série de códigos 3xx.
- **Códigos de erro do cliente:** indicam que ocorreu algum erro ou problema com o pedido do cliente. Encontram-se na série de códigos 4xx.
- **Códigos de erro do servidor:** indicam que ocorreu algum erro ou problema durante o processamento do pedido do cliente. Encontram-se na série de códigos 5xx.

2.3.2 gRPC

Como já apontado anteriormente, quando se trata de estabelecer comunicações síncronas entre microsserviços baseadas no estilo de pedido/resposta, a abordagem mais convencional e comumente seguida passa por construí-los como serviços RESTful, onde a aplicação ou serviço é definida como um conjunto de recursos que podem ser acedidos e ter o seu estado alterado através de chamadas de rede realizadas por meio do protocolo HTTP. Contudo, para a maioria dos casos de utilização na construção de comunicações internas, os serviços RESTful acabam por se mostrar bastante volumosos, ineficientes e propensos a erros (Indrasiri and Kuruppu, 2020).

A necessidade de uma tecnologia de comunicação interna que fosse mais escalável e eficiente que serviços RESTful abriu portas para que gRPC entrasse em cena.

Introdução

Internamente, e com o objetivo de ligar milhares de microsserviços distribuídos por múltiplos centros de dados e construídos com recurso a tecnologias distintas, a Google fazia uso de uma *framework* RPC chamada *Stubby* (Talwar, 2016). Apesar da *framework Stubby* apresentar funcionalidades convenientes à sua utilização, esta não se encontra padronizada para ser utilizada como uma *framework* genérica, dada a sua forte ligação à infraestrutura interna da Google. Como resposta a este problema, e com o objetivo de fornecer a mesma escalabilidade, desempenho e funcionalidade da *framework Stuby*, em 2015, a Google lançou gRPC como uma estrutura RPC *open-source* - uma infra-estrutura RPC normalizada, de uso geral e multiplataforma (Indrasiri and Kuruppu, 2020).

De maneira geral, é possível definir gRPC como uma tecnologia de comunicação que permite ligar, invocar, operar e depurar aplicações distribuídas tão facilmente como o simples ato de invocar uma função local (Indrasiri and Kuruppu, 2020; Vettor and Smith, 2021).

O básico

A primeira tarefa aquando do início do desenvolvimento de uma aplicação gRPC é a definição de uma interface de serviço. A definição da interface de serviço (Indrasiri and Kuruppu, 2020):

- Contém informação sobre como o serviço pode ser consumido
- Apresenta informação sobre que métodos os consumidores deste serviço podem chamar remotamente
- Inclui informação sobre que parâmetros e formatos de mensagem devem ser utilizados ao invocar os métodos disponíveis
- A linguagem especificada durante a definição do serviço é conhecida como Linguagem de Definição de Interface (*Interface Definition Language – IDL*)

gRPC utiliza buffers de protocolo (*protocol buffers* ou *protobuf*) como IDL para definir a interface de serviço – a definição da interface de serviço é especificada num ficheiro proto, que nada mais é do que um ficheiro de texto normal com uma extensão *.proto* (Himschoot, 2021). Os buffers de protocolo são um mecanismo extensível, independente de plataforma e de linguagem, para a serialização de dados estruturados. Os serviços gRPC são definidos num formato de buffer de protocolo, com parâmetros do método RPC e tipos de retorno especificados como mensagens de buffer de protocolo (Indrasiri and Kuruppu, 2020). Código 2 apresenta um simples exemplo de um serviço gRPC de consulta de livros, definido através de buffers de protocolo, com o objetivo de auxiliar na compreensão de como um serviço é definido.

```
syntax = "proto3";

package com.book;

message Book {
    int64 isbn = 1;
    string title = 2;
    string author = 3;
}

message GetBookRequest {
    int64 isbn = 1;
}

message GetBookViaAuthor {
    string author = 1;
}

service BookService {
    rpc GetBook (GetBookRequest) returns (Book) {}
    rpc GetBooksViaAuthor (GetBookViaAuthor) returns (stream Book) {}
    rpc GetGreatestBook (stream GetBookRequest) returns (Book) {}
    rpc GetBooks (stream GetBookRequest) returns (stream Book) {}
}
```

Código 2 – Exemplo de definição de serviço gRPC através de buffers de protocolo

A definição de serviço (*Language Guide (proto3) | Protocol Buffers | Google Developers*, 2021):

- Começa com a versão de buffer de protocolo usada (neste caso `proto3`)
- Apresenta um nome de *package* (`com.book`). Este nome é opcional, e é usado de maneira a evitar conflitos entre os nomes dos tipos de mensagens de protocolo
- Contém os vários tipos/formatos de mensagens de protocolo (`Book`, `GetBookRequest`, `GetBookViaAuthor`). A definição de cada um destes tipos de mensagens especifica os campos (pares nome/valor) que se pretende incluir no mesmo, onde cada campo tem um nome e um tipo
- Como é possível observar, cada campo presente na definição do tipo de mensagem é acompanhado de um número (`int64 isbn = 1`). Estes números servem para identificar o campo no formato binário da mensagem, e não devem ser alterados uma vez que o tipo de mensagem esteja a ser utilizado. Aqui, vale a pena notar que:
    - Os números dos campos no intervalo de 1 a 15 levam um byte a codificar, incluindo o número do campo e o tipo do campo

    - Os números do campo no intervalo de 16 a 2047 necessitam de dois bytes.
    - Portanto, os números de 1 a 15 devem ser reservados para elementos de mensagens muito frequentes
- Define a interface de serviço de um serviço gRPC (`service BookService`). Dentro desta definição são descritos os vários métodos, incluindo os parâmetros esperados e o formato da mensagem devolvida

Assim que a definição do serviço esteja completa, esta pode ser utilizada para gerar código do lado do servidor e código do lado do cliente, através do compilador de buffer de protocolo (*protoc*). Como gRPC não depende da linguagem de implementação escolhida, através de um único ficheiro *.proto* é possível gerar código para o servidor e código para o cliente para qualquer uma das linguagens de programação suportadas.

HTTP/2

HTTP/2 é a segunda versão *major* do protocolo HTTP, e foi introduzida como resposta a alguns dos problemas encontrados com a versão anterior (HTTP/1). Em HTTP/2, toda a comunicação entre um cliente e um servidor é realizada através de uma conexão TCP capaz de transportar qualquer número fluxos bidirecionais de bytes. Para compreender HTTP/2, é importante alguma familiarização com os seguintes conceitos (Indrasiri and Kuruppu, 2020).

- ***Stream:*** Fluxo bidirecional de bytes dentro de uma ligação. Uma *stream* é capaz de transportar uma ou mais mensagens.
- ***Frame:*** Representa a menor unidade de comunicação em HTTP/2. Cada *frame* contém um cabeçalho que, no mínimo, identifica a *stream* a que o *frame* pertence.
- **Mensagem (*Message*):** Sequência completa de *frames* que mapeia para uma mensagem HTTP que consiste em um ou mais *frames*.

HTTP/2 suporta todas as principais funcionalidades observadas em HTTP/1, mas de uma forma mais eficiente. Aplicações que tiram partido de HTTP/2 tendem a ser mais rápidas, simples e robustas. As novas funcionalidades introduzidas em HTTP/2 incluem:

- **Uso de protocolo binário:** Originalmente, HTTP era um protocolo baseado em texto, o que significa que os pacotes trocados durante a comunicação eram legíveis por humanos. Com a introdução de HTTP/2, foi adotada uma estrutura binária dado o facto que esta é mais compacta e mais eficiente durante o processamento (de Saxcé, Oprescu and Chen, 2015).
- ***Multiplexing*:** HTTP/2 introduz a noção de *stream:* cada pedido realizado por um cliente é atribuído a uma determinada *stream*, e todas as *streams* são agrupadas através de uma única ligação TCP. Como consequência, o número de ligações TCP a serem tratadas pelo servidor é reduzido, e os pedidos podem ser respondidos simultaneamente(de Saxcé, Oprescu and Chen, 2015) .

- **Compressão:** HTTP/2 é acompanhado do algoritmo HPACK (Peon and Ruellan, 2015) que especifica como comprimir cabeçalhos HTTP, o que resulta numa reduzida utilização de rede.
- ***Push* por parte do servidor:** É comum, quando um servidor recebe um pedido de uma página web, o mesmo poder antecipar a realização de uma série de outros pedidos por parte do cliente com o objetivo de obter todas as dependências da página (CSS, imagens, entre outros). Em HTTP/1, o servidor não pode fornecer conteúdos que não foram explicitamente requisitados; em HTTP/2 o servidor pode "empurrar" dados que considere úteis - o cliente opta por recusar ou aceitar os dados, sendo os mesmos armazenados na cache do navegador para utilização posterior (de Saxcé, Oprescu and Chen, 2015). Estratégias de *push* de informação por parte do servidor devem ser consideradas ao adotar esta técnica para garantir melhorias de performance (Zimmermann *et al.*, 2017).
- **Mecanismo de priorização:** Este mecanismo é usado para que o cliente possa especificar um certo ranqueamento entre as *streams* existentes numa conexão HTTP/2. O mecanismo pode permitir ao servidor (por exemplo) dar maior prioridade a ficheiros CSS e JavaScript do que a imagens, o que pode resultar em menores tempos para o carregamento de páginas (de Saxcé, Oprescu and Chen, 2015). Apesar do mecanismo de priorização usado poder influenciar o desempenho no carregamento de uma página web, o mesmo deve ser acompanhado de outros processos de otimização para que a velocidade de carregamento da página seja garantida (Wijnants *et al.*, 2018).

gRPC usa HTTP/2 como protocolo de transporte para o envio de mensagens através da rede, sendo esta uma das principais razões por trás de gRPC ser uma *framework* RPC de alto desempenho. Quando uma aplicação cliente cria um canal gRPC, o que acontece é que uma ligação HTTP/2 é criada com o servidor. Uma vez que o canal esteja estabelecido, o mesmo pode ser reutilizado para enviar múltiplas chamadas remotas para o servidor (Indrasiri and Kuruppu, 2020):

- Chamadas remotas são mapeadas para *streams* em HTTP/2
- Mensagens enviadas nestas chamadas remotas são enviadas como *frames* HTTP/2
- Um *frame* pode transportar uma mensagem gRPC de comprimento pré-estabelecido
- Caso a mensagem seja grande, a mesma pode abranger vários *frames* de dados

A Figura 6 apresenta um exemplo de uma conexão HTTP/2.

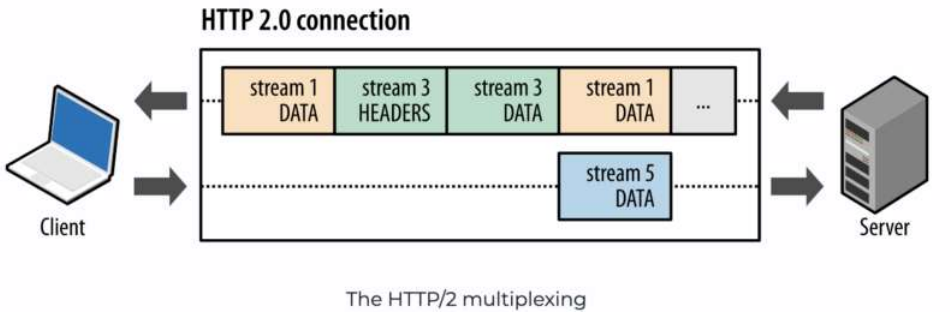

Figura 6 – Conexão HTTP/2

Estrutura das mensagens de Pedido/Resposta

Em gRPC, o cliente é aquele que inicia a chamada remota através de uma mensagem de pedido (*request message*). Esta mensagem de pedido consiste em três componentes: cabeçalhos do pedido, mensagem de comprimento pré-fixado, e um sinal (*flag*) de fim de *stream* que notifica o servidor sobre o término do envio de informação (Indrasiri and Kuruppu, 2020).

Código 3 apresenta um exemplo dos elementos que constituem uma mensagem de pedido. O

```
HEADERS (flags = END_HEADERS)
:method = POST
:scheme = http
:path = /BookService/GetBook
:authority = example.com
grpc-timeout = 1S
content-type = application/grpc
grpc-encoding = gzip
authorization = Bearer xxxxxx

DATA (flags = END_STREAM)
<Length-Prefixed Message>
```

Código 3 – Exemplo de sequência de elementos numa mensagem de pedido

exemplo é composto por dois *frames*: HEADERS e DATA. Em HEADERS é possível observar:

- **:method –** Define o método HTTP. Em gRPC, este valor é sempre POST
- **:scheme –** Caso TLS (*Transport Level Security*) esteja a ser usado este valor é "https", caso contrário é "http"
- **:path –** Define o caminho. Em gRPC, este valor é construído com "/", seguido do nome do serviço (neste caso BookService), seguido de "/" e finalmente o nome do método (GetBook)
- **:authority –** Define o nome do anfitrião virtual do URI (*Uniform Resource Identifier*) alvo
- **grpc-timeout –** Define o tempo limite de chamada. Caso este valor não seja especificado, o servidor deve assumir que o tempo limite é infinito

- **content-type –** Como o nome indica, define o tipo de conteúdo. Em gRPC, este valor deve começar com "application/grpc", caso contrário o servidor gRPC responde com HTTP 415 (este erro indica que o tipo de conteúdo não é suportado).
- **grpc-encoding** – Define o tipo de compressão da mensagem. Os seguintes valores são válidos: "identity", "gzip", "deflate", "snappy" ou {*custom*}
- **authorization** – Metadados opcionais usados para aceder ao *endpoint*

O fim da mensagem de pedido é indicado através da *flag* "END_STREAM" no último DATA *frame* enviado. Caso não existam mais dados a enviar e exista a necessidade de fechar a *stream* de pedido, esta *flag* deve ser enviada (Indrasiri and Kuruppu, 2020).

A mensagem de resposta é gerada pelo servidor em resposta a um pedido de um cliente e também pode ser dividida em três componentes principais: cabeçalhos da resposta, mensagem de comprimento pré-fixado, e *trailers*.

```
HEADERS (flags = END_HEADERS)
:status = 200
grpc-encoding = gzip
content-type = application/grpc

DATA
<Length-Prefixed Message>

HEADERS (flags = END_STREAM, END_HEADERS)
grpc-status = 0 # OK
grpc-message = xxxxxx
```

Código 4 – Exemplo de sequência de elementos numa mensagem de resposta

Código 4 apresenta um exemplo dos elementos que constituem uma mensagem de resposta. Após análise ao *frame* HEADERS inicial é possível constatar que tem atributos em comum com o *frame* HEADERS de uma mensagem de pedido, apresentado de novo apenas o atributo **:status**, que indica o estado do pedido HTTP. Vale também notar que a *flag* "END_STREAM" não é enviada dentro de um *frame* DATA de uma mensagem de resposta; esta *flag* é enviada através de um *frame* HEADER chamado *trailer*. *Trailers* são enviados no fim, e servem para notificar o cliente sobre o facto de que o envio da mensagem de resposta foi terminado. Estes contêm:

- **grpc-status –** Define o código de estado gRPC (Tibrewal *et al.*, 2020)
- **grpc-message –** Definido quando existe algum erro. Descreve o erro

Padrões de comunicação

Existem quatro padrões fundamentais de comunicação que são utilizados em aplicações baseadas em gRPC: RPC simples (ou unário), *streaming* do lado do servidor, *streaming* do lado do cliente, e *streaming* bidirecional. Estes padrões são explorados de seguida.

- **RPC Simples:** Neste modo de operação, existe sempre um único pedido e uma única resposta a esse pedido. Como é possível observar na Figura 7, e como discutido anteriormente, a mensagem de pedido contém os cabeçalhos, seguidos de uma mensagem de comprimento pré-fixado, que pode estender-se por um ou mais *frames*, e finalmente uma sinalização de fim da *stream* (*End of Stream flag*) que marca o fim da mensagem de pedido e que “meio-fecha” a conexão. Diz-se que a conexão fica "meio-fechada" pois, apesar do cliente não ser capaz de enviar mais mensagens, o mesmo pode ainda receber mensagens provenientes do servidor. Após receber a totalidade da mensagem, o servidor cria e envia a resposta à mesma através de uma mensagem de resposta de formato semelhante ao discutido anteriormente e que é possível observar na Figura 7. A comunicação termina assim que o servidor envia os *trailers*.

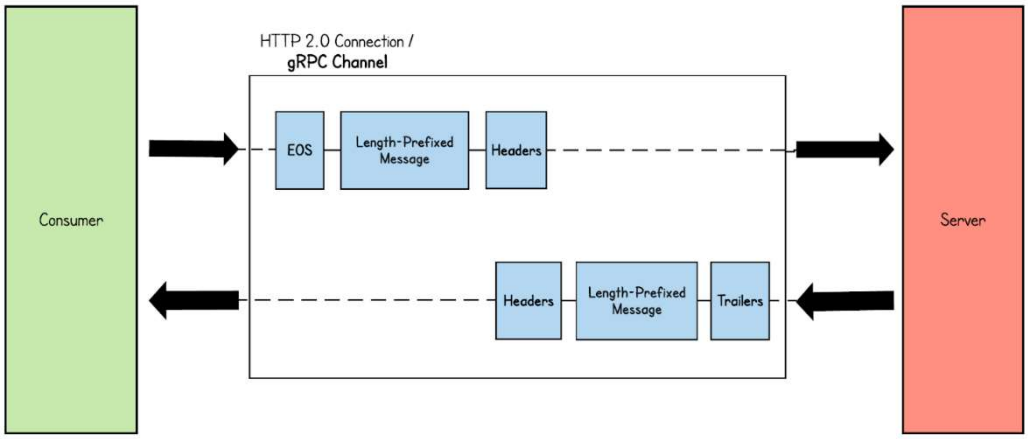

Figura 7 – Exemplo de RPC Simples (Indrasiri and Kuruppu, 2020)

- ***Streaming* RPC do lado do servidor:** Na perspetiva de uma aplicação cliente, tanto a comunicação através de RPC simples como o *streaming* RPC do lado do servidor apresentam o mesmo fluxo – como evidenciado pela Figura 8, o cliente apenas envia uma mensagem de pedido. Do lado do servidor, este espera até receber toda a mensagem de pedido, e seguidamente envia uma mensagem de resposta que contém várias mensagens de comprimento pré-fixado.

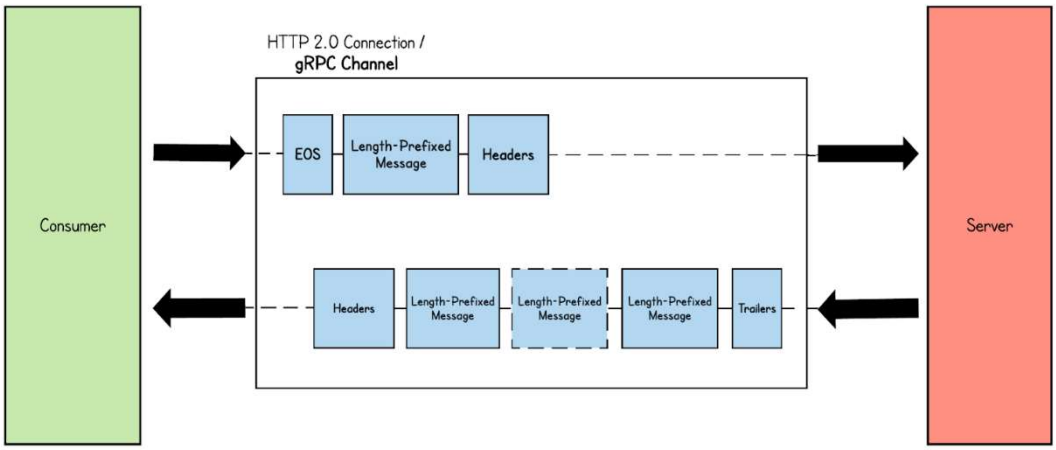

Figura 8 – Exemplo de *streaming* do lado do servidor (Indrasiri and Kuruppu, 2020)

- ***Streaming* RPC do lado do cliente:** Como é possível observar pela Figura 9, *streaming* RPC do lado do cliente acaba por seguir uma lógica quase oposta ao *streaming* do lado do servidor. Aqui, o cliente envia várias mensagens e o servidor responde com apenas uma única mensagem.

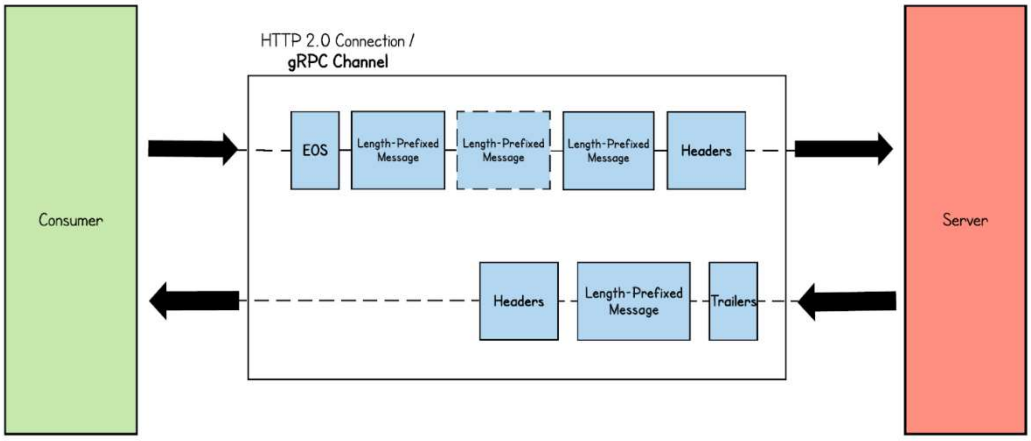

Figura 9 – Exemplo de *streaming* do lado do cliente (Indrasiri and Kuruppu, 2020)

- ***Streaming* RPC bidirectional:** Este padrão (Figura 10) acaba por ser uma junção dos dois padrões anteriores. Neste, o cliente estabelece a ligação através do envio dos cabeçalhos. Assim que a conexão é estabelecida tanto o cliente como o servidor enviam uma série de mensagens de comprimento pré-definido simultaneamente, sem aguardarem que o outro termine. Ambos podem fechar a conexão do seu lado, impossibilitando o envio de mais mensagens.

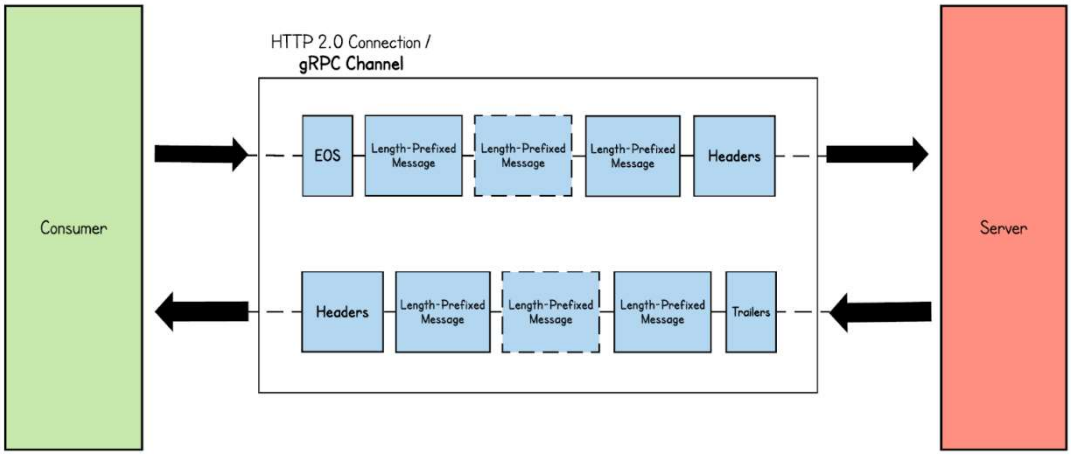

Figura 10 – Exemplo de *streaming* bidirecional (Indrasiri and Kuruppu, 2020)

2.3.3 Outras abordagens à comunicação entre microsserviços

Existem várias possibilidades no que toca a comunicação entre microsserviços que não são exploradas neste documento, visto o mesmo se focar em gRPC e REST. No entanto, é relevante na mesma mencionar brevemente algumas das potenciais alternativas:

- **Thrift:** Thrift permite definir tipos de dados e interfaces de serviço através de ficheiros *.thrift*. Semelhante ao que acontece em gRPC, tendo por base a definição da interface, é posteriormente possível gerar código que permite a construção de aplicações cliente e aplicações servidor, capazes de comunicar entre si mesmo através de diferentes linguagens de programação (Indrasiri and Siriwardena, 2018)
- **Uso de *message broker*:** *Message broker* é um software que permite a aplicações, sistemas e serviços comunicar e trocar informação entre si mesmos. Isto é realizado através da tradução de uma mensagem do protocolo de mensagem do produtor da mesma, para o protocolo de mensagem do recetor. Isto permite que serviços enviem mensagens uns para os outros diretamente, mesmo tendo sido escritos em linguagens diferentes ou implementados em plataformas diferentes (Magnoni, 2015). Vale a pena notar que o uso de um *message broker* deve ser visto como algo que complementa o uso de soluções como gRPC ou REST (comumente utilizadas em comunicação síncrona), visto o mesmo se focar na comunicação assíncrona entre aplicações
- ***Falcor:*** *Falcor* é uma biblioteca JavaScript criada pela Netflix (Husai, Taylor and Paulson, 2015). Esta biblioteca permite representar fontes de dados remotas como um único modelo de domínio através de JSON Graph (uma convenção de para a modelação de informação organizada como um grafo para um único objeto JSON). Devido a funcionalidades de *caching* que *Falcor* oferece, a utilização da rede para pedidos ao servidor tende a ser otimizada (Beták, 2016)

- **GraphQL:** GraphQL, que se tornou *open-source* em 2015 (Sharma, 2021), é uma linguagem de consulta e ambiente de execução cuja prioridade é fornecer exatamente os dados solicitados pelos clientes (Porcello and Banks, 2018). Representa uma alternativa a REST (sendo capaz até de apresentar desempenho superior (Seabra et al., 2019)) que, desde o seu lançamento, tem sido adotada cada vez por mais utilizadores (Hartig & Pérez, 2017). Existem várias caraterísticas frequentemente associadas a esta tecnologia (Porcello and Banks, 2018; Brito, Mombach and Valente, 2019; Sharma, 2021):
    - **Declarativa** - consultas em GraphQL são consideradas declarativas, ou seja, o cliente declara com exatidão os campos em que está interessado e que devem ser considerados para a construção da resposta, resultando numa resposta que "espelha" a consulta realizada
    - **Hierárquica** - GraphQL é de natureza hierárquica, seguindo naturalmente as relações entre objetos. Campos estão inseridos dentro de outros campos e o formato da consulta é igual ao formato da resposta a essa consulta
    - **Fortemente tipada** - cada nível de uma consulta GraphQL corresponde a um tipo particular, e cada tipo descreve um conjunto de campos disponíveis. Tipos vão desde os simples valores primitivos (strings, booleans, integers) até objetos de maior complexidade. Semelhante a SQL, o que permite a GraphQL fornecer mensagens de erro descritivas antes de executar uma consulta
    - **Introspetiva -** um servidor GraphQL pode ser consultado sobre quais tipos o mesmo suporta, permitindo a outras ferramentas e aplicações clientes construírem sobre esta informação, possibilitando a geração automática de documentação
    - ***Client-Driven* -** tendo em conta que o formato da resposta dada por um servidor depende completamente da consulta determinada pelo cliente, é possível adicionar novas funcionalidades e dados do lado do servidor sem impactar negativamente as aplicações cliente

  Apesar de GraphQL apresentar um conjunto de caraterísticas relevantes ao desenvolvimento de uma solução baseada em microsserviços, e apesar do seu crescimento em popularidade, a adoção desta tecnologia tende a ser mais adequada quando aplicada a serviços ou APIs viradas para o exterior de uma aplicação, onde os clientes necessitam de maior controlo sobre os dados que consomem (Indrasiri and Kuruppu, 2020).

2.4 Trabalhos Similares

O principal objetivo desta secção é identificar e apresentar alguns trabalhos publicados que apresentam um tema (ou temas) semelhantes ao explorado ao longo deste documento – aqui procura-se focar em trabalhos que explorem gRPC ou REST, e/ou que estudem a adoção destas abordagens num cenário de desenvolvimento de microsserviços e/ou que realizem uma comparação que considere pelo menos uma destas abordagens.

Para isso são listados os diferentes trabalhos, onde para cada trabalho é apresentado um sumário que descreve, sucintamente, algumas caraterísticas do mesmo e o seu conteúdo:

- ***Developing the guidelines for migration from RESTful microservices to gRPC*** (Stefanic, 2021): Neste trabalho, o autor apresenta a tecnologia gRPC, descreve o seu uso em ambientes de produção e realiza uma comparação entre esta tecnologia e REST. Para além disso o autor apresenta um processo de decisão que tem como objetivo auxiliar empresas a decidir se devem adotar gRPC para a comunicação entre microsserviços. O trabalho termina com um caso prático de migração de REST para gRPC que leva o autor a concluir que: gRPC apresenta algumas vantagens como performance e consistência de dados, contudo, esta tecnologia não deve ser vista como uma substituta absoluta de REST e, devido à imaturidade da tecnologia e ao facto de que a mesma é desenvolvida de acordo com interesses internos, a mesma apresenta ainda alguns problemas que podem afastar algumas empresas da sua adoção
- ***REST or GraphQL? A Performance Comparative Study*** (Seabra, Nazário and Pinto, 2019): Este trabalho compara a performance entre aplicações que disponibilizam uma interface REST e aplicações que oferecem uma interface GraphQL - semelhante ao objetivo do presente trabalho, mas tendo em consideração diferentes abordagens. São avaliadas métricas como pedidos por segundo, tempo por pedido, tempo de duração de pedidos concorrentes e taxa de transferência. O trabalho conclui que serviços GraphQL apresentaram desempenho superior em dois terços das aplicações avaliadas nas métricas pedidos por segundo e taxa de transferência
- ***REST: An Alternative to RPC for Web Services Architecture*** (Feng, Shen and Fan, 2009): Neste trabalho REST é estudado como alternativa ao RPC. Para isso, os dois estilos são apresentados, analisados e comparados no que toca à sua escalabilidade, acoplamento e segurança. É concluído que a arquitetura de serviços web RESTful é melhor na área da escalabilidade, acoplamento e desempenho
- ***REST vs GraphQL: A Controlled Experiment*** (Brito and Valente, 2020)**:** Neste artigo, é descrita uma experiência controlada com 22 estudantes, a quem foi pedido que implementassem oito *queries* de acesso a um serviço web, utilizando GraphQL e REST Os resultados apresentados mostram que o uso de GraphQL requer menos esforço para a implementação de *queries* quando comparado com REST, principalmente quando as consultas REST incluem *endpoints* mais complexos, com vários parâmetros. Foi também concluído que o uso de GraphQL supera REST mesmo entre participantes mais experientes e entre participantes com experiência anterior em REST, mas sem experiência anterior em GraphQL

2.5 Análise

No decorrer deste capítulo foram detalhados vários cenários de comunicação que são vulgarmente observados em aplicações compostas por microsserviços. De entre estes cenários, é possível notar a presença daqueles que a implementação se pretende estudar,

nomeadamente: comunicação síncrona, comunicação em ambiente poliglota, comunicação de baixa latência e alto rendimento, comunicação em ambientes com restrições de rede e, finalmente, comunicação direta entre *frontend* e *backend*.

Com base neste contexto, existe a necessidade de desenhar uma solução que tenha em consideração as abordagens/tecnologias que se procura explorar (REST, gRPC), e que permita observar a sua adoção para a implementação dos vários cenários.

3 Análise de Valor

A análise de valor pode ser definida como "um processo de revisão sistemática que é aplicado aos desenhos de produtos existentes, a fim de comparar a função do produto requerida por um cliente para satisfazer os seus requisitos ao menor custo consistente com o desempenho especificado e fiabilidade necessária" ou, mais resumidamente, como "um processo sistemático, formal e organizado de análise e avaliação" (Rich and Holweg, 2000).

O objetivo deste capítulo é expor a análise de valor realizada. Para isso, o capítulo começa com uma secção sobre o processo de inovação onde é aplicado o *New Concept Development Model* (NCD) (Koen *et al.*, 2001). De seguida é apresentado o valor da solução, que inclui o *Value Proposition Canvas*. Finalmente o capítulo termina expondo a aplicação da técnica AHP, que foi usada para selecionar a ideia a seguir.

3.1 Processo de Inovação

O processo de inovação pode ser dividido em três partes: o FFE (*Fuzzy Front End*), o processo de desenvolvimento de novos produtos (*New Product Development process*), e por último a Comercialização (Figura 11).

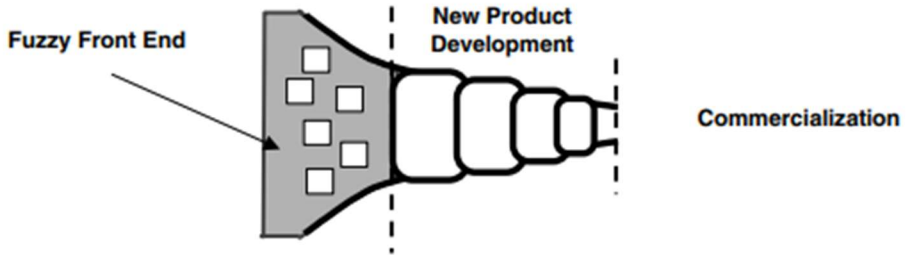

Figura 11 – Divisão do processo de inovação em três áreas (Belliveau, Griffin and Somermeyer, 2004)

A primeira parte do processo de inovação, o *Fuzzy Front End*, é definido pelas atividades (atividades estas que tendem a ser caóticas, imprevisíveis e não-estruturadas) que antecedem

o processo formal e estruturado de desenvolvimento de novos produtos (NPD). Este primeiro passo do processo de inovação é considerado como "uma das maiores oportunidades de melhoria do processo global de inovação" (Koen *et al.*, 2001).

NPD, a segunda parte do processo de inovação, aborda a implementação das ideias com origem na etapa anterior, e é definida como "um disciplinado e bem definido conjunto de tarefas e passos que descrevem os meios normais pelos quais uma empresa converte repetitivamente ideias embrionárias em produtos ou serviços comercializáveis" (Belliveau, Griffin and Somermeyer, 2004). As principais diferenças entre as duas primeiras etapas podem ser observadas na Tabela 2.

Tabela 2 – Comparação FEE/NPD

	***Fuzzy Front End* (FEE)**	***New Product Development* (NPD)**
Natureza do Trabalho	- Experimental, frequentemente caótico - Momentos "eureka" - Difícil de programar	Disciplinado e orientado para os objetivos com um plano de projeto
Data de comercialização	Imprevisível ou incerta	Alto nível de certeza
Financiamento	Variável	Orçamentado
Expetativas de receitas	Muitas vezes incerta, com um alto nível de especulação	Previsível, com crescente certeza, análise, e documentação à medida que a data de lançamento do produto se aproxima
Atividade	Indivíduos e equipas que realizam investigação para minimizar o risco e otimizar o potencial	Produto multifuncional e/ou equipa de desenvolvimento do processo
Medidas de progresso	Conceitos reforçados	Realização de objetivos

Em último está a fase de Comercialização. A fase de Comercialização engloba "o processo de levar um novo produto do desenvolvimento ao mercado. Inclui geralmente o lançamento e o aumento da produção, materiais de marketing e desenvolvimento de programas, da cadeia de fornecimento, de canais de vendas, de formação, e de serviços e apoio.

3.1.1 Modelo New Concept Development (NCD)

De modo a combater a falta de uma linguagem comum e/ou existência de termos que possibilitassem a comparação de processos FEE, e determinar um conjunto de melhores práticas a seguir, foi proposto o modelo de *New Concept Development* (NCD) (Koen *et al.*, 2001).

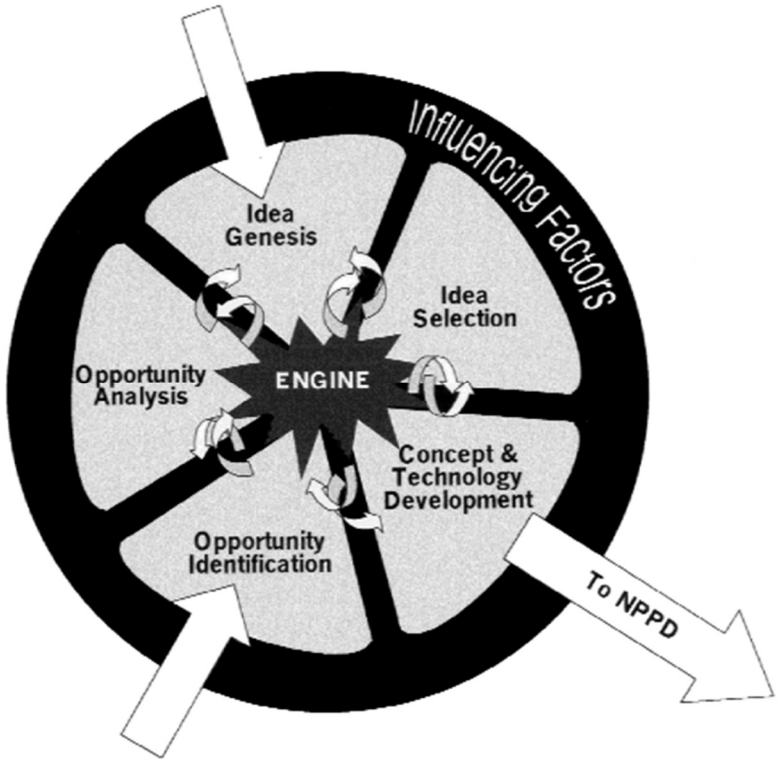

Figura 12 – Modelo *New Concept Development* (Koen *et al.*, 2001)

O modelo NCD (Figura 12) é composto por três partes (Koen *et al.*, 2001; Belliveau, Griffin and Somermeyer, 2004):

- O motor (*engine* ou parte central do modelo) é responsável pelos principais aspetos empresariais incluindo a liderança, cultura e estratégia da organização, e conduz os cinco elementos-chave que compõem o FEE
- A área interna define os cinco elementos-chave do FFE: identificação de oportunidade, análise de oportunidade, génese e enriquecimento de ideias, seleção de ideias, e definição de conceitos e tecnologia
- Os fatores de influência são maioritariamente de natureza externa – política governamental, clientes, concorrentes, clima político e económico, entre outros. Afetam todo o processo de inovação, incluindo o FEE, o NPD e a Comercialização, e são relativamente incontroláveis pela organização

3.1.2 Identificação de Oportunidade

Esta fase é habitualmente guiada pelos objetivos definidos por uma organização, e é aqui que a mesma identifica as oportunidades comerciais e tecnológicas que poderá seguir, de uma forma mais ou menos estruturada. Uma oportunidade varia deste uma direção totalmente nova para o negócio até uma pequena atualização de um produto existente (Koen *et al.*, 2001).

As fontes e métodos utilizados pela organização podem variar desde ferramentas formais e sistemáticas, tais como o mapeamento de cenários futuros ou métodos de resolução de problemas, até abordagens menos formais, tais como conversas e perceções individuais (Dewulf, 2013).

Num estudo realizado em 2020 pela plataforma O'Reilly (Loukides and Swoyer, 2020) é possível perceber que, nos últimos anos, cada vez mais organizações, equipas e desenvolvedores têm vindo a adotar a arquitetura baseada em microsserviços para o desenvolvimento das suas soluções de software. Para além disso, e apesar da adoção de REST continuar a dominar, o interesse em tecnologias como gRPC tem vindo a aumentar ("State of APIs Developer Survey 2021 Report," 2021), visto esta resolver algumas das questões mais prominentes no que toca à comunicação entre microsserviços.

Tendo em conta o contexto apresentado, surge a oportunidade de explorar a tecnologia gRPC, e estudar como esta se compara com outras alternativas de maior popularidade em determinados cenários de comunicação dentro de aplicações baseadas em microsserviços.

3.1.3 Análise de Oportunidade

Nesta fase, a oportunidade é avaliada de forma a garantir que vale a pena dar seguimento à mesma. Para isso é necessária informação adicional como estudos de mercado, experiências ou pesquisas. Esta informação deve ser analisada para que a oportunidade identificada seja traduzida em oportunidades comerciais e tecnológicas específicas para a organização (Koen *et al.*, 2001).

Tendo em consideração o facto de que gRPC é ainda algo recente (como mencionado anteriormente, foi publicado em 2015) torna-se difícil compreender, através de dados concretos, qual o valor que a *framework* realmente apresenta. Num inquérito realizado pela organização RapidAPI a cerca de 1500 utilizadores da sua plataforma, observou-se que, no ano de 2021 ("State of APIs Developer Survey 2021 Report," 2021), 5,4% dos inquiridos usa atualmente gRPC em ambiente de produção – este valor representa um aumento dos 4,1% que tinham sido observados no ano anterior ("Developer Survey 2020 Report," 2020), o que parece indicar um acréscimo interesse e adoção da tecnologia.

Focando agora na relevância da arquitetura baseada em microsserviços, a plataforma O'Reilly realizou, no início de 2020, um inquérito a cerca de 1500 dos seus utilizadores com o objetivo de averiguar em que nível se encontra atualmente a adoção desta arquitetura (Loukides and

Swoyer, 2020). Ao analisar os resultados, é possível observar que apenas cerca de 23% dos inquiridos não usam microsserviços dentro da usa organização e que, para além disso, não só uma grande parte dos inquiridos usa microsserviços, como 61% destes adotou a abordagem há um ano ou mais (à data do inquérito). É assim possível concluir que, atualmente, a abordagem desta arquitetura possui grande relevância para o desenvolvimento de software.

Neste momento, já existem alguns casos de estudo de sucesso de adoção de gRPC, compilados pela Cloud Native Computing Foundation (*Case studies | Cloud Native Computing Foundation*, 2022), que reportam ganhos como melhor performance, eficiência, fiabilidade, aumento da produtividade das equipas de desenvolvimento, entre outros. Alguns destes casos de estudo incluem organizações de grande relevância como Netflix, Spotify, New York Times, Nokia, entre outras, o que geralmente contribui para a promoção da tecnologia, e que pode eventualmente resultar numa maior adoção da mesma e em melhorias a longo prazo.

A relevância que a arquitetura baseada em microsserviços apresenta atualmente, as possibilidades oferecidas por gRPC e o seu crescimento, e a adoção com sucesso desta tecnologia por parte de algumas organizações, faz com que o caminho lógico a seguir seja o estudo da tecnologia num contexto de aplicações baseadas em microsserviços.

3.1.4 Génese e Enriquecimento de Ideias

A geração e enriquecimento da ideia diz respeito ao nascimento, desenvolvimento e amadurecimento de uma ideia concreta. Durante este processo (que é iterativo e evolutivo) ideias são construídas, demolidas, combinadas, reformuladas, modificadas e melhoradas (Koen *et al.*, 2001).

Após a análise da oportunidade, surge então a necessidade de refletir sobre a mesma e discutir ideias que se alinhem com os objetivos definidos. Como fruto desta reflexão surgiram as seguintes ideias:

- Desenvolver aplicações protótipo que permitam estudar algumas das capacidades de gRPC, e o seu uso em certos cenários de comunicação entre serviços, comparando com o estilo REST
- Estudar projetos *open-source* que recorram a gRPC e ao estilo REST para comunicação entre serviços
- Avaliar o desenvolvimento de uma aplicação baseada em microsserviços realizado por terceiros através de inquéritos (onde diferentes aplicações recorrem a diferentes abordagens à comunicação)

3.1.5 Seleção de Ideias

Numa organização, o problema, de um modo geral, não se encontra na geração de ideias – o problema reside geralmente no processo de selecionar quais as ideias a perseguir e quais ideias vão contribuir para o maior valor comercial. O processo de seleção de ideias envolve uma série

iterativa de atividades que, tipicamente, incluem repetidas passagens pela identificação de oportunidades, análise de oportunidades, e geração e enriquecimento de ideias (Koen *et al.*, 2001; Belliveau, Griffin and Somermeyer, 2004).

Decidiu-se então que o foco deveria ser no desenvolvimento de aplicações protótipo e avaliação de cenários de comunicação em aplicações compostas por microsserviços - cada aplicação apresenta os mesmos cenários de comunicação, mas usa diferentes abordagens à implementação da mesma (gRPC e REST).

O processo de seleção de ideia pode ser observado na secção Analytic Hierarchy Process (AHP).

3.1.6 Definição de conceito

Representa o elemento final do modelo NCD e a única saída para o NPD, e envolve o desenvolvimento de um caso de negócio baseado em vários fatores: estimativas sobre o potencial do mercado, necessidades dos vários clientes, requisitos de investimento, avaliações à concorrência, entre outros (Koen *et al.*, 2001).

O objetivo do trabalho é estudar a tecnologia gRPC quando aplicada à comunicação entre microsserviços, e como esta se compara com o estilo REST em cenários de comunicação comuns. Como tal, devem ser desenvolvidos protótipos que permitam estudar os diferentes cenários, e que forneçam informação sobre o comportamento de cada abordagem quando aplicada aos diferentes cenários.

3.2 Valor da Solução

Dado que a presente tese aborda a adoção de uma abordagem específica à comunicação entre microsserviços (gRPC), não existe concretamente um produto ou serviço específico a ser desenvolvido com clientes externos em mente. Dado este contexto, o capítulo atual procura apresentar o valor que é oferecido a potenciais utilizadores desta tecnologia, e a utilizadores que procurem conhecimento sobre abordagens à comunicação entre microsserviços.

3.2.1 Noção de valor

Um produto ou serviço é geralmente considerado como tendo bom valor caso o seu desempenho e custo sejam considerados adequados (esta consideração depende, normalmente, do contexto em que a mesma é realizada) (Miles, 2015).

Qualquer tentativa de potenciar o valor de um produto deve considerar dois elementos (Rich and Holweg, 2000):

- O primeiro surge do uso do produto (conhecido como valor de uso) – resulta da medida das propriedades, qualidades e características que fazem com que o produto

realize um trabalho ou serviço. Este valor é o preço pago pelo comprador ou o custo incorrido pelo fabricante para assegurar que o produto desempenha a sua função pretendida de forma eficiente

- O segundo elemento provém da posse do produto (valor estimado ou valor de estima) – resulta da medida das propriedades e características que aumentam a atração de vendas ou que atraem clientes e criam neles um forte desejo de possuir o produto. Este valor é o preço pelo comprador ou o custo incorrido que vai além do valor de uso

3.2.2 Proposta de Valor

A proposta de valor pode ser entendida como uma visão global do pacote de produtos e serviços que, em conjunto, representam valor para um segmento de clientes específico. A proposta descreve como uma organização/empresa se diferencia dos seus principais concorrentes, e é a razão pela qual os clientes tendem a comprar a uma determinada empresa e não a outra (Osterwalder and Pigneur, 2003).

De forma a analisar o valor da oportunidade identificada e definir a proposta de valor, foi usado o *Value Proposition Canvas*. O *Value Proposition Canvas* apresenta duas áreas principais (Osterwalder *et al.*, 2015):

- **Perfil do (Segmento do) Cliente:** descreve um segmento de cliente específico através de um modelo estruturado e de forma detalhada. Aqui, o cliente é decomposto nos seus trabalhos (*jobs*), dores (*pains*) e ganhos (*gains*)
- **Mapa de Valor (da Proposta)**: descreve as características de uma proposta de valor específica de uma forma mais estruturada e detalhada. Decompõe a proposta de valor em produtos e serviços, aliviadores de dor (*pain relievers*), e criadores de ganhos (*gain creators*)

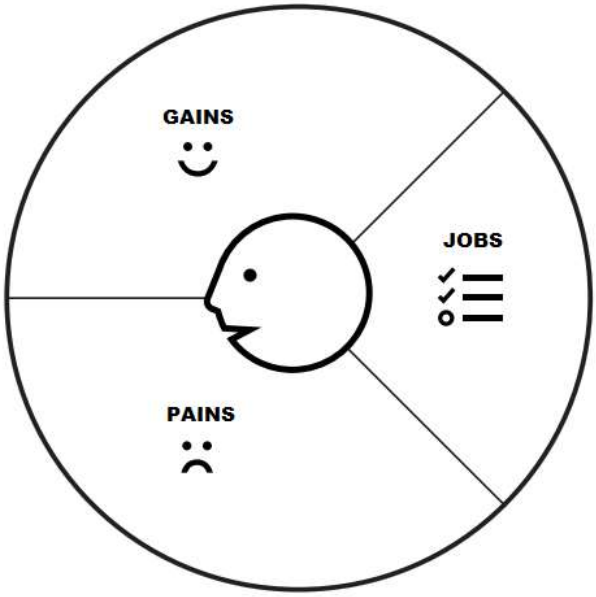

Figura 13 – Perfil do Cliente (Osterwalder *et al.*, 2015)

A Figura 13 representa o Perfil do Cliente, que é dividido três áreas (Osterwalder *et al.*, 2015):

- **Trabalhos:** Descrevem aquilo que os clientes estão a tentar fazer (ou atingir) no seu trabalho ou na sua vida. Um trabalho pode ser uma tarefa que o cliente procura realizar ou completar, um problema que procura resolver, ou uma necessidade que procura satisfazer. Existem quatro tipos de trabalho: funcional, social, pessoal/emocional, e de apoio
- **Dores:** Descrevem tudo o que incomoda/irrita os clientes antes, durante e depois da realização de determinada tarefa, ou algo que os impede de completar determinada tarefa. Habitualmente, procura-se identificar três tipos de dores: resultados indesejados ou problemas com o resultado (uma solução não funciona ou apresenta problemas); obstáculos (impedem de completar ou até de começar um determinado trabalho); riscos (aquilo que pode correr mal e apresentar consequências negativas)
- **Ganhos:** Descrevem os resultados e benefícios que os clientes procuram obter. Podem ser divididos em quatro tipos: obrigatórios, necessários, desejados e inesperados

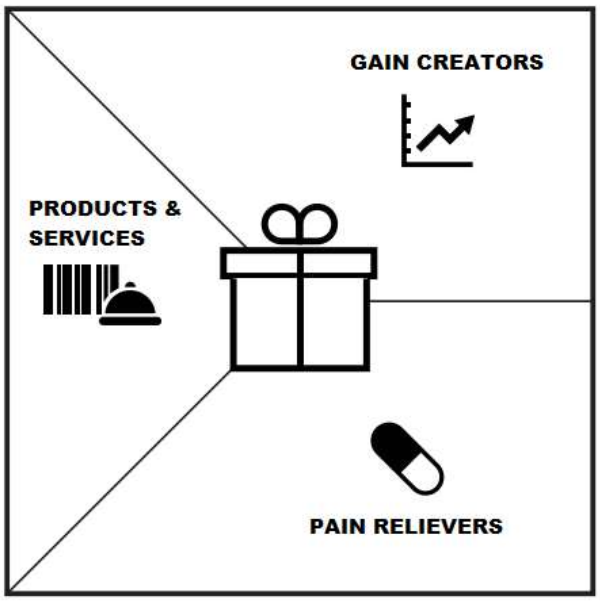

Figura 14 – Mapa de Valor (Osterwalder *et al.*, 2015)

A Figura 14 representa o Mapa de Valor, que é também dividido em três áreas (Osterwalder *et al.*, 2015):

- **Produtos e Serviços:** Enumeração de todos os produtos e serviços que o negócio espera oferecer aos seus clientes e em que a proposta de valor se baseia
- **Aliviadores de Dor:** Descreve exatamente como o conjunto de produtos e serviços irão aliviar as "dores" do cliente. Delineiam como se pretende eliminar ou reduzir alguns dos problemas enfrentados pelos clientes antes, durante ou após a realização de determinado trabalho
- **Criadores de Ganhos:** Descreve como o conjunto de produtos e serviços criará ganhos. Detalham como se pretende produzir os resultados e benefícios que o cliente espera, deseja, ou ficaria surpreendido com

Com os conceitos base em mente é possível proceder à aplicação do *Canvas* a este trabalho (Figura 15).

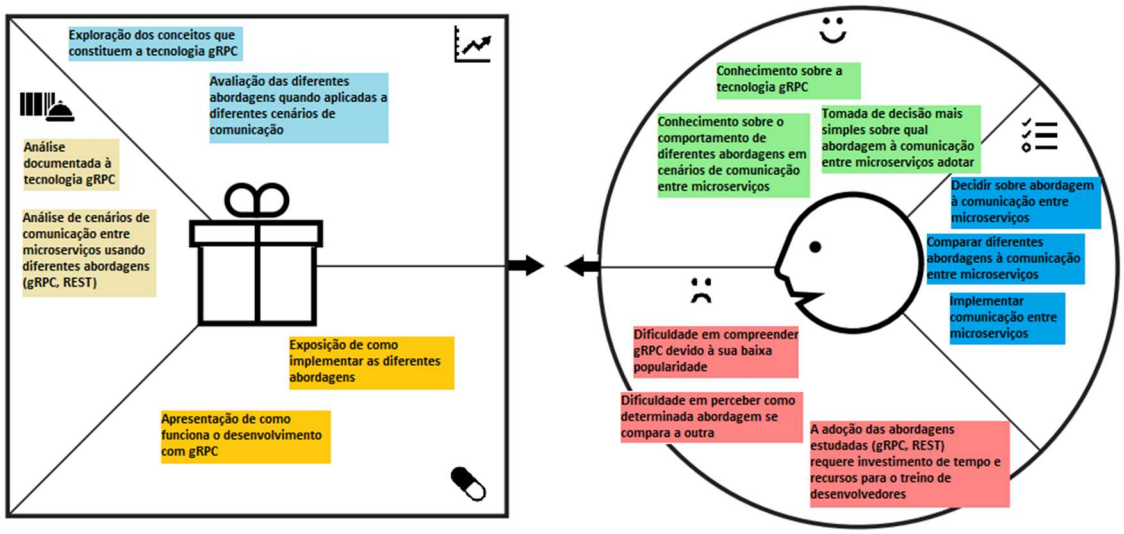

Figura 15 – *Value Proposition Canvas*

Tendo em mente o facto de que este trabalho procura explorar a tecnologia gRPC e estudar a aplicação de diferentes abordagens a cenários de comunicação entre microsserviços, o *Value Proposition Canvas* apresentado (Figura 15) acaba por ser ligeiramente fora do comum, uma vez que não existe, concretamente, um produto ou serviço a ser proposto.

Analisando com mais detalhe o *Value Proposition Canvas* proposto, é possível concluir que os principais aspetos do mesmo são:

- O produto trata-se de uma exploração documentada da tecnologia gRPC, que analisa cenários de comunicação tendo em conta outra abordagem – o estilo arquitetural REST
- Espera-se trazer conhecimento sobre as abordagens mencionadas e a sua utilização na comunicação entre microsserviços aos consumidores do documento final
- Espera-se trazer exemplos práticos da adoção das diferentes abordagens, o que pode servir como uma introdução inicial às mesmas, potencialmente agilizando a sua adoção por parte dos consumidores
- A exploração de diferentes cenários de comunicação auxilia os consumidores na sua tomada de decisão sobre que abordagem seguir

3.2.3 Analytic Hierarchy Process (AHP)

Como apresentado na secção Génese e Enriquecimento de Ideias, existem três diferentes alternativas que foram propostas para o estudo que se procura realizar, existindo assim a necessidade de se encontrar qual opção deve ser seguida. Para isso, será usado o método AHP.

AHP é um método, desenvolvido por Thomas L. Saaty (Saaty, 1990), usado para auxiliar na organização e análise de decisões complexas, e na resolução de problemas técnicos e de gestão. A principal vantagem deste método é que organiza fatores tangíveis e intangíveis de forma

sistemática, e fornece uma solução estruturada, mas relativamente simples, aos problemas de tomada de decisão (Palcic and Lalic, 2009).

AHP é composto pelos seguintes passos (Saaty, 1990):

- **Desenvolvimento de uma hierarquia de decisão:** Este representa o primeiro passo deste processo. O objetivo principal deste passo é a desconstrução do problema numa hierarquia, mais concretamente numa árvore hierárquica onde:
    - O primeiro nível (ou nível superior) apresenta o principal objetivo do processo
    - O segundo nível apresenta os critérios que contribuem para a avaliação
    - O terceiro nível (ou nível inferior) contém as alternativas consideradas
- **Comparação das alternativas e critérios:** Neste passo, devem ser atribuídas propriedades aos critérios definidos no passo anterior, através de uma matriz de comparação. Saaty define uma escala de nove valores para comparação a fim de manter a matriz consistente.
- **Definição da prioridade relativa de cada critério:** Esta fase passa por estabelecer as prioridades de cada alternativa relativamente aos critérios definidos e termina com o cálculo do vetor de prioridades
- **Avaliação de consistência:** Envolve o cálculo do Índice de consistência e da Razão de Consistência (RC) para medir o quão as estimativas obtidas foram consistentes
- **Construção da matriz de comparação em pares para cada critério de acordo com cada alternativa:** A construção da matriz e a determinação da prioridade relativa de cada critério são realizados novamente, tendo agora em conta a importância de cada alternativa
- **Cálculo das prioridades compostas:** Para encontrar a prioridade composta para cada alternativa, deve ser multiplicado o peso do critério pelo seu correspondente peso normalizado
- **Escolha da alternativa**

A. Desenvolvimento de uma hierarquia de decisão

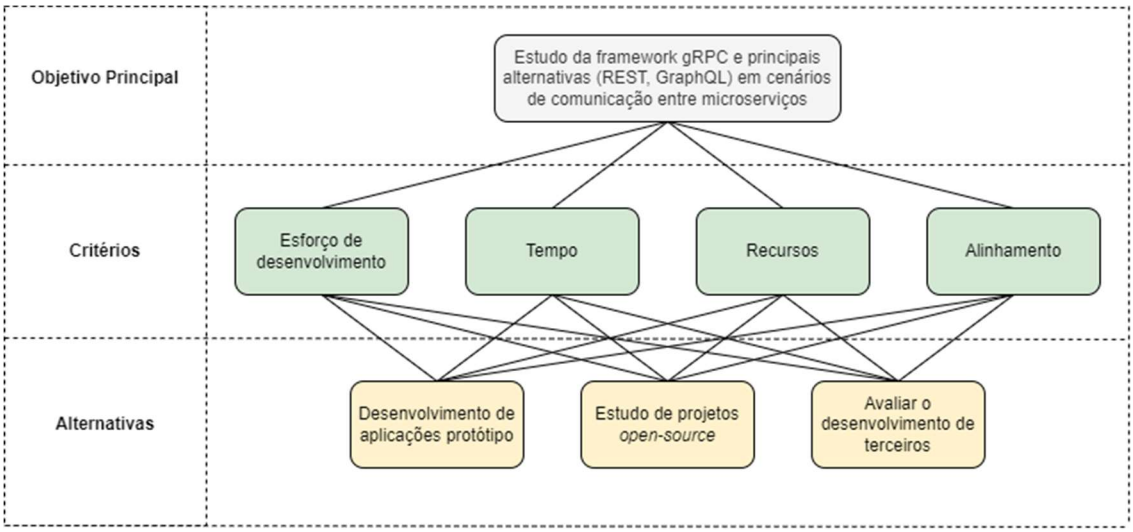

Figura 16 – Árvore hierárquica

A Figura 16 apresenta a árvore hierárquica desenvolvida para o primeiro passo do processo de seleção de ideia. Como é possível observar, e como mencionado anteriormente, a árvore é composta por três níveis, onde é apresentado o objetivo principal do processo, os critérios que vão levar à seleção de uma das alternativas, e as alternativas consideradas.

Aqui, importa destacar os critérios considerados:

- **Esforço de desenvolvimento:** O esforço de desenvolvimento engloba todo o trabalho necessário para a realização do projeto. Este critério será avaliado de acordo com estimativas de esforço visto existir alguma dificuldade em determinar o esforço envolvido na adoção de qualquer alternativa com exatidão
- **Tempo:** Este critério considera o tempo que se espera que cada alternativa necessite para a sua realização
- **Recursos:** Aqui engloba-se tudo o que se considera como necessário para a construção e desenvolvimento da alternativa. Exemplos de recursos são pessoas, ferramentas, materiais, entre outros
- **Alinhamento:** Aqui considera-se o quão a alternativa se enquadra com os objetivos do trabalho

É importante também realçar que, para facilitar a compreensão de algumas tabelas, as alternativas serão referenciadas da seguinte forma:

- **Alternativa A:** Desenvolvimento de aplicações protótipo
- **Alternativa B:** Estudo de projetos *open-source*
- **Alternativa C:** Avaliar o desenvolvimento de aplicações por parte de terceiros

B. Comparação das alternativas e critérios e cálculo da prioridade relativa

Tabela 3 – Escala fundamental (Saaty, 1990)

Nível de importância	Definição	Explicação
1	Igual importância	As duas atividades contribuem igualmente para o objetivo
3	Fraca importância	A experiência e o julgamento favorecem levemente uma atividade em relação à outra
5	Forte importância	A experiência e o julgamento favorecem fortemente uma atividade em relação à outra
7	Muito forte importância	Uma atividade é muito fortemente favorecida em relação a outra
9	Importância absoluta	A evidência favorece uma atividade em relação a outra com o mais alto grau de certeza
2, 4, 6, 8	Valores intermediários	Quando se procura uma condição de compromisso entre duas definições

A Tabela 3 representa a escala fundamental proposta por Saaty para a definição de prioridades dos critérios.

Tabela 4 – Matriz de Comparação

	Esforço de desenvolvimento	Tempo	Recursos	Alinhamento
Esforço de desenvolvimento	1	1/2	1/8	1/8
Tempo	2	1	1/6	1/3
Recursos	8	6	1	2
Alinhamento	8	3	1/2	1
Soma	19	21/2	43/24	83/24

Tabela 5 – Matriz Normalizada e Prioridade Relativa

	Esforço de desenvolvimento	Tempo	Recursos	Alinhamento	Prioridade Relativa
Esforço de desenvolvimento	0.0526	0.0476	0.0698	0.0361	0.051541
Tempo	0.1053	0.0952	0.0930	0.0964	0.097478
Recursos	0.4211	0.5714	0.5581	0.5783	0.532233
Alinhamento	0.4211	0.2857	0.2791	0.2892	0.318748

A Tabela 4 apresenta a matriz de comparação desenvolvida, onde é possível observar a importância que os critérios têm quando comparados entre si. Com base nestes valores, foi desenvolvida a matriz normalizada e calculado o vetor de prioridade (ou vetor próprio) (Tabela 5). O resultado observado na Tabela 5 evidência a prioridade dos critérios: Recursos > Alinhamento > Tempo > Esforço de desenvolvimento.

C. Avaliação de consistência

De modo a avaliar a consistência das prioridades relativas obtidas no ponto anterior, é necessário calcular o Índice de Consistência (IC) e a Razão de Consistência (RC). A consistência de prioridades indica a existência de consistência no julgamento dos critérios ao longo da definição de prioridades (o julgamento é confiável).

Para calcular a RC é necessário primeiro obter o valor de $\mathrm{l}_{\max}$. Este valor representa o maior valor próprio da matriz de comparação (matriz A) e é obtido a partir da seguinte equação:

$$\mathrm{Ax} = \mathrm{l}_{\max}\,\mathrm{x} \tag{1}$$

Uma vez obtido o valor de $\mathrm{l}_{\max}$, é possível calcular o IC (onde η representa o nº de critérios):

$$IC = \frac{\mathrm{l}_{\max} - \eta}{\eta - 1} \tag{2}$$

Após se obter o valor do IC, é possível calcular a RC:

$$RC = \frac{IC}{IR} \tag{3}$$

O Índice Aleatório (IR) trata-se de um valor tabelado (Tabela 6).

Tabela 6 – Valores do Índice Aleatório

η	1	2	3	4	5	6	7	8	9
IR	0	0	0.58	0.90	1.12	1.24	1.32	1.41	1.45

De seguida, é então apresentado o cálculo do valor de $\mathrm{l}_{\max}$:

$$\begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{8} & \frac{1}{8} \\ 2 & 1 & \frac{1}{6} & \frac{1}{3} \\ 8 & 6 & 1 & 2 \\ 8 & 3 & \frac{1}{2} & 1 \end{bmatrix} \times \begin{bmatrix} 0.051541 \\ 0.097478 \\ 0.532233 \\ 0.318748 \end{bmatrix} = \begin{bmatrix} 0.206652 \\ 0.395514 \\ 2.166921 \\ 1.289623 \end{bmatrix} \quad (4)$$

$$\mathrm{l}_{\max} = \frac{\frac{0.206652}{0.051541} + \frac{0.395514}{0.097478} + \frac{2.166921}{0.532233} + \frac{1.289623}{0.318748}}{4} = 4.046064 \quad (5)$$

Com este valor, é então possível calcular o IC e a RC (onde IR = 0.9 dado que η = 4):

$$IC = \frac{4.046064 - 4}{4 - 1} = 0.015355 \quad (6)$$

$$RC = \frac{0.015355}{0.9} = 0.017061 \quad (7)$$

Tendo em conta que o valor obtido para RC, podemos considerar que os valores das prioridades relativas são consistentes.

D. Construção da matriz de comparação em pares para cada critério de acordo com cada alternativa

De seguida são apresentadas várias matrizes de comparação entre as diferentes alternativas para os diferentes critérios:

- **Esforço de Desenvolvimento:** Tabela 7 e Tabela 8
- **Tempo:** Tabela 9 e Tabela 10
- **Recursos:** Tabela 11 e Tabela 12
- **Alinhamento:** Tabela 13 e Tabela 14

Tabela 7 – Matriz de Comparação (Esforço de desenvolvimento)

Esforço de desenv.	Alt. A	Alt. B	Alt. C
Alternativa A	1	1/3	1/5
Alternativa B	3	1	1/3
Alternativa C	5	3	1
Soma	9	13/3	23/15

Tabela 8 – Matriz Normalizada e Prioridade Relativa (Esforço de desenvolvimento)

Esforço de desenv.	A	B	C	Prioridade Relativa
Alternativa A	0.1111	0.0769	0.1304	0.1062
Alternativa B	0.3333	0.2308	0.2174	0.2605
Alternativa C	0.5556	0.6923	0.6522	0.6333

Seguindo o mesmo processo apresentado anteriormente para o cálculo do RC, obteve-se 0.019357, ou seja, os valores apresentados são consistentes.

Tabela 9 – Matriz de Comparação (Tempo)

Tempo	Alt. A	Alt. B	Alt. C
Alternativa A	1	1/3	1/5
Alternativa B	3	1	1/3
Alternativa C	5	3	1
Soma	9	13/3	23/15

Tabela 10 – Matriz Normalizada e Prioridade Relativa (Tempo)

Tempo	A	B	C	Prioridade Relativa
Alternativa A	0.1111	0.0769	0.1304	0.1062
Alternativa B	0.3333	0.2308	0.2174	0.2605
Alternativa C	0.5556	0.6923	0.6522	0.6333

Como a avaliação é semelhante à anterior, também aqui se obteve um RC igual a 0.019357, ou seja, os valores são consistentes.

Tabela 11 – Matriz de Comparação (Recursos)

Recursos	Alt. A	Alt. B	Alt. C
Alternativa A	1	2	7
Alternativa B	1/2	1	5
Alternativa C	1/7	1/5	1
Soma	23/14	16/5	13

Tabela 12 – Matriz Normalizada e Prioridade Relativa (Recursos)

Recursos	A	B	C	Prioridade Relativa
Alternativa A	0.6087	0.6250	0.5385	0.5907
Alternativa B	0.3043	0.3125	0.3846	0.3338
Alternativa C	0.0870	0.0625	0.0769	0.0755

Para os valores apresentados nas tabelas Tabela 11 e Tabela 12, foi chegou-se a um RC de 0.007088, o que significa que os valores apresentados são consistentes.

Tabela 13 – Matriz de Comparação (Alinhamento)

Alinhamento	Alt. A	Alt. B	Alt. C
Alternativa A	1	2	5
Alternativa B	1/2	1	3
Alternativa C	1/5	1/3	1
Soma	17/10	10/3	9

Tabela 14 – Matriz Normalizada e Prioridade Relativa (Alinhamento)

Alinhamento	A	B	C	Prioridade Relativa
Alternativa A	0.5882	0.6000	0.5556	0.5813
Alternativa B	0.2941	0.3000	0.3333	0.3092
Alternativa C	0.1176	0.1000	0.1111	0.1096

Para as matrizes relacionadas com o critério Alinhamento (Tabela 13 e Tabela 14) obteve-se um RC de 0.001848, o que significa que os valores propostos são consistentes.

E. Cálculo das prioridades compostas e escolha da alternativa

Para obter as prioridades compostas das alternativas é necessário multiplicar a matriz de prioridade (construída com as prioridades relativas apresentadas nas tabelas anteriores), pelo vetor que define os pesos dos diferentes critérios (valores presentes na coluna Prioridade Relativa da Tabela 5):

$$\begin{matrix} Alt.A \\ Alt.B \\ Alt.C \end{matrix} \begin{bmatrix} 0.1062 & 0.1062 & 0.5907 & 0.5813 \\ 0.2605 & 0.2605 & 0.3338 & 0.3092 \\ 0.6333 & 0.6333 & 0.0755 & 0.1096 \end{bmatrix} \times \begin{bmatrix} 0.051541 \\ 0.097478 \\ 0.532233 \\ 0.318748 \end{bmatrix} = \begin{bmatrix} 0.515504 \\ 0.315036 \\ 0.169492 \end{bmatrix} \quad (8)$$

A alternativa com maior prioridade representa o resultado final do processo AHP. De acordo com os resultados obtidos, a Alternativa A, que representa o desenvolvimento de aplicações protótipo, é a mais adequada para o projeto.

4 Análise e Desenho

Este capítulo apresenta uma descrição detalhada das soluções propostas para o estudo de diferentes alternativas para a comunicação entre microsserviços, nomeadamente a adoção da *framework* gRPC e do estilo arquitetural REST.

4.1 Introdução

Uma vez que o objetivo principal deste trabalho é observar a adoção de cada uma das alternativas estudadas em cenários de comunicação entre microsserviços, e onde estas se adequam, existe a necessidade de desenhar e desenvolver uma aplicação que implique a existência de vários contextos de domínio e que seja composta por vários serviços. Estes serviços devem apresentar uma responsabilidade clara e bem definida, as fronteiras entre estes devem estar bem estabelecidas, e devem existir relações de comunicação entre os mesmos. As relações de comunicação entre os serviços devem permitir colocar em prática os cenários estabelecidos no decorrer do Contexto e Estado da Arte, para que estes possam ser estudados.

Tendo em consideração as imposições definidas, foram consideradas duas opções para a seleção de um ponto de partida para o desenho da solução:

- Definir um tema de contextualização das aplicações protótipo a desenvolver e a partir daqui estabelecer um conjunto de requisitos e o desenho da aplicação
- Seguir (ou adaptar) o desenho de uma aplicação apresentada num livro/artigo/projeto *open-source* composta por microsserviços que apresentasse uma série de cenários de comunicação relevantes para este trabalho. Este desenho seria posteriormente usado para o desenvolvimento das aplicações protótipo

Após estudo, análise e desenvolvimento inicial da primeira opção (que tinha como tema base o desenvolvimento de uma aplicação de uma loja online), chegou-se à conclusão de que o desenho definido não potencializava alguns dos cenários de comunicação estabelecidos. Com isto em mente, decidiu-se então definir como ponto de partida uma proposta já estabelecida

de uma aplicação composta por microsserviços – a *Drone Delivery App*, que será explorada com maior detalhe na secção 4.2.

Neste capítulo, é apresentada a *Drone Delivery App*, bem como aos conceitos de domínio que serão considerados para este trabalho, e o desenho da solução que se procura desenvolver.

4.2 Drone Delivery App

Esta secção tem como objetivo apresentar a *Drone Delivery App*. Para isso é realizada uma curta introdução onde se detalha a origem deste projeto, seguida do cenário que motivou o desenvolvimento do mesmo, e finalmente uma conclusão que expõe os motivos que levaram à escolha deste projeto e algumas considerações sobre o uso do mesmo neste trabalho.

4.2.1 O que é

A *Drone Delivery App* é um projeto referência, apresentado na documentação para o *Azure Architecture Center* (*Azure Architecture Center | Microsoft Docs*, 2022), que tem como objetivo expor um conjunto de práticas para a construção e execução de uma arquitetura de microsserviços. Ao longo da documentação o projeto é apresentado através de uma série de artigos que descrevem os principais desafios, decisões e padrões de design.

4.2.2 Cenário

A Fabrikam Inc. é uma empresa prestes a iniciar um serviço de entrega através da sua frota de drones. Outras empresas registam-se com o serviço, e os utilizadores podem solicitar um drone para recolha da mercadoria para entrega. Quando um cliente agenda uma recolha, um drone é atribuído à recolha e o utilizador é notificado com um tempo estimado para entrega. Enquanto a entre está em curso, o cliente podem acompanhar a localização do drone, com o tempo estimado para entrega continuamente atualizado (*Design a microservices architecture - Azure Architecture Center*, 2022).

Em geral, o cenário envolve o agendamento de drones, o seguimento de pacotes, a gestão de contas de utilizadores, e o armazenamento e análise de dados históricos. Para além disto, a Fabrikam quer chegar rapidamente ao mercado e depois iterar rapidamente, acrescentado novas funcionalidades e capacidades. A Fabrikam espera também que diferentes partes do sistema tenham requisitos muito diferentes para o armazenamento e consulta de dados. Todas estas considerações levam a Fabrikam a escolher uma arquitetura baseada em microsserviços como base para a *Drone Delivery App* (*Design a microservices architecture - Azure Architecture Center*, 2022).

Tendo por base este cenário, espera-se que, para este trabalho, a aplicação permita que:

- Um cliente possa requisitar um drone para recolha de mercadorias

- O remetente da embalagem seja capaz de gerar uma etiqueta (código de barras ou RFID) para colocar na mesma
- Um drone recolha um pacote no local de origem e o entregue no local de destino
- Quando um cliente marca uma entrega, o sistema fornece um tempo previsto de chegada
- Até que um drone tenha levantado o pacote, o cliente possa cancelar a entrega.
- Um cliente seja notificado quando a entrega requisitada é concluída
- Seja possível solicitar uma confirmação de entrega ao cliente, sob a forma de uma assinatura ou impressão digital
- Utilizadores possam consultar o histórico de uma entrega concluída

De modo a agilizar o desenvolvimento deste sistema, para este trabalho, são assumidas algumas concessões:

- Não serão desenvolvidas funcionalidades relacionadas com a gestão de utilizadores (exceto as necessárias à implementação de uma das funcionalidades listadas anteriormente)
- O primeiro aponto aplicar-se-á também à gestão de drones

4.2.3 A escolha do projeto

Como descrito anteriormente, a escolha do projeto *Drone Delivery App* como ponto de partida ao desenho de uma solução para este trabalho foi tomada como alternativa ao desenvolvimento de uma aplicação de compras online. Esta decisão apresenta como principais motivos:

- O facto de a arquitetura proposta para esta aplicação contemplar vários microsserviços – explorado na secção 4.4
- A existência de múltiplos cenários de comunicação entre os vários serviços
- A facilidade de acesso, devido à documentação existente, às várias decisões tomadas no desenvolvimento do projeto

Embora o desenho proposto para o projeto *Drone Delivery App* se enquadre, em grande parte, com os objetivos delineados para este trabalho, torna-se imprescindível salientar que o mesmo não será seguido integralmente – tendo em conta o contexto do presente trabalho e os objetivos do mesmo, algumas adaptações serão realizadas (e documentadas com justificações).

4.3 Domínio

Na definição dos microsserviços, para além de se procurar obter baixo acoplamento e uma alta coesão funcional, é importante desenvolver os mesmo em torno das capacidades de negócio

necessárias para o correto funcionamento do mesmo (*Domain analysis for microservices - Azure Architecture Center*, 2022).

Para a definição dos microsserviços que compõem a *Drone Delivery App*, é seguida uma estratégia orientada ao design do domínio, visto esta abordagem desempenhar um papel importante no estabelecimento de um conjunto de serviços bem definidos. Esta estratégia compreende os seguintes passos (*Domain analysis for microservices - Azure Architecture Center*, 2022):

1. Análise do domínio do negócio com o objetivo de compreender os requisitos funcionais da aplicação. Daqui deve resultar uma descrição informal do domínio
2. Definição dos *bounded contexts*. Cada *bounded context* contém um modelo de domínio que representa um subdomínio particular da aplicação maior
3. Definição, para cada *bounded context*, definir o domínio
4. Com base nos passos anteriores, identificar os serviços da aplicação

4.3.1 Análise ao domínio

Figura 17 – Análise inicial ao domínio (*Domain analysis for microservices - Azure Architecture Center*, 2022)

A Figura 17 apresenta a análise inicial ao domínio apresentada na documentação do projeto. Este domínio contém (*Domain analysis for microservices - Azure Architecture Center*, 2022):

- ***Shipping* –** representa o núcleo do diagrama. Tudo o resto existe com o objetivo de reforçar esta funcionalidade
- ***Drone management* –** outra área essencial ao negócio. Esta funcionalidade está intimamente relacionada com a **reparação de drones** (*drone repair*) e **análise preditiva** (*predictive analysis*) usada para prever quando um drone necessitará de manutenção
- ***ETA analysis* –** fornece estimativas quanto ao tempo estimado para a recolha e entrega de determinado pacote

- ***Third-Party transportation* –** o transporte por parte de terceiros tem como objetivo permitir o agendamento do transporte de um pacote, caso não seja possível completar o mesmo inteiramente recorrendo a drones
- ***Drone sharing* –** a partilha de drones é entendida como uma possível extensão do negócio principal que tem como objetivo permitir o aluguer de drones excedentários. Esta caraterística é listada como não fazendo parte do lançamento inicial da aplicação
- ***Video surveillance* –** a vigilância por vídeo representa uma outra área para a qual a empresa poderá expandir-se mais tarde
- ***User accounts, Invoicing, Call center* –** contas de utilizadores, faturação e centro de atendimento são subdomínios que servem de suporte ao negócio principal

Embora, e como é possível observar pela análise inicial ao domínio, existam várias áreas de domínio possivelmente relevantes para a construção desta aplicação, nem todas as áreas enunciadas serão exploradas neste trabalho, nomeadamente:

- *Drone repair* e *predictive analysis* – apesar de serem áreas de suporte à gestão de drones, dado o contexto do trabalho e a necessidade de simplificar o desenvolvimento da solução (devido a restrições de tempo/recursos) não serão consideradas
- *Drone sharing* – área definida como não fazendo parte de um lançamento inicial
- *Video surveillance* – área definida como não fazendo parte de um lançamento inicial
- *Invoicing, User Rating, Loyalty* e *Call center* - áreas de suporte à gestão do negócio principal. Desconsideradas, pois isso permite simplificar o desenvolvimento da solução, e porque este trabalho procura apresentar um caso de estudo e não implementar uma solução para um negócio real

4.3.2 Bounded Contexts

Como estabelecido na estratégia delineada para a definição dos serviços, após análise inicial ao domínio é necessário estabelecer os *bounded contexts* considerados para o desenho e desenvolvimento da solução.

O desenho de um modelo de domínio único para todos os serviços (ou até para apenas alguns dos serviços) tende a tornar-se desnecessariamente complexo e torna mais difícil a evolução do mesmo, visto que quaisquer alterações terão de satisfazer as necessidades de todos os subsistemas. Isto leva a que, tipicamente, se concebam modelos separados para cada um dos contextos considerados, onde cada modelo contém apenas as entidades, caraterísticas e atributos que são relevantes dentro do seu contexto particular (*Domain analysis for microservices - Azure Architecture Center*, 2022). Contudo existem entidades que existem em mais do que um contexto – por exemplo, um drone pode existir no contexto de gestão de drones, mas também no contexto da gestão de entregas, apresentando caraterísticas diferentes dependendo do contexto em que se encontra.

Aqui entra o conceito de *bounded context* (traduzido para contexto limitado) - limite dentro de um domínio onde se aplica um determinado modelo de domínio. A Figura 18 apresenta os *bounded contexts* propostos para este domínio.

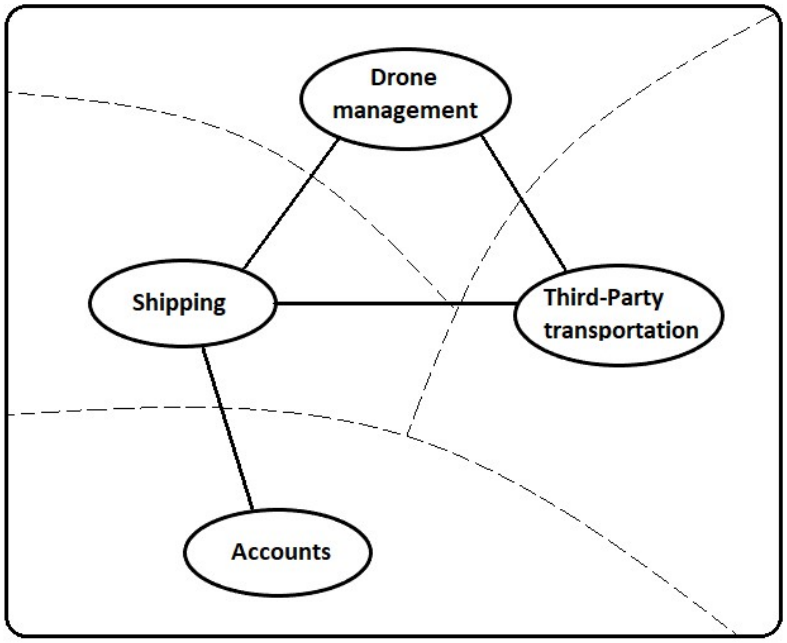

Figura 18 – *Bounded Contexts - Drone Delivery*

4.3.3 Modelo de domínio

A Figura 19, a Figura 20, a Figura 21 e a Figura 22 apresentam o modelo de domínio para cada *bounded context* considerado, e que tem como objetivo responder às necessidades apresentadas pelo cenário exposto anteriormente (apresentado através de várias figuras de modo a facilitar a apresentação). A divisão do domínio por *bounded contexts* auxilia no momento da definição dos microsserviços a desenvolver. É importante mencionar que a documentação apresentada para este projeto (*Using tactical DDD to design microservices | Azure Architecture Center*, 2022) apresenta apenas uma proposta de um desenho orientado ao domínio para o agregado *Delivery*, existindo assim a necessidade de propor um desenho original para as restantes áreas (gestão de contas e gestão de utilizadores).

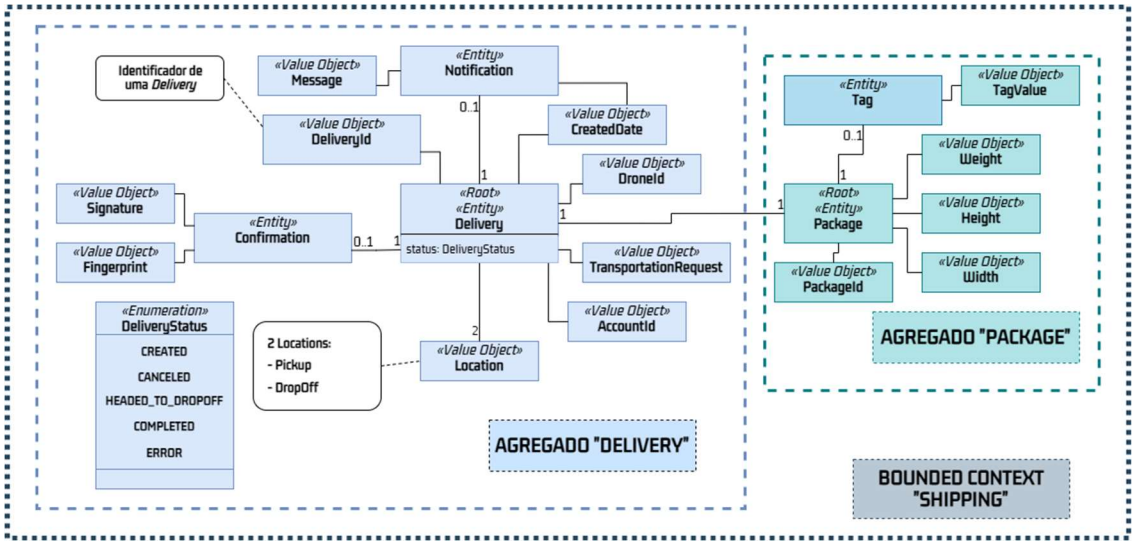

Figura 19 – Representação do *bounded context "Shipping"*

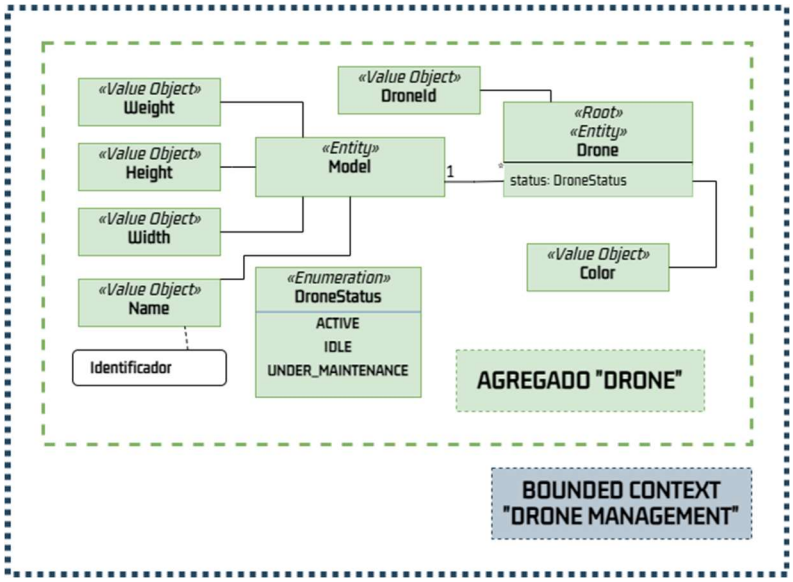

Figura 20 – Representação do *bounded context "Drone Management"*

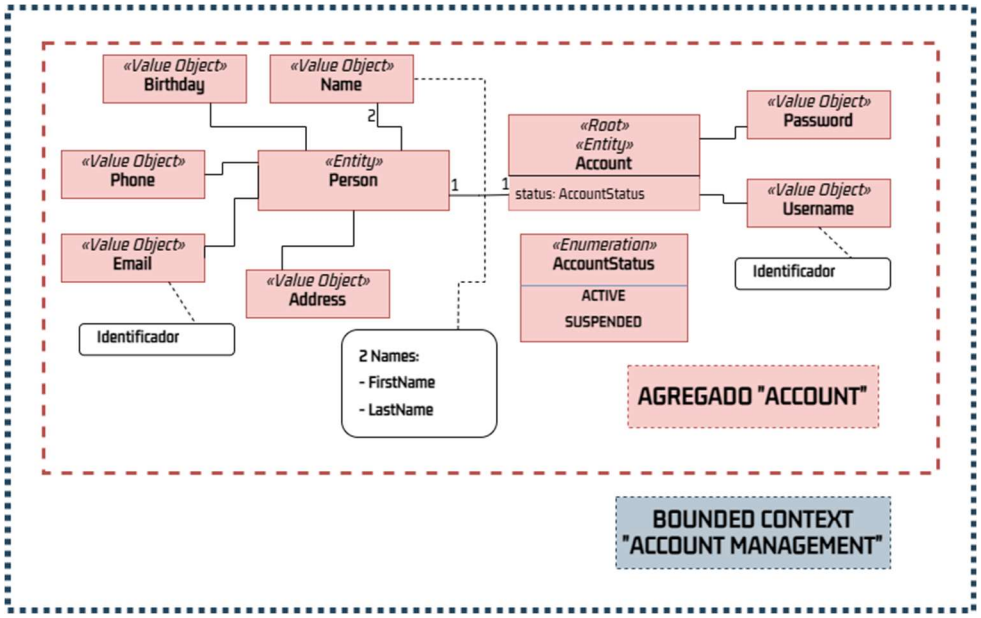

Figura 21 – Representação do *bounded context "Account Management"*

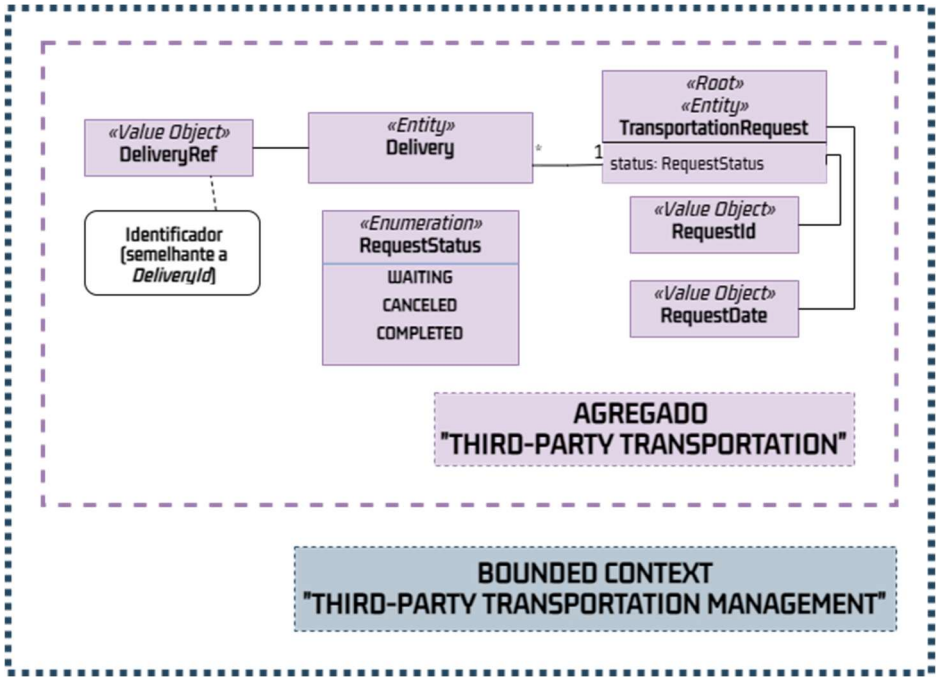

Figura 22 – Representação do *bounded context "Third-Party Transportation Management"*

Algumas notas sobre as figuras apresentadas:

- ***Person*** **–** Conjunto de informações necessários representar uma pessoa no contexto desta aplicação
- ***Account*** **–** Representa o utilizador da *Drone Delivery App*
- ***Delivery*** **–** Representa uma entrega de um pacote e contém um conjunto de informações relevantes para levantamento e entrega do mesmo

- ***Package* –** Representa aquilo que é transportado pelos drones
- ***Confirmation* –** Confirmação (por exemplo assinatura, impressão digital, etc), que é opcional, de que o cliente recebeu uma entrega
- ***Tag* –** Código de barras ou identificador do pacote
- ***Drone* –** Representa um drone dentro da *Drone Delivery App*
- ***Model* –** Conjunto de informações sobre o modelo de um drone
- ***Transportation Request* –** Pedido de entrega a ser realizada por terceiros
- Como apresentado na Figura 18, é sugerida a divisão do *bounded context* associado a *Shipping* nos agregados *Delivery* e *Package* (*Using tactical DDD to design microservices | Azure Architecture Center*, 2022).

4.4 Arquitetura

Decidir como decompor um sistema num conjunto de microsserviços pode ser uma tarefa complexa. Felizmente existem algumas estratégias que podem auxiliar durante este processo, sendo as duas principais a decomposição por capacidade de negócio e a decomposição por subdomínio (Richardson, 2018).

A documentação relativa ao projeto *Drone Delivery App* (*Identify microservice boundaries | Azure Architecture Center*, 2022), sugere ainda a divisão de serviços pelos agregados que compõe o modelo de domínio. Tendo por base esta indicação, foi desenvolvida a arquitetura apresentada na Figura 23.

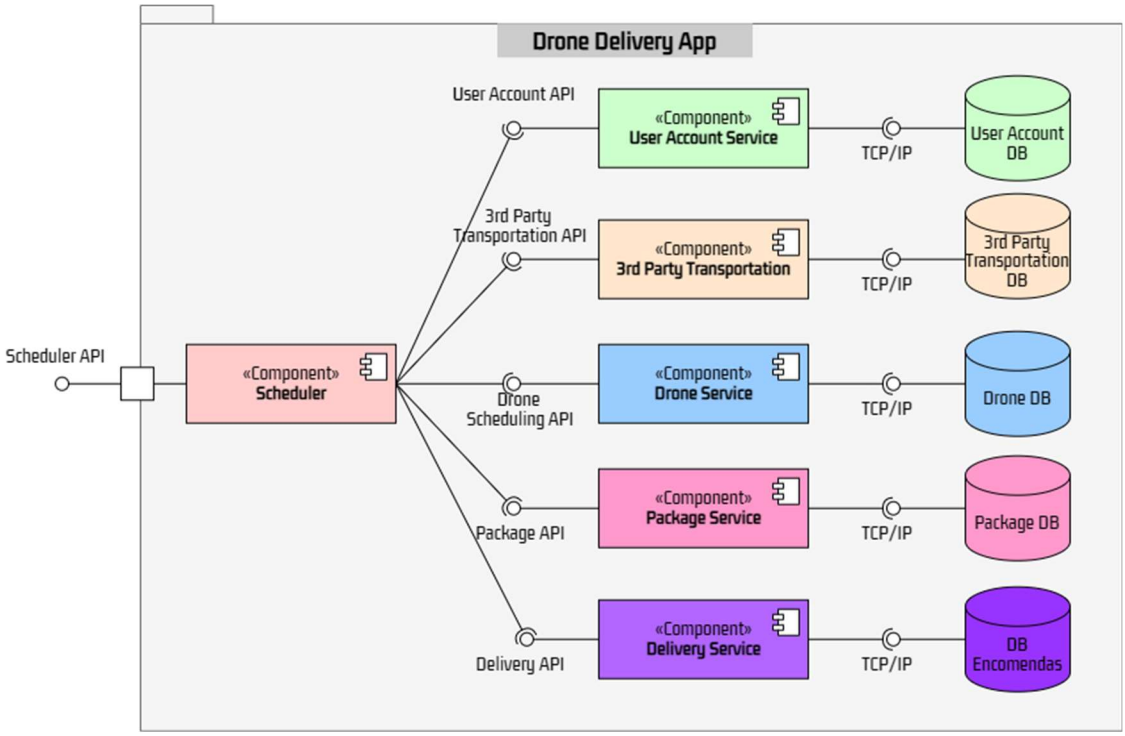

Figura 23 – Arquitetura da *Drone Delivery App*

A arquitetura apresentada é composta pelos seguintes serviços:

- ***Scheduler –*** Responsável por coordenar as operações necessárias ao escalonamento de drones para entrega de pacotes**.**
- ***User Account Service –*** Serviço responsável pela gestão de utilizadores
- ***3rd Party Transportation –*** Serviço que tem como responsabilidade a gestão de pedidos de transporte por terceiros
- ***Drone Service –*** Serviço responsável pela gestão de drones
- ***Package Service –*** Serviço responsável pela gestão de pacotes
- ***Delivery Service –*** Serviço responsável pela gestão de entregas

Uma das caraterísticas da arquitetura proposta é o facto de que cada microsserviço é dono da sua própria base de dados – padrão base de dados por serviço (Richardson, 2022). Apesar de não ser algo que apresente grande relevância para o estudo, é uma prática comum no desenvolvimento de aplicações compostas por microsserviços (Richardson, 2018).

5 Implementação da solução

Este capítulo, tal como o nome indica, tem como objetivo expor a implementação da solução apresentada ao longo do capítulo 4. Para isso, este capítulo fornece uma descrição detalhada dos principais pontos de maior relevo para a implementação da solução, abordando e comparando o desenvolvimento da aplicação que recorre ao estilo REST e da aplicação que adota a *framework* gRPC.

5.1 Linguagens

Um dos objetivos propostos para este trabalho é a investigação da capacidade de comunicação em ambientes poliglotas que as diferentes abordagens oferecem. Para isso foi definido que, o desenvolvimento de cada uma das aplicações deveria recorrer a pelo menos duas linguagens de programação. De forma a estabelecer as linguagens e ferramentas a utilizar para o desenvolvimento, foram analisadas as possibilidades que cada abordagem oferecia.

5.1.1 REST

O desenvolvimento de APIs RESTful é independente de qualquer linguagem de programação – o fator mais relevante na seleção da linguagem/*framework* é a capacidade da ferramenta selecionada suportar HTTP para a realização de pedidos baseados na web (Doyle, Ferguson and McKenzie, 2021).

5.1.2 gRPC

Tal como no desenvolvimento de APIs RESTful, também no desenvolvimento recorrendo à *framework* gRPC existe uma vasta lista de possibilidades, embora não tão extensa, no que toca à linguagem de programação selecionada para a construção de uma solução.

Como mencionado na secção 2.3.2, uma das principais caraterísticas de gRPC é a utilização de buffers de protocolo para a definição da interface de um serviço – esta definição é realizada recorrendo a ficheiros *.proto* que, através de uma ferramenta de compilação chamada *protoc*, são usados para gerar código (numa das várias linguagens suportadas) do lado do servidor e do lado do cliente. A lista atual de linguagens oficialmente suportadas inclui (Chalin and Ranney, 2021):

- C#
- C++
- Dart
- Go
- Java
- Kotlin
- Node
- Objective-C
- PHP
- Python
- Ruby

Para além disso, existem também alguns projetos desenvolvidos pela comunidade que procuram trazer gRPC a outras linguagens (ex.: grpc-swift (*grpc/grpc-swift: The Swift language implementation of gRPC.*, 2022), grpc-rust (*stepancheg/grpc-rust: Rust implementation of gRPC*, 2022), grpc-perl (*joyrex2001/grpc-perl: Perl 5 implementation of gRPC using the official gRPC shared library.*, 2022)).

5.1.3 Conclusão

Na secção 4.4, definiu-se que cada aplicação protótipo seria composta por seis serviços diferentes:

- Scheduler
- User Account Service
- 3rd Party Transportation Service
- Drone Service
- Package Service
- Delivery Service

Tendo por base este plano, estabeleceu-se então a seguinte distribuição tecnológica:

Tabela 15 – Stack tecnológica de cada serviço

Serviço	Stack
Scheduler	Spring Boot
User Account Service	Spring Boot + MySQL
3rd Party Transportation Service	Node.js + MongoDB
Drone Service	Spring Boot + MySQL
Package Service	Node.js + MongoDB
Delivery Service	Node.js + MongoDB

A Tabela 15 permite perceber que *Spring Boot* (*framework open-source* baseada em *Java*) e *Node.js* (ambiente de execução de *JavaScript*) foram selecionadas como as principais tecnologias base para o desenvolvimento da solução. Esta escolha é apoiada pelos seguintes pontos:

- As tecnologias selecionadas possibilitam o desenvolvimento das duas aplicações protótipo consideradas (REST e gRPC – *Java* e *JavaScript* pertencem à lista apresentada anteriormente de linguagens oficialmente suportadas)
- Familiaridade no que toca ao desenvolvimento de aplicações recorrendo a estas tecnologias
- Velocidade de desenvolvimento dada a maturidade e popularidade das tecnologias, facilitando a construção da solução e a resolução de problemas.

5.2 Análise à implementação

Como já declarado, com este trabalho pretende-se, não só, mas principalmente, expor como implementar, recorrendo a gRPC, comunicação numa aplicação composta por vários microsserviços, onde existe necessidade de troca de informação entre os mesmos e como esta se compara com a adoção do estilo REST. Para isso, ao longo deste segmento é explorada a implementação de um dos cenários apresentados anteriormente em 4.2.2: requisitar drone para recolha/entrega de mercadorias – a escolha deste cenário prende-se com o facto da concretização do mesmo envolver comunicação entre vários microsserviços e de este representar a principal funcionalidade da aplicação.

Tendo em conta a proposta desta secção, no decorrer desta parte do documento tenciona-se:

- Apresentar a sequência de comandos que a requisição de um drone para recolha/entrega de mercadoria implica (dentro da aplicação desenvolvida)
- Abordar os principais pontos do desenvolvimento deste cenário através da comparação do trabalho realizado nos dois protótipos, incluindo:

- Definição da interface
- Error Handling (tratamento de erros)
- Comunicação entre os serviços

5.2.1 Requisição de drone para recolha/entrega de mercadoria

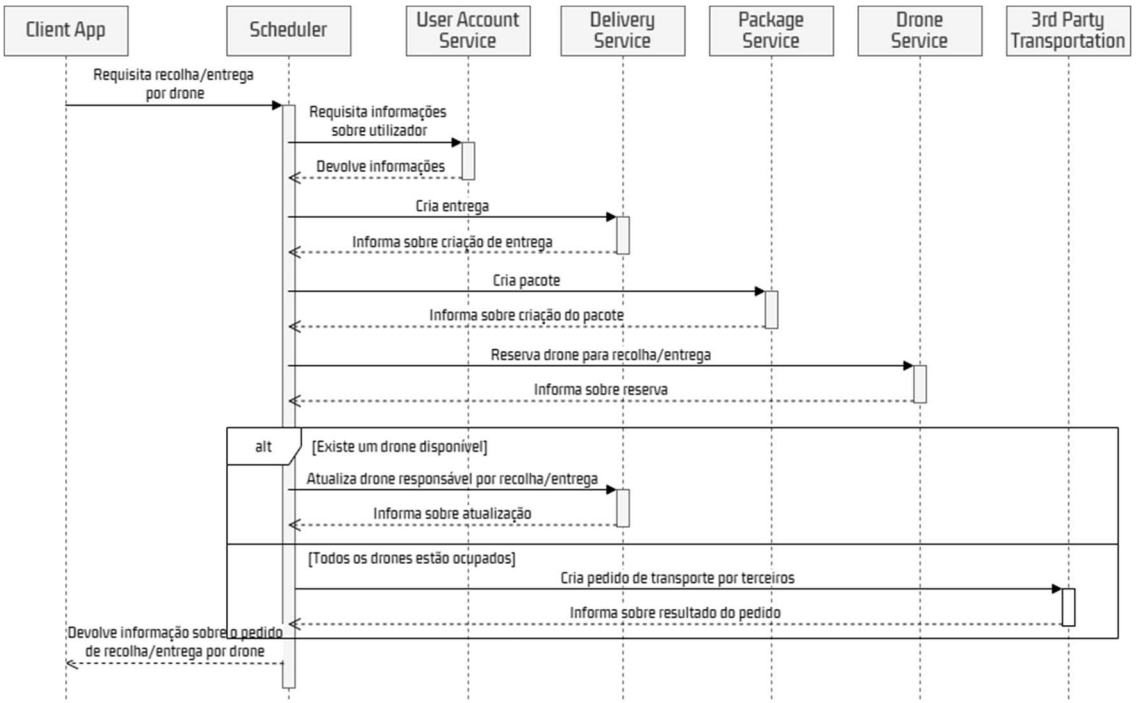

Figura 24 – Diagrama que expõe a sequência de ações realizadas durante a requisição de um drone para recolha/entrega de mercadoria

A Figura 24 apresenta a sequência de pedidos e a lógica envolvida para o processamento de um pedido de requisição de drone. Como é possível observar pela figura, decidiu-se dar maior preferência a uma lógica de orquestração, onde existe um serviço central que orienta e conduz os vários processos, tal como "o maestro de uma orquestra" (Neuman, 2015) (neste caso, *Scheduler* representa este serviço central). O lado negativo de optar por esta abordagem é a existência do risco do serviço central se tornar a principal autoridade governante da lógica da aplicação (Neuman, 2015).

De modo a realizar a comparação entre REST e gRPC, ambos os protótipos desenvolvidos seguem a sequência apresentada.

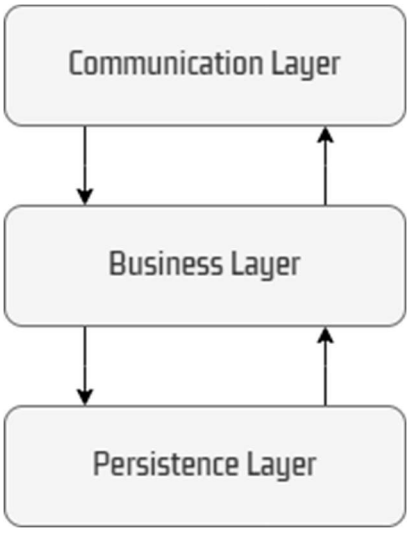

Figura 25 – Divisão por camadas seguida pelos serviços

É relevante também realçar que os serviços responsáveis por determinado domínio (ou seja, todos exceto *Scheduler*) seguem uma arquitetura por camadas semelhante ao apresentado na Figura 25, em ambos os protótipos. Desta forma a única diferença (por exemplo) entre o *User Account Service* que adota o estilo REST e o *User Account Service* que adota gRPC é a camada de comunicação (apresentada na Figura 25 como *Communication Layer*).

Outro aspeto importante da sequência apresentada (Figura 24) é o uso exclusivo de comunicação síncrona – mesmo existindo algumas operações que fariam sentido realizar de forma assíncrona, decidiu-se contra essa opção de modo a apresentar um maior número comunicações onde o desempenho é relevante.

Por último, interessa do mesmo modo mencionar que durante o processamento de um pedido de requisição de um drone, e como é possível visualizar recorrendo à Figura 24, existem várias ocorrências de comunicação entre serviços que empregam diferentes tecnologias (situações que se pretendiam explorar neste trabalho).

5.2.2 Definição da interface

No que se refere à definição de uma API (sobretudo uma API REST) existem essencialmente duas abordagens adotadas (Vasudevan, 2017):

- ***Code First:*** "*Code First*" é uma abordagem mais tradicional à construção de APIs, onde o desenvolvimento do código fonte da aplicação ocorre após a definição dos requisitos empresariais. Aqui, a documentação acaba por ser construída a partir do código. Esta abordagem encontra-se geralmente associada a uma maior rapidez de desenvolvimento e à construção de APIs internas
- ***Design First:*** "*Design First*" defende a conceção do contrato da API antes de se inicializar o desenvolvimento de qualquer código da aplicação. Apesar de representar uma

abordagem relativamente nova, a sua adoção é uma opção viável, principalmente considerando os vários formatos de descrição de API atualmente existentes (OpenAPI, RAML, entre outros). Esta abordagem é frequentemente associada a um desenvolvimento onde existe necessidade de uma fácil compreensão da API por parte dos desenvolvedores, onde estes representam o principal público-alvo

REST

Na implementação da solução proposta para este trabalho decidiu-se começar pela conceção do protótipo que adota REST como estilo arquitetural, seguindo-se uma abordagem de *Code First*. Esta decisão baseou-se não só no facto de existir alguma familiaridade para com a abordagem, mas especialmente pelos pontos mencionados anteriormente – rapidez de desenvolvimento e o não consumo das APIs por outros desenvolvedores.

No geral, a criação de uma API REST tende a passar pelas seguintes fases:

- Definição dos recursos que a aplicação é responsável por gerir
- Criação das representações (ex.: representação JSON ou XML)
- Definir e mapear as ações a realizar sobre os recursos para métodos HTTP e parâmetros de consulta (*query parameters*)

Em relação aos pontos listados acima, é relevante mencionar que durante esta secção do documento será dado maior foco à definição das ações sobre os recursos, abordando ligeiramente a representação dos mesmos.

```
public class SchedulerController {
    private final SchedulerService service;

    @PostMapping("/deliveries")
    public ResponseEntity<ScheduleDeliveryResponseDTO> scheduleDelivery(
            @RequestBody ScheduleDeliveryRequestDTO request) {
        return new ResponseEntity<>(
            service.scheduleDelivery(request), HttpStatus.CREATED
        );
    }

    @PutMapping("/deliveries/{drone}/pickup")
    public ResponseEntity<PickupPackageResponseDTO> pickupPackage(
            @PathVariable(value = "drone") String drone) {
        return new ResponseEntity<>(
            service.pickupPackage(drone), HttpStatus.OK
        );
    }

    (…)
}
```

Código 5 – Extrato do *SchedulerController*

```
export default () => {
  const controller = new DeliveryController();
  const app = Router();

   app.get("/deliveries/:deliveryId", (req, res) => {
     return controller.get(req, res);
  });

  app.patch("/deliveries/:deliveryId", (req, res) => {
    return controller.updateDelivery(req, res);
  });

  app.post("/deliveries", (req, res) => {
    return controller.createDelivery(req, res);
  });
 (…)
}
```

Código 6 – Extrato da definição dos "caminhos" existentes no *Delivery Service*

```
export class DeliveryController {
  constructor() { this.service = new DeliveryService(); }

  async get(req, res) {
    try {
      const delivery = await this.service.get(req.params.deliveryId);
      res.status(200).send(delivery);
    } catch (err) { handleError(res, err); }
  }

  async updateDelivery(req, res) {
    try {
      await this.service.updateDelivery(
        req.params.deliveryId,
        req.body
      );
      res.status(204).send();
    } catch (err) {
      handleError(res, err);
    }
  }

  async createDelivery(req, res) {
    try {
      const delivery = await this.service.createDelivery(req.body);
      res.status(201).send(delivery);
    } catch (err) { handleError(res, err); }
   }
 (…)
}
```

Código 7 – Extrato do *Delivery Controller*

Os trechos de código apresentados acima foram retirados de dois dos serviços mais relevantes para a requisição de um drone (Figura 24): *Scheduler Service* e *Delivery Service*. Código 5 (extraído do *Scheduler Service*) e Código 6 (unido a Código 7, extraídos do *Delivery Service*) representam os controladores – dentro dos serviços desenvolvidos, estas classes representam a forma como aplicações cliente acedem à API REST, contendo a lógica necessária à disponibilização dos recursos e ações oferecidas via HTTP.

Conforme estabelecido na secção 5.1.3, *Scheduler Service* e *Delivery Service* foram desenvolvidos tendo como base diferentes tecnologias: Código 5 representa uma definição típica de um *Controller* em *Spring,* enquanto Código 6 apresenta uma simples definição de rotas semelhante ao habitualmente observado em aplicações que usam a *framework Express* (*framework web* para *Node.js*). Apesar de existir o recurso a diferentes tecnologias, é possível observar um conjunto de pontos que ambos os serviços apresentam em comum:

- **Interação através de URIs:** Cada recurso tem um identificador que o identifica de forma única (URI) – por exemplo, de forma a obter a *delivery* identificada por "1234" em específico, deve ser usado um URI semelhante a "http://delivery-management.com/deliveries/1234"
- **Uso de métodos HTTP:** HTTP dispõe de vários métodos que indicam o tipo de operação que se pretende realizar, sendo da responsabilidade do desenvolvedor da API garantir que uma certa operação é realizada utilizando o método correto. Como regra geral, durante o desenvolvimento foi usado:
    - **GET** para obter recursos (em é usado GET para obter uma *delivery* pelo seu identificador)
    - **POST** para criar recursos (observado em Código 5 e Código 6)
    - **PUT/PATCH** para atualizar recursos (observado em Código 6 para atualizar uma *delivery*)
    - **DELETE** para eliminar recursos existentes
- **Uso de códigos HTTP:** Em APIs REST, códigos HTTP são usados de modo a informar aplicações cliente do resultado do pedido. Este ponto é parcialmente observável em Código 5 e Código 7, onde códigos da família 200 são utilizados para comunicar o sucesso do pedido. O uso de códigos de erro será explorado em 5.2.3
- **Interação através da troca de representações de recursos:** Voltando em parte ao primeiro ponto, a interação com as APIs REST baseia-se na troca de representações de recursos. Por exemplo, em Código 7 um pedido para obter uma *delivery* devolve uma representação baseada em Código 8.

```
export class DeliveryResponseDTO {
  constructor(delivery) {
    this.username = delivery.account;
    this.deliveryId = delivery._id;
    this.pickupCoordinates = delivery.pickup;
    this.dropOffCoordinates = delivery.dropOff;
    this.created = delivery.created;
    this.status = delivery.status;
    this.expedited = delivery.expedited;
    this.drone = delivery.drone ? delivery.drone : null;
    this.transportationRequest = delivery.transportationRequest
      ? delivery.transportationRequest
      : null;
  }
}
```

Código 8 – Representação de *Delivery*

gRPC

Ao contrário do que tinha acontecido no desenvolvimento do protótipo em que os serviços adotam o estilo REST, no desenvolvimento do protótipo que utiliza gRPC foi adotada a abordagem *Design First* – dada a natureza da tecnologia, existe a necessidade de inicializar o desenvolvimento pela definição dos ficheiros *.proto* que são usados para estabelecer os vários serviços gRPC.

```
syntax = "proto3";

package schedulerservice;

import "google/protobuf/timestamp.proto";

option java_multiple_files = true;
option java_package = "pt.isep.tmdei.schedulerservice";

message CreateDeliveryRequest {
  string username = 1;
  int64 pickup_latitude = 2;
  int64 pickup_longitude = 3;
  int64 dropOff_latitude = 4;
  int64 dropOff_longitude = 5;
  int32 weight = 6;
  int32 height = 7;
  int32 width = 8;
}

message CreateDeliveryResponse {
  string username = 1;
  string delivery = 2;
  string package = 3;
  string status = 4;
  google.protobuf.Timestamp created = 5;
  int64 drone = 6;
  string transportation_request = 7;
}

message PickupPackageRequest {
  string drone = 1;
}

message PickupPackageResponse {
  string delivery = 1;
  int64 pickup_latitude = 2;
  int64 pickup_longitude = 3;
  int64 dropOff_latitude = 4;
  int64 dropOff_longitude = 5;
  string status = 6;
}
(…)

service SchedulerService {
  rpc scheduleDelivery(CreateDeliveryRequest) returns
(CreateDeliveryResponse);
  rpc pickupPackage(PickupPackageRequest) returns (PickupPackageResponse);
  (…)
}
```

Código 9 – Trecho retirado de *scheduler_service.proto*

Código 9 apresenta parte do ficheiro *.proto* que contém a definição do serviço gRPC presente em *Scheduler Service*. Comparando com o design REST, é observável o uso de "verbos" em alternativa aos "nomes" usados no primeiro. Este ficheiro contém as mensagens, o serviço e os métodos RPC suportados por *Scheduler Service*, seguindo as sugestões estabelecidas em 2.3.2.

De notar (e como já tinha sido determinado em 2.3.2) que o ficheiro *.proto* apresentado (Código 9) é válido para geração de código em qualquer uma das linguagens apresentadas anteriormente como suportadas oficialmente por gRPC. Contudo, verifica-se também a existência de algumas opções (retratadas em Código 9 através da palavra *option*) que, apesar de não alterarem o significado geral do conteúdo do ficheiro, afetam a maneira como este é tratado em determinados contextos – no caso apresentado temos (*Language Guide (proto3) | Protocol Buffers | Google Developers*, 2022):

- **java_multiple_files:** Usado para gerar ficheiros *.java* separados para cada uma das classes/enums/etc geradas para cada uma das mensagens, serviços e enumerações
- **java_package:** Usado na geração de código *Java* ou *Kotlin* de modo a especificar o nome do pacote que irá conter os ficheiros gerados
- Existem também outras opções (optimize_for, cc_enable_arenas, entre outras) que afetam a geração de código noutras linguagens ou até a possibilidade de definir opções personalizadas

```
@Data
@NoArgsConstructor
public class ScheduleDeliveryResponseDTO {

    private String username;
    private String delivery;
    private String package;
    private String status;
    private Date created;
    private TransportationDTO transportation;

}
```

Código 10 – Representação da resposta à requisição de um drone para recolha/entrega de mercadoria no *Scheduler Service* que adota REST

Comparando o formato da resposta que *Scheduler Service* que adota REST devolve à requisição de um drone (Código 10) com a resposta que o mesmo serviço, mas que adota gRPC, devolve (Código 9), é possível concluir que estes apresentam maioritariamente um formato muito próximo um do outro, exceto:

- No primeiro existe o uso do objeto *TransportationDTO* (embutido em *ScheduleDeliveryResponseDTO*) enquanto na mensagem *CreateDeliveryResponse* o uso de um objeto embutido não acontece. Este facto reflete uma escolha do desenvolvedor, pois também na definição de um ficheiro *.proto* existe a possibilidade de usar mensagens dentro de mensagens
- Na mensagem *CreateDeliveryResponse* é usado o tipo *Timestamp* de modo a especificar o momento em que a requisição de drone foi criada. O tipo *Date* (ou outros semelhantes) não existe na definição de um ficheiro *.proto*, sendo recomendada a utilização de *Timestamp* para comunicar informação relacionada com pontos temporais. *Timestamp* representa um ponto no tempo independente de qualquer fuso horário ou calendário, e é codificado como uma contagem de segundos e frações de

segundo numa resolução de nanossegundos. Tudo isto resulta, na prática, num formato de compreensão humana mais difícil e na possível necessidade de realizar conversões entre este tipo e outros mais comuns e facilmente interpretáveis do lado da aplicação cliente

Após definir um ficheiro *.proto* é necessário implementar a lógica do serviço gRPC que contém o conjunto de métodos RPC especificados. Para isso é necessário compilar a definição do serviço gRPC e gerar o código fonte na linguagem escolhida. A geração do código pode ser realizada através da compilação manual do ficheiro *.proto* (usando o compilador indicado), ou através de ferramentas de automação, como *Maven*, *Gradle* ou outras, que contém um conjunto de regras estabelecidas de geração de código aquando a construção do projeto (Indrasiri and Kuruppu, 2020).

Nos serviços *Node.js* (*Delivery Service*, *Package Service* e *3rd Party Transportation Service*) o código é gerado dinamicamente em tempo de execução (através de funcionalidade fornecida por *@grpc/proto-loader*). Após carregar a definição do serviço, tudo o que resta fazer é definir o comportamento que cada método gRPC deve ter - Código 11, onde existe um "mapeamento" entre os métodos RPC e métodos oferecidos por *PackageController*, responsável pelo tratamento dos mesmos.

```
import protoLoader from "@grpc/proto-loader";
import { PackageController } from "../api/controller/controller.js";

const PACKAGE_SERVICE_PROTO_PATH = "./proto/package_service.proto";

export default ({ server, grpc }) => {
  const controller = new PackageController();

  var packageDefinition = protoLoader.loadSync(PACKAGE_SERVICE_PROTO_PATH, {
    keepCase: true,
    longs: String,
    enums: String,
    defaults: true,
    oneofs: true,
  });
  const packageService = grpc.loadPackageDefinition(packageDefinition);

  server.addService(packageService.packagemanagement.PackageService.service,
{
    createPackage: controller.createPackage,
    createTagForPackage: controller.createTagForPackage,
  });
};
```

Código 11 – Carregamento da definição do serviço *Package Service*

Visto que os serviços *Spring Boot* desenvolvidos utilizam *Maven* como ferramenta de gestão e automação do processo de construção, decidiu-se recorrer ao *protobuf-maven-plugin* – um *plugin* que integra o compilador de buffers de protocolo no ciclo *Maven*. A configuração do mesmo pode ser observada em Código 12.

```
<plugin>
	<groupId>org.xolstice.maven.plugins</groupId>
	<artifactId>protobuf-maven-plugin</artifactId>
	<version>0.6.1</version>
	<configuration>
		<protocArtifact>com.google.protobuf:protoc:3.19.2:exe:${os.det
	ected.classifier}</protocArtifact>
		<pluginId>grpc-java</pluginId>
		<pluginArtifact>io.grpc:protoc-gen-grpc-
	java:1.45.1:exe:${os.detected.classifier}</pluginArtifact>
	</configuration>
	<executions>
		<execution>
			<goals>
				<goal>compile</goal>
				<goal>compile-custom</goal>
			</goals>
		</execution>
	</executions>
</plugin>
```

Código 12 – Configuração do *protobuf-maven-plugin*

Com esta configuração, a compilação do projeto gera os serviços de acordo com os ficheiros *.proto* existentes – a Figura 26 apresenta parte do resultado da geração de código em *User Management Service* (são também geradas classes para as mensagens). Como é possível observar foi gerado uma classe *UserServiceGrpc* como resposta ao serviço, de nome *UserService*, declarado no ficheiro *.proto* deste projeto (semelhante ao observado em Código 9).

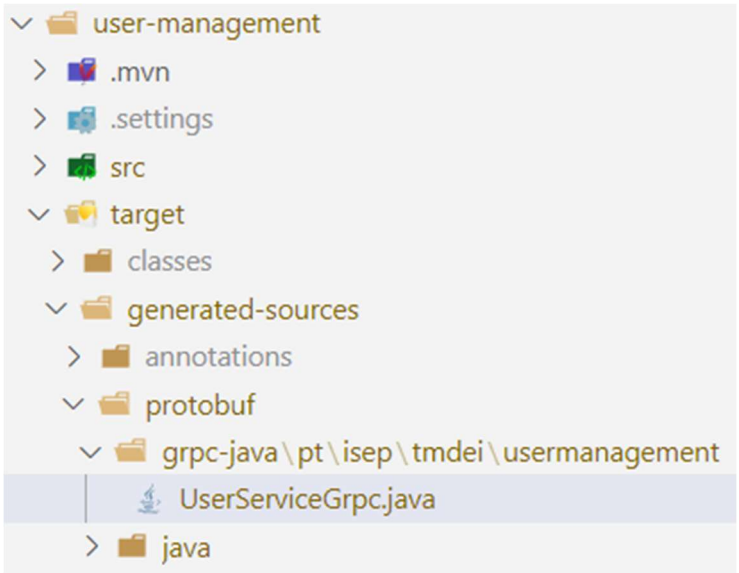

Figura 26 – Resultado da geração de código em *User Management Service*

Após a geração destes ficheiros, que contêm o código mais baixo nível necessário para estabelecer a comunicação gRPC (e outros que contêm classes que representam as mensagens), é necessário adicionar a lógica de negócio através da extensão das classes geradas (Código 13).

```
@GrpcService
@RequiredArgsConstructor
public class UserManagementGrpcService extends
UserServiceGrpc.UserServiceImplBase {

    private final UserAccountService service;

    public void getUserAccount(GetAccountRequest request,
            io.grpc.stub.StreamObserver<GetAccountResponse>
responseObserver) {
        responseObserver.onNext(service.getAccount(request.getUsername()));
        responseObserver.onCompleted();
    }

}
```

Código 13 – *UserManagementService* que estende uma das classes geradas

5.2.3 Tratamento de erros

De modo a comunicar falhas no processamento de um pedido, permitindo assim que aplicações cliente prossigam para a chamada seguinte ou que realizem uma outra operação, é necessário tratar e comunicar corretamente os erros que ocorrem.

Dentro dos protótipos desenvolvidos, e em ambas as tecnologias, a abordagem aos erros foi a mesma:

- Lançamento de exceções específicas
- Tratamento dessas exceções, traduzindo-as em erros específicos dependendo do protocolo de comunicação usado

Em *Node.js*, foram construídas exceções personalizadas que contêm a mensagem que descreve o erro e um código que identifica o tipo de erro que esta representa. Código 14 e Código 15 mostram a exceção lançada case se tente atualizar o drone responsável (ou identificador do pedido de transporte, observável em Figura 24) por uma *Delivery* que não exista na base de dados de *Delivery Service*.

```
export function DeliveryNotFoundException(delivery) {
  const error = new Error("There is no delivery with id = " + delivery);
  error.status = 404;
  return error;
}
```

Código 14 – *DeliveryNotFoundException* presente em *Delivery Service* (REST)

```
import grpc from "@grpc/grpc-js";

export function DeliveryNotFoundException(delivery) {
  const error = new Error("There is no delivery with id = " + delivery);
  error.status = grpc.status.NOT_FOUND;
  return error;
}
```

Código 15 – *DeliveryNotFoundException* presente em *Delivery Service* (gRPC)

As exceções são lançadas e tratadas na função *handleError* (Código 16 e Código 17) – aqui a resposta é construída de acordo com o conteúdo do erro captado.

```
export const handleError = (res, err) => {
  res.status(err.status || 500).json({
    message: err.message || "Internal Server Error",
  });
};
```

Código 16 – *handleError* (REST)

```
import grpc from "@grpc/grpc-js";

export const handleError = (callback, err) => {
  return callback({
    code: err.status || grpc.status.INTERNAL,
    message: err.message,
  });
};
```

Código 17 – *handleError* (gRPC)

Nos serviços *Spring Boot* é possível observar o mesmo raciocínio - Código 18 apresenta a exceção lançada caso um utilizador não seja encontrado no *User Service* (Figura 24).

```
public class AccountNotFoundException extends RuntimeException {

    public AccountNotFoundException(final String message) {
        super(message);
    }

}
```

Código 18 – *AccountNotFoundException* presente em *User Service* (igual no serviço REST e no serviço gRPC)

Estas exceções são posteriormente apanhadas pela classe marcada como *@ControllerAdvice* em REST ou pela classe marcada como *@GrpcAdvice* em gRPC (Código 19 e Código 20).

@ControllerAdvice e *@GrpcAdvice* servem o mesmo propósito: permitir o tratamento de exceções lançadas pela aplicação num componente global e central.

```
@ControllerAdvice
public class ControllerExceptionHandler {

    @ExceptionHandler(value = { AccountNotFoundException.class })
    public ResponseEntity<ErrorMessage> accountNotFoundException
    (RuntimeException ex, WebRequest request) {
        ErrorMessage message = new ErrorMessage(
                HttpStatus.NOT_FOUND.value(), new Date(), ex.getMessage()
               );
        return new ResponseEntity<ErrorMessage>(
                message,
                HttpStatus.NOT_FOUND
               );
    }
}
```

Código 19 – *@ControllerAdvice* observável em *User Service* (REST)

```
@GrpcAdvice
public class ServiceExceptionHandler {

    @GrpcExceptionHandler(AccountNotFoundException.class)
    public Status accountNotFoundException(RuntimeException ex) {
        return Status.NOT_FOUND.withDescription(ex.getMessage());
    }

}
```

Código 20 – *@ServiceAdvice* observável em *User Service* (gRPC)

Através dos exemplos apresentados acima (Código 14 a Código 20) é possível chegar à conclusão que, dentro da mesma tecnologia, a implementação do tratamento e comunicação de erros nas diferentes abordagens (REST vs. gRPC) acaba por ser muito semelhante.

A principal diferença prende-se com os códigos de erro devolvidos:

- Em REST, em caso de erro, deve ser devolvido um erro da família 400 (erro do lado do cliente) ou da família 500 (erro do lado do servidor)
- gRPC, é mais limitado, devendo ser devolvido um dos códigos presentes na Tabela 16

Tabela 16 – Erros gRPC

Código	Número	Descrição
OK	0	Sucesso
CANCELED	1	Operação cancelada (pelo cliente)
UNKNOWN	2	Erro desconhecido
INVALID_ARGUMENT	3	Cliente especificou um argumento inválido
DEADLINE_EXCEEDED	4	Limite expirado antes da conclusão da operação
NOT_FOUND	5	Entidade requisitada não foi encontrada
ALREADY_EXISTS	6	Entidade já existe
PERMISSION_DENIED	7	Cliente não tem permissões para executar a operação
RESOURCE_EXHAUSTED	8	Algum recurso foi esgotado
FAILED_PRECONDITION	9	Sistema não se encontra no estado necessário para a execução da operação
ABORTED	10	Operação abortada
OUT_OF_RANGE	11	Operação tentada para além do intervalo válido
UNIMPLEMENTED	12	Operação não implementada/disponível
INTERNAL	13	Erro interno
UNAVAILABLE	14	O serviço encontra-se indisponível
DATA_LOSS	15	Perda ou corrupção irrecuperável de dados
UNAUTHENTICATED	16	Pedido não possui credenciais de autenticação válidas para a operação

5.2.4 Comunicação com outros serviços

Como estabelecido anteriormente, a requisição de um drone para recolha/entrega de mercadoria (Figura 24), envolve uma série de chamadas tipicamente realizadas entre o *Scheduler Service* e todos os outros serviços. Nesta secção será explorada a forma como essa comunicação é realizada nos dois protótipos.

REST

Apesar de não existir nenhuma restrição que implique que o desenvolvimento de uma API REST tenha de obrigatoriamente usar HTTP, a verdade é que praticamente todas as implementações

REST utilizam este protocolo, dado o bom mapeamento que existe entre este e o estilo REST – este é também o caso da aplicação protótipo desenvolvida.

Assim, no que toca a realizar pedidos HTTP entre serviços, foi apenas necessário configurar clientes HTTP - Código 21 apresenta o cliente usado em *Scheduler Service* para comunicar com *Delivery Service*.

```
public class DeliveryServiceClientImpl extends ServiceClient implements
DeliveryServiceClient {

    private RestTemplate restTemplate;

    public DeliveryServiceClientImpl(String url, String prefix) {
        super(url, prefix);
        this.restTemplate = new RestTemplate(new
HttpComponentsClientHttpRequestFactory());
    }

    @Override
    public ResponseEntity<CreateDeliveryResponseDTO> createDelivery(String
username, CoordinatesDTO pickupCoordinates,
            CoordinatesDTO dropOffCoordinates) {
        var request = new CreateDeliveryRequestDTO();
        request.setUsername(username);
        request.setPickupCoordinates(pickupCoordinates);
        request.setDropOffCoordinates(dropOffCoordinates);
        return restTemplate.postForEntity(this.basePath() + "/deliveries",
request, CreateDeliveryResponseDTO.class);
    }

    @Override
    public ResponseEntity<GetDeliveryResponseDTO> getDelivery(String
delivery) {
        return restTemplate.getForEntity(this.basePath() + "/deliveries/" +
delivery, GetDeliveryResponseDTO.class);
    }
    (…)
}
```

Código 21 – Implementação do cliente para comunicação entre o *Scheduler Service* e *Delivery Service*

Dada a abordagem seguida no desenvolvimento deste protótipo (*Code First*) existiu a necessidade de "duplicar" objetos entre aplicações: por exemplo, o objeto para qual a entidade *Delivery* é serializada durante um pedido para obter uma entrega pelo seu identificador, deve ser "manualmente" criado no serviço que origina a resposta (neste caso *Delivery Service*) e no serviço que recebe a resposta (neste caso *Scheduler Service*), tornando, potencialmente, a manutenção da solução mais complexa. Este problema é ainda mais agravado quando existe a necessidade de lidar com diferentes linguagens (como é o caso). O impacto deste problema poderia ter sido reduzido caso a abordagem *Design First* tivesse sido também aqui adotada. Dito isto, a abordagem *Design First* apresenta também o seu conjunto de desafios que serão explorados mais à frente.

gRPC

De forma a realizar chamadas a outros serviços existem dois passos necessários:

- Geração do código do lado do cliente
- Configuração do cliente

Para gerar o código do lado do cliente (neste caso o objetivo é gerar as classes *Stub*, que representam implementações completas de clientes), em *Scheduler Service* (*Spring Boot*), é fundamental configurar o *plugin* responsável pela geração de código e incluir no projeto os ficheiros *.proto* que definem os serviços com os quais se pretende comunicar (Figura 27). Como a linguagem é praticamente irrelevante para a geração de código a partir de um ficheiro *.proto*, estes podem ser diretamente transferidos de um serviço para o outro sem necessidade de realizar qualquer alteração. No que toca à configuração do *plugin* de geração de código do lado do cliente, esta (com o *plugin* usado) não sofre qualquer alteração quando comparada com o apresentado em Código 12.

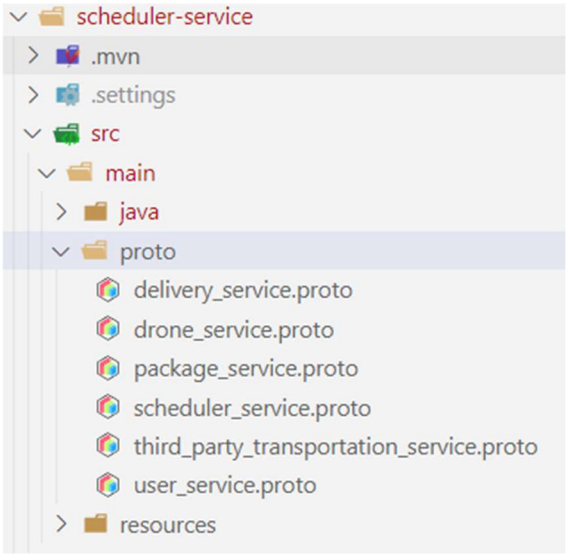

Figura 27 – Inclusão de ficheiros *.proto*

Como é possível observar pela Figura 27, a estratégia seguida neste protótipo para a inclusão de ficheiros *.proto* de outros serviços foi a cópia do ficheiro de um serviço para o outro. Esta abordagem permite que as mensagens, serviços e métodos RPC sejam facilmente replicados ao longo de vários serviços, mas apresenta (pelo menos) um problema óbvio: caso o ficheiro que define um serviço sofra alguma alteração, essa mesma alteração deve ser replicada em outros serviços consumidores do serviço inicial. De modo a combater este problema (algo que não foi realizado no desenvolvimento deste protótipo) existe a necessidade de implementar uma estratégia de partilha destes ficheiros a partir de uma única fonte (por exemplo download dos ficheiros a partir de um único repositório dedicado ao versionamento dos mesmos).

```java
@Component
public class DroneServiceClientImpl implements DroneServiceClient {

    @GrpcClient("drone-management")
    private DroneServiceBlockingStub droneManagementStub;

    @Override
    public BookDroneResponse bookDrone(BookDroneRequest request) {
        return droneManagementStub.bookDrone(request);
    }

    @Override
    public IdleDroneResponse idleDrone(IdleDroneRequest request) {
        return droneManagementStub.idleDrone(request);
    }

}
```

Código 22 – Cliente presente em *Scheduler Service* para comunicação com *Drone Service*

Após geração do código, tudo o que resta é usar a classe *Stub* gerada (neste caso *DroneServiceBlockingStub*, injetada e configurada através da anotação *@GrpcClient*) para realizar as chamadas a outro serviço.

5.2.5 Comunicação entre *frontend* e *backend*

Apesar de não ter sido desenvolvido nenhum componente *frontend*, nem nenhum componente deste género ser apresentado na arquitetura da solução (Figura 23), decidiu-se na mesma explorar brevemente (visto ser uma das propostas deste trabalho) a comunicação *frontend-backend*.

No que toca a comunicar com serviços que disponibilizem uma API REST, tudo o que é necessário da parte da aplicação *frontend* é o uso de um cliente HTTP de modo a realizar os vários pedidos. Esta simplicidade não existe, atualmente, na comunicação com serviços gRPC, encontrando-se nesta área, uma das principais fraquezas associadas ao uso desta tecnologia: dado o ainda relativamente pequeno ecossistema (em comparação com o protocolo convencional REST/HTTP), o apoio ao gRPC em aplicações *browser* e móveis encontra-se ainda nas fases primitivas (Indrasiri and Kuruppu, 2020), não sendo possível chamar um serviço gRPC HTTP/2 a partir de uma aplicação deste género (Newton-King, 2022b).

De modo a combater este problema foi construída a especificação gRPC-web – a ideia base é que o *browser* envie pedidos HTTP normais e que exista um *proxy* à frente do servidor que seja capaz de traduzir os pedidos e respostas do servidor em algo utilizável pelo *browser* (Figura 28).

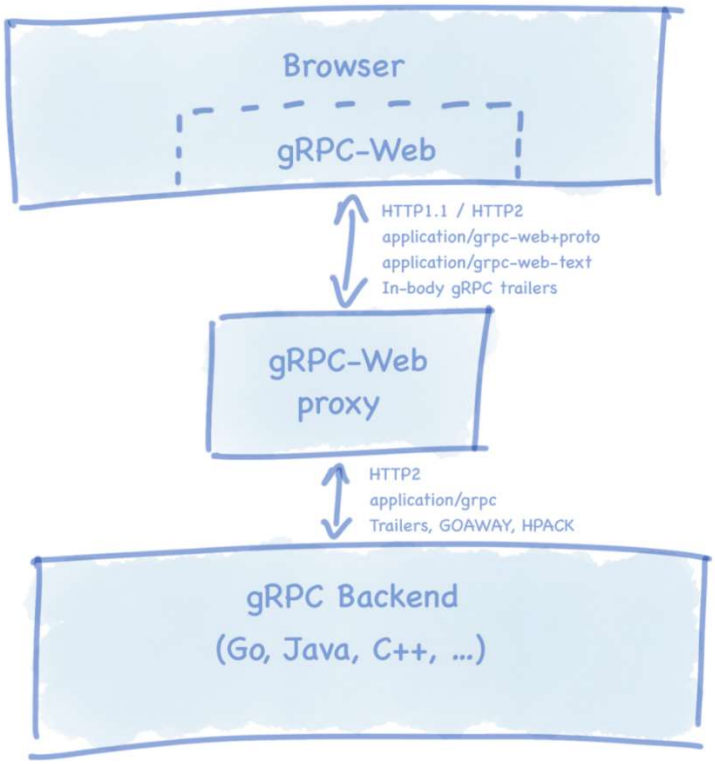

Figura 28 – Uso de gRPC em aplicações *frontend* (Brandhorst, 2019)

Existem atualmente duas implementações gRPC-web (Google e Improbable), que permitem a comunicação entre o *browser* e serviços gRPC (Brandhorst, 2019). O uso de gRPC-web envolve:

- Definição do ficheiro *.proto*
- Configuração de um *proxy* que encaminhe pedidos entre o *browser* e o(s) serviço(s) gRPC
- Geração de código cliente através do compilador de ficheiros *.protoc*
- Escrita do código cliente que realiza pedidos ao *proxy*

Outro ponto negativo do uso de gRPC-web, é que, mesmo após toda a configuração necessária, nem todas os padrões de comunicação associados a gRPC são suportados: apenas RPC simples e *streaming* do lado do servidor são atualmente suportados (Brandhorst, 2019).

6 Avaliação da solução

Este capítulo, tal como o próprio nome indica, tem como função expor o processo a seguir para analisar e avaliar a solução final, bem como apresentar e refletir sobre os resultados obtidos. Para isso será apresentada a hipótese que se procura estudar, seguida da descrição das grandezas que servirão como base à sua avaliação e apresentação da metodologia de avaliação seguida, terminando com uma análise aos resultados obtidos.

6.1 Hipótese

Uma hipótese de investigação é uma "proposição específica, clara e testada ou uma declaração preditiva sobre o possível resultado de um estudo de investigação científica baseado numa propriedade particular de uma população, tais como diferenças presumidas entre grupos sobre uma variável particular ou relações entre variáveis". A especificação da(s) hipótese(s) de investigação é um dos mais importantes passos no planeamento de um estudo de investigação científica (Lavrakas, 2008).

Uma boa hipótese deve (Mourougan and Sethuraman, 2017):

- Ser simples e específica
- Apresentar poder explicativo
- Indicar a relação prevista entre as variáveis consideradas
- Ser verificável
- Consistente com o conjunto de conhecimentos existentes
- Ser declarada da forma mais simples e concisa possível

No contexto deste trabalho, e considerando os cenários de comunicação envolvidos no desenvolvimento de cada protótipo, a seguinte hipótese foi definida:

- **Hipótese 1 (H1):** A aplicação que recorre a gRPC para comunicação entre os serviços que a compõe apresenta desempenho superior à aplicação que adota o estilo REST

6.2 Indicadores de Avaliação

Esta secção tem como principal objetivo esclarecer quais os indicadores que serviram como base à avaliação das hipóteses apresentadas anteriormente. Este passo é importante dado que serve de indicação sobre qual a informação recolhida com o objetivo de realizar a avaliação desejada.

Como indicado pela hipótese apresentada anteriormente, pretende-se focar a avaliação num conjunto de indicadores de desempenho:

- De maneira geral, desempenho pode ser definido como a eficácia que um sistema de software apresenta no que diz respeito a restrições de tempo e a atribuição de recursos. Tempo de resposta, rendimento e utilização são alguns dos índices de desempenho mais tradicionais (Cortellessa, di Marco and Inverardi, 2011). Neste projeto será dada maior ênfase aos seguintes índices de desempenho:
    - Tempo de resposta – tempo decorrido desde o envio de um pedido até imediatamente após a resposta ser recebida
    - Número de transações por segundo – número de pedidos por segundo
    - Tamanho das transações – tamanho, em bytes, das transações realizadas com a aplicação

6.3 Metodologia de Avaliação

Esta secção tem como finalidade descrever a metodologia de avaliação usada neste trabalho. A metodologia de avaliação depende dos indicadores de avaliação definidos anteriormente e representa o conjunto de atividades que devem ser realizadas de forma a avaliar as hipóteses propostas na secção 6.1.

6.3.1 Plano para avaliação do desempenho

A avaliação do desempenho das diferentes aplicações protótipo foi realizado recorrendo ao *software* JMeter. JMeter é uma aplicação concebida para testar e examinar o desempenho e comportamento funcional de aplicações cliente/servidor. JMeter atua como cliente e mede o tempo de resposta, assim como cargas de CPU, utilização de memória, entre outras métricas (Halili, 2008).

Para este trabalho JMeter será foi usado para recolher (*Apache JMeter - User's Manual: Glossary*, 2019):

- ***Elapsed Time***: tempo decorrido desde imediatamente antes do pedido ser enviado até imediatamente após a última resposta ser recebida
- ***Latency*****:** tempo decorrido desde imediatamente antes do pedido ser enviado até imediatamente após a primeira resposta ser recebida. O tempo inclui todo o processamento necessário para a construção do pedido, bem como a construção da primeira parte da resposta
- ***Average*****:** média de tempo decorrido para a conclusão de cada amostra
- ***Median*****:** valor que divide as amostras em duas metades iguais
- ***90% Line (90th Percentile)*****:** representa o valor abaixo do qual se encontram 90% das amostras (JMeter permite também obter *95% Line* e *99% Line*)
- ***Standard Deviation*****:** medida estatística padrão que indica o quão longe a população avaliada se encontra da mediana
- ***Throughput*****:** medida calculada através da divisão do número de pedidos pelo tempo total. O tempo considerado vai desde o início da primeira amostra até ao fim da última amostra
- ***Received/Sent KB/Sec*****:** taxa de transferência em Kilobytes/segundo (recebido e enviado)

É importante mencionar que as aplicações avaliadas foram executadas em ambientes idênticos e submetidas a cargas semelhantes para a recolha das métricas descritas acima. Para garantir que o ambiente de execução era semelhante em ambos os protótipos, foram usados containers Docker, onde foram limitados os recursos a que as instâncias tinham acesso – cada aplicação (6 serviços) foi limitada a 4 *CPU cores* e 3GB de memória.

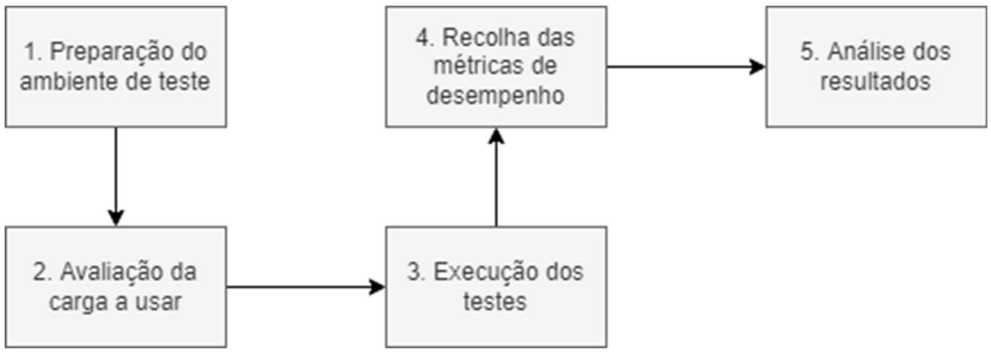

Figura 29 – Fases da avaliação

A Figura 29 apresenta as várias fases propostas inicialmente para o processo de avaliação:

- **Preparação do ambiente de teste:** Consiste do *deploy* das aplicações a serem testadas para containers idênticos
- **Avaliação da carga a usar:** Através do teste de diferentes cargas, deve-se chegar a uma carga a que todas as aplicações consigam responder dentro de 600ms

- **Execução dos testes:** Aqui deve ser delineado e executado o plano de testes (sequência de pedidos a realizar)
- **Recolha das métricas de desempenho:** As métricas de desempenho devem ser recolhidas e arquivadas para posterior análise
- **Análise dos resultados:** Análise e apresentação das métricas recolhidas

6.3.2 Preparação do ambiente de teste

A preparação do ambiente de teste foi realizada recorrendo à tecnologia de containers, nomeadamente através das ferramentas *docker* e *docker-compose*, permitindo a rápida construção e implantação dos vários serviços. Para cada projeto foi definido um *Dockerfile* (Código 23 e Código 25Código 24) e para cada aplicação um *docker-compose.yml* (Código 24) que permite instanciar as aplicações a partir de um único comando.

```
FROM node:14

WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .

CMD [ "node", "./app.js" ]
```

Código 23 – *Dockerfile* típico dos serviços *Node.js*

```
FROM adoptopenjdk/openjdk11:alpine as build

WORKDIR /app

COPY mvnw .
COPY .mvn .mvn

COPY pom.xml .

RUN ./mvnw dependency:go-offline -B

COPY src src

RUN ./mvnw package -DskipTests
RUN mkdir -p target/dependency && (cd target/dependency; jar -xf ../*.jar)

FROM adoptopenjdk/openjdk11:alpine

ARG DEPENDENCY=/app/target/dependency

COPY --from=build ${DEPENDENCY}/BOOT-INF/lib /app/lib
COPY --from=build ${DEPENDENCY}/META-INF /app/META-INF
COPY --from=build ${DEPENDENCY}/BOOT-INF/classes /app

ENTRYPOINT ["java","-
cp","app:app/lib/*","pt.isep.tmdei.dronemanagement.DroneManagementApplicatio
n"]
```

Código 25 – *Dockerfile* típico dos serviços *Spring Boot*

```
version: "3.8"

services:
  scheduler:
    container_name: scheduler
    build:
      context: scheduler-service
      dockerfile: Dockerfile
    ports:
      - "${SCHEDULER_SERVICE_PORT}:${INTERNAL_PORT}"
    restart: always
    environment:
      SERVER_SERVLET_CONTEXT-PATH: $SCHEDULER_SERVICE_PREFIX
      USER-SERVICE_URL: $USER_SERVICE_URL
      ...
    networks:
      - backend

  user-management: ...

  delivery-management: ...

  ...

networks:
  backend:
```

Código 24 – Parte de o ficheiro *docker-compose.yml* de um dos protótipos

6.3.3 Avaliação da carga a usar

De modo a estabelecer a carga a usar durante os testes de desempenho, foram efetuados vários testes sucessivos (após definir um caso de teste, apresentado em 6.3.4), com diferentes configurações, de modo a chegar à configuração que permitisse obter um tempo de resposta inferior ou igual a 600 milissegundos.

Em *JMeter*, a configuração da carga de testes envolveu:

- **Número de *threads* (utilizadores):** Este número representa a quantidade de utilizadores virtuais que se espera que se conectem ao servidor em simultâneo - o número de *threads* que realiza pedidos ao servidor em paralelo
- **Período de *ramp-up*:** Tempo (em segundos) que indica o período que o *JMeter* deve demorar a inicializar o número de *threads* definido. Com um número de *threads* igual a 10, e um período de *ramp-up* igual a 100, o *JMeter* inicializará uma *thread* a cada 10 segundos
- **Total de repetições:** Número de vezes que o caso de teste é iterado

No fim foi estabelecida a configuração apresentada na Tabela 17, o que resultou em 2000 execuções de cada pedido presente no caso de teste.

Tabela 17 – Configuração *JMeter*

Configuração	Valor
Número de t*hreads*	10
Período de *ramp-up*	10
Total de repetições	200

6.3.4 Execução dos testes

Para testar o desempenho das aplicações protótipo foi definido um caso de teste que envolve uma sequência de pedidos normalmente envolvidos no início e conclusão do processo de recolha/entrega por drone. O caso de teste envolve a seguinte sequência de pedidos (Figura 30):

1. Requisição de drone para recolha/entrega
2. Atualização de um pacote com uma etiqueta
3. Levantamento da encomenda (por parte do drone)
4. Pedido do tempo de espera estimado
5. Conclusão da entrega da encomenda (por parte do drone)
6. Confirmação da entrega

7. Pedido da lista de todas as encomendas realizadas (por utilizador)

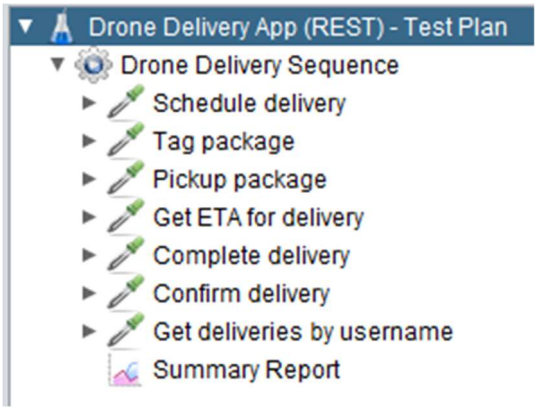

Figura 30 – Caso de teste (REST)

Após estabelecer o caso de teste de cada aplicação protótipo, foram executados os testes e recolhidos os resultados.

6.4 Análise dos resultados

A Tabela 18 e a Tabela 19 apresentam os resultados obtidos.

Tabela 18 – Resultados do protótipo que adota o estilo REST

Pedidos	**Execuções**	**Tempos de resposta (ms)**		***Throughput***	**Rede (KB/seg)**
		Média	**Mediana**		**Recebido**
Total	14000	97.85	78.00	96.73	5116.38
Requisição de drone	2000	168.33	147.00	13.83	5.42
Atribuição de etiqueta a pacote	2000	7.53	6.00	13.91	4.52
Levantamento de encomenda	2000	89.17	74.00	13.91	4.77
Pedido de tempo de espera	2000	52.85	27.00	13.91	2.55
Conclusão de entrega	2000	153.77	131.00	13.91	1.39
Confirmação de entrega	2000	100.29	82.00	13.91	6.13
Pedido de lista de encomendas	2000	112.99	96.00	13.90	5122.96

Tabela 19 – Resultados do protótipo que adota gRPC

Pedidos	Execuções	Tempos de resposta (ms)		*Throughput*	Rede (KB/seg)
		Média	Mediana		Recebido
Total	14000	66.63	25.00	128.33	6175.08
Requisição de drone	2000	88.43	64.00	18.35	3.89
Atribuição de etiqueta a pacote	2000	9.42	9.00	18.54	1.18
Levantamento de encomenda	2000	43.09	15.00	18.60	3.65
Pedido de tempo de espera	2000	26.79	8.00	18.68	0.37
Conclusão de entrega	2000	59.79	31.00	18.74	0.05
Confirmação de entrega	2000	35.67	10.00	18.83	2.59
Pedido de lista de encomendas	2000	203.20	161.50	18.91	6357.68

A Figura 31 e Figura 32 apresentam comparação direta da média de tempos de resposta e *throughput* (ignorando por enquanto a linha "Pedido de lista de encomendas" cujos resultados serão explorados mais à frente) entre o protótipo que adota REST e o protótipo que adota gRPC.

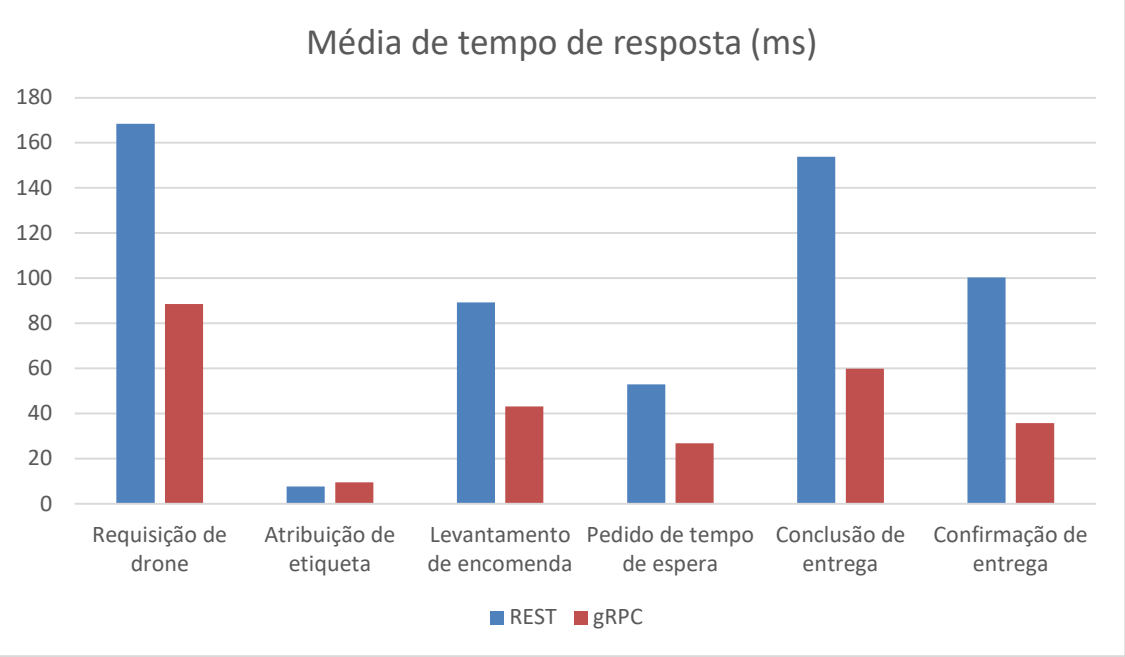

Figura 31 – Comparação da média do tempo de resposta

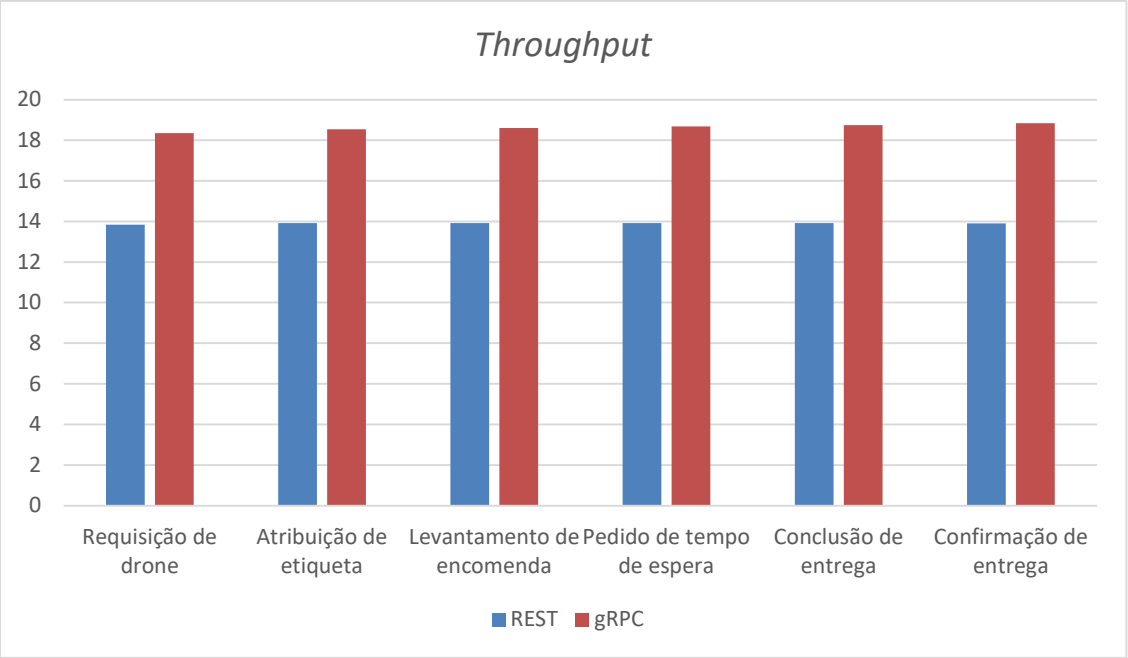

Figura 32 – Comparação do *throughput*

Olhando para a comparação entre a média de tempos de resposta (Figura 31) é possível observar que, no geral, o protótipo que adota gRPC apresenta um tempo de resposta a pedidos inferior ao protótipo que adota REST o que faz com que, naturalmente, o primeiro acabe por apresentar um *throughput* superior (Figura 32).

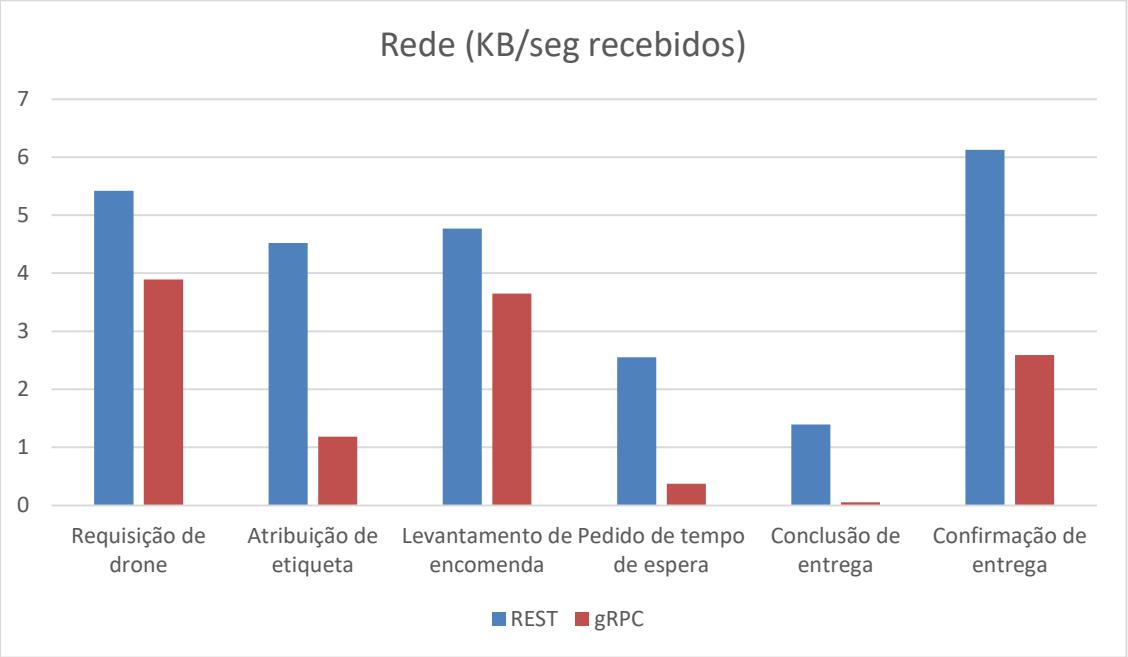

Figura 33 – Comparação de KB/seg recebidos

Outra das medidas de avaliação relevantes para este trabalho está relacionada com o uso da rede – para isso é apresentada a comparação entre os KB/seg recebidos durante a execução do caso de teste em cada um dos protótipos (Figura 33). Aqui, mais uma vez, gRPC é superior – devido ao formato de serialização altamente eficiente oferecido por gRPC, mensagens podem chegar a ser 60% a 80% mais pequenas (quando comparado à serialização JSON) (Vettor and Smith, 2021), o que contribui diretamente para os tempos de resposta apresentados (Figura 31).

Importa mencionar que apesar da ferramenta *JMeter* permitir também capturar os KB/seg enviados, essa medida não é apresentada, pois no caso de teste do protótipo gRPC este valor foi sempre de 0.0KB/seg – este facto representa uma possível limitação do *plugin JMeter* (*jmeter-grpc-request*) necessário à realização de pedidos gRPC.

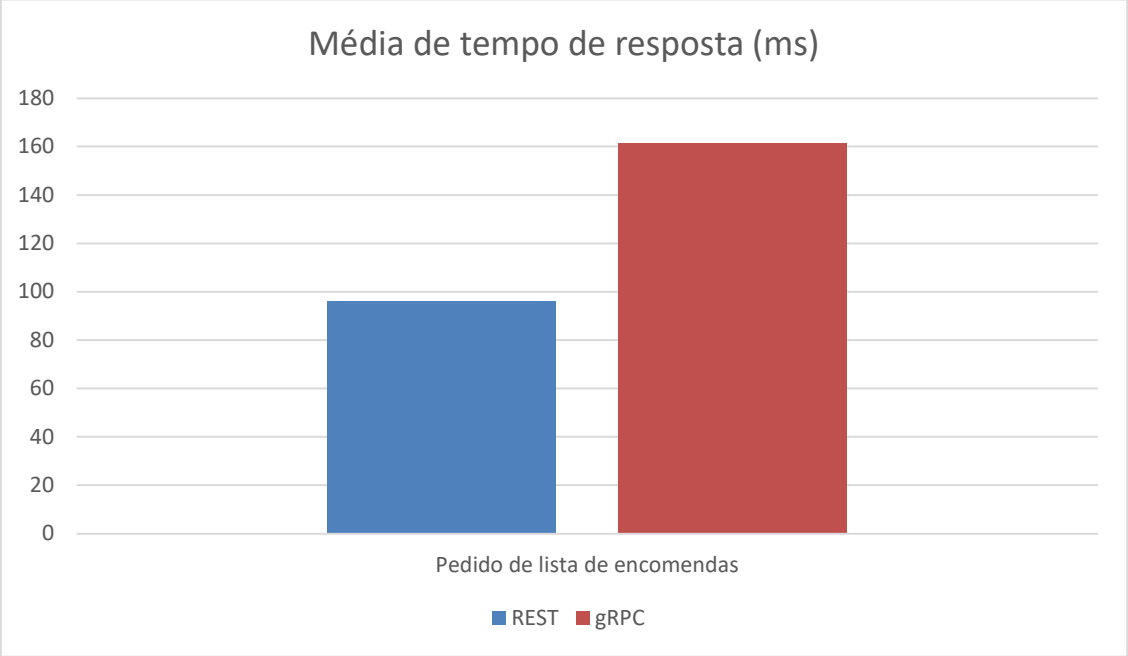

Figura 34 – Comparação entre os tempos de resposta para “Pedido de lista de encomendas”

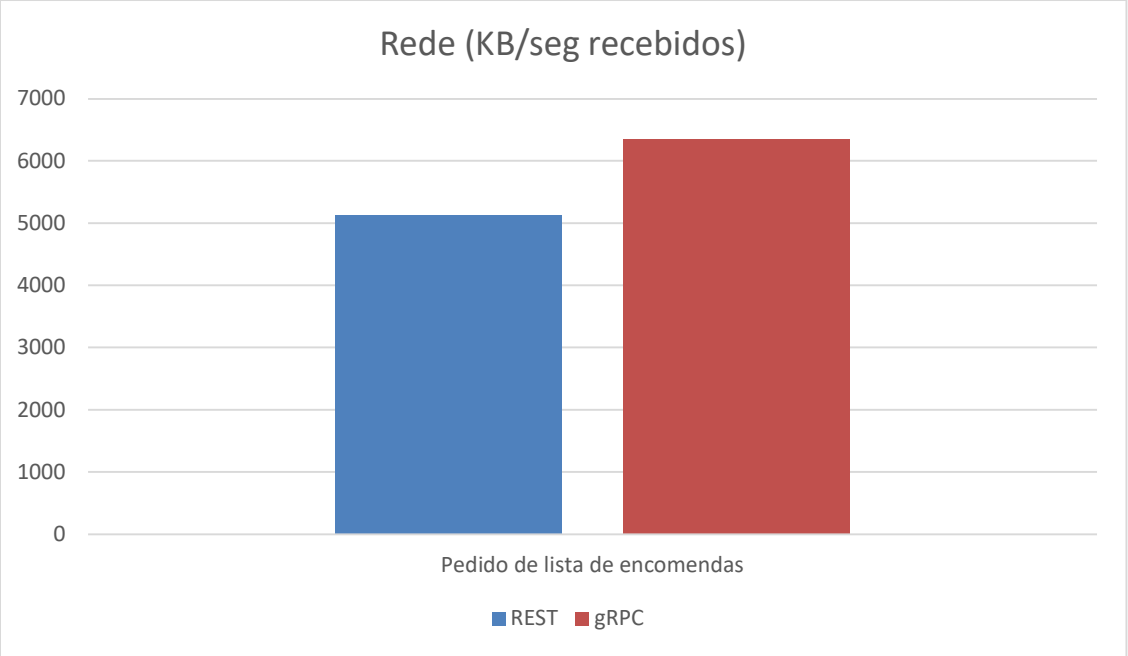

Figura 35 – Comparação de KB/seg recebidos para "Pedido de lista de encomendas"

As figuras Figura 34 e Figura 35 apresentam a comparação entre os resultados obtidos em ambos os protótipos para um dos pedidos não apresentados nos gráficos anteriores – "Pedido de lista de encomendas" que devolve uma coleção de todas as encomendas realizadas pelo utilizador. Este caso foge do observado nos outros pedidos – aqui o protótipo que adota REST não só apresenta consideravelmente menor tempo de resposta como também a leitura de KB/seg recebidos apresenta valores menores.

O principal motivo por trás desta observação está relacionado com o tamanho da mensagem:

- Ao longo do caso de teste são criadas encomendas recorrendo ao mesmo utilizador, ou seja, a coleção de encomendas devolvidas em resposta à requisição da lista de encomendas vai aumentando em tamanho com o decorrer do caso de teste. Não são usadas técnicas que poderiam reduzir o tamanho da resposta de modo a "forçar" o transporte de grandes quantidades de dados e possibilitar a testagem do mesmo
- Um dos pontos desfavoráveis ao uso de gRPC, é que esta tecnologia não foi concebida para transportar grandes mensagens (por definição o limite padrão do tamanho da mensagem é de 4MB; é possível (mas não recomendado) expandir este limite) – durante a serialização e desserialização, a mensagem é carregada em completo para memória, o que afeta diretamente a performance do servidor (Newton-King, 2022a). Em casos onde existe a necessidade de transportar este tipo de mensagens é recomendado (Newton-King, 2022a):
    - Divisão da mensagem em "pedaços" mais pequenos, realizando a sua transmissão através dos métodos de *streaming* gRPC (algo não implementado no protótipo)

- Não usar gRPC, disponibilizando um *endpoints* HTTP para este tipo de transmissões de dados

A análise aos resultados apresentada em ao longo desta secção tem como objetivo permitir chegar a uma conclusão quanto à hipótese apresentada, ou seja, responder se o protótipo que adota gRPC para comunicação entre os serviços que o compõe apresenta desempenho superior à aplicação que adota o estilo REST.

De acordo com os resultados demonstrados, o protótipo que adota gRPC, perante as mesmas condições, apresenta desempenho superior na maioria dos casos. Contudo, em casos onde as mensagens apresentam uma grande quantidade de dados (superando o tamanho máximo definido) e devido à necessidade de carregar a totalidade de uma mensagem em memória durante a serialização/desserialização, o protótipo que adota gRPC apresenta desempenho consideravelmente inferior ao protótipo que adota o estilo REST. Em casos semelhantes ao descrito, é recomendado o uso de outras estratégias como gRPC *streaming* ou a disponibilização de um *endpoint* HTTP.

7 Conclusão

O capítulo atual expõe as conclusões associadas aos diferentes cenários de comunicação entre serviços tendo em consideração a implementação e as experiências documentadas ao longo do documento. Para isso, o capítulo começa por comentar os cenários de comunicação propostos, resume brevemente as conclusões sobre a avaliação executada, apresenta algumas reflexões sobre o desenvolvimento recorrendo a gRPC, seguindo-se as limitações e possíveis pontos de desenvolvimento futuro, e por fim uma apreciação final sobre o trabalho desenvolvido.

7.1 Sobre os cenários de comunicação

Como apresentado na secção 1.3 pretende-se com este trabalho retirar também algumas conclusões relativamente a alguns cenários observados comumente em aplicações compostas por microsserviços, nomeadamente:

- Comunicação síncrona entre serviços
- Comunicação em ambiente poliglota
- Comunicação de baixa latência e alto rendimento
- Comunicação em ambientes com restrições de rede (banda larga disponível)
- Comunicação direta entre *frontend* e *backend*

7.1.1 Comunicação síncrona entre serviços

Em casos onde existe necessidade de comunicação síncrona entre serviços, a maior diferença entre as duas abordagens encontra-se no desempenho que estas apresentam. O tópico do desempenho é explorado a maior detalhe em 7.1.3.

7.1.2 Comunicação em ambiente poliglota

No que diz respeito ao desenvolvimento de uma aplicação onde existe comunicação entre serviços construídos recorrendo a diversas linguagens, qualquer uma das alternativas estudadas apresenta-se como uma opção viável, dado que a linguagem de programação tem pouco ou nenhum impacto na adoção da mesma. No que toca a este tópico foi possível observar que a maior diferença se prende com a abordagem seguida na definição da interface do serviço (*Code First* vs *Design First*) e com a possibilidade de gerar código a partir da mesma, sendo a geração de código a partir de uma única definição algo extremamente útil na construção de uma aplicação deste género.

Apesar da geração de código a partir de uma especificação ser possível também no desenvolvimento de aplicações REST, gRPC apresenta uma ligeira vantagem por dois motivos:

- A obrigatoriedade de definir a interface do serviço (ou serviços) através de ficheiros *.proto*, obrigando a uma abordagem *Design First*, algo que no desenvolvimento REST é opcional (por exemplo através da especificação *OpenAPI*)
- O facto de que a geração de código está automaticamente incorporada em todas as linguagens oficialmente suportadas, algo que no desenvolvimento de uma aplicação REST tende a ser alcançado, por exemplo, através do uso um documento *OpenAPI* em conjunto com uma ou várias ferramentas de geração de código desenvolvidas por terceiros

7.1.3 Comunicação de baixa latência e alto rendimento

Pelos resultados apresentados em 6.4, é possível afirmar que em casos onde é necessária a existência de baixa latência e alto rendimento durante a comunicação entre serviços, o uso de gRPC tende a ser favorável. Contudo, especial atenção deve ser dada em situações onde exista o transporte de grandes quantidades de dados.

7.1.4 Comunicação em ambientes com restrições de rede

A comunicação num ambiente como o implicado neste cenário segue praticamente o mesmo raciocínio que o apresentado em 7.1.3. Segundo as tabelas Tabela 18 e Tabela 19 a quantidade de KB/seg recebidos durante a execução do caso de teste tende a ser menor (na maioria dos casos) no protótipo que adota gRPC, o que coloca esta abordagem na posição de liderança no que diz respeito à comunicação em situações onde existem restrições de rede. Esta liderança atribuída a gRPC é limitada dado o cuidado particular que deve existir caso exista a necessidade de transferência simultânea de mensagens extensas em tamanho.

7.1.5 Comunicação direta entre *frontend* e *backend*

Em relação à comunicação entre componentes *frontend* e *backend* a abordagem REST é, neste momento e como demonstrado em 5.2.5, claramente superior. O suporte para o uso de gRPC em componentes *frontend* encontra-se ainda numa fase embrionária, não existindo ainda suporte para todos os modos gRPC, e dada a necessidade de configurar um *proxy* que serve como ponte entre o *browser* e o(s) serviço(s) gRPC.

7.2 Resultados

Neste trabalho, para além do estudo do desenvolvimento de diferentes protótipos recorrendo a duas diferentes abordagens (estilo REST vs gRPC), foi também realizada uma avaliação com o objetivo de determinar qual das duas aplicações protótipo desenvolvidas apresentaria desempenho superior (capítulo 6). A avaliação de desempenho foi avaliada através da execução de um caso de teste agregador de várias operações envolvidas na principal funcionalidade planeada para cada aplicação (requisição de drone para recolha/entrega de mercadorias).

Como consequência dos resultados obtidos, concluiu-se que o protótipo que adota gRPC aparenta oferecer melhor desempenho na maioria dos casos. Porém, em situações onde são transmitidas mensagens com grandes quantidades de dados (por definição o tamanho máximo de uma mensagem gRPC é 4MB), o protótipo que adota o estilo REST manifestou melhor desempenho.

7.3 Reflexões gerais sobre o desenvolvimento gRPC

Durante o decorrer desta secção serão apresentados alguns tópicos que têm como objetivo oferecer uma perspetiva geral da experiência do estudante durante o desenvolvimento recorrendo a gRPC.

7.3.1 Documentação

A documentação oficial da *framework* expõe conceitos-chave, melhores práticas e recomendações gerais que auxiliam no momento de iniciar o desenvolvimento de serviços gRPC. Contudo, embora comece a existir mais conteúdo sobre a *framework* (a maior parte das fontes sobre gRPC usadas para este trabalho têm menos de 5 anos), a quantidade do mesmo fica ainda muito aquém do número de recursos e trabalhos acadêmicos que abordam o estilo REST. O estilo REST (atualmente) apresenta uma comunidade muito mais rica o que tende a facilitar o desenvolvimento e resolução de problemas relacionados com o mesmo.

7.3.2 Dificuldade

A dificuldade de desenvolver um serviço gRPC prende-se com a necessidade de compreender e utilizar a linguagem de *protocol buffers* de modo a estruturar as mensagens, os serviços e os métodos RPC que se pretende disponibilizar. Felizmente, esta linguagem encontra-se detalhadamente documentada (*Language Guide (proto3) | Protocol Buffers | Google Developers*, 2022).

Após este passo, o principal desafio prende-se com a configuração da geração de código – desenvolvedores já com alguma experiência com a abordagem *Code First* encontrarão aqui um processo familiar e de fácil compreensão.

Relativamente à quantidade de código "manual" necessária ao desenvolvimento de aplicações baseadas em microsserviços, caso seja seguida a mesma abordagem e tendo como referência as aplicações protótipo desenvolvidas, nenhuma das abordagens (REST vs gRPC) tende a apresentar diferenças relevantes.

7.3.3 No geral

Para concluir, a adoção de gRPC para o desenvolvimento de uma aplicação baseada em microsserviços apresenta-se como uma opção bastante adequada. Das principais vantagens observadas, destacam-se:

- Em sistemas baseados em microsserviços é comum observar a existência de serviços desenvolvidos com recurso a diferentes linguagens de programação. gRPC encaixa perfeitamente neste tipo de casos, permitindo a geração de código (do lado do cliente e do lado do servidor) em múltiplas linguagens a partir de uma única definição de serviço
- Na maior parte dos casos, a comunicação entre serviços recorrendo a gRPC apresenta desempenho superior a aplicações que adotam o estilo REST. Num contexto onde simplicidade e escalabilidade são trocadas por desempenho, como é o caso de aplicações compostas por microsserviços, qualquer possível ganho em desempenho ganha maior importância

7.4 Limitações e futuro do trabalho

Na sequência do trabalho desenvolvido, existem algumas limitações que se tornam evidentes e que abrem possíveis pontos de desenvolvimento futuro do mesmo:

- Ao longo deste trabalho, e durante a exposição do desenvolvimento recorrendo a gRPC (secção 5.2), foi dado o principal foco a um dos modos gRPC – RPC simples, onde a aplicação cliente envia um único pedido e obtém uma única resposta. O foco foi mantido neste modo dada a sua semelhança com o funcionamento de um pedido HTTP,

facilitando assim a comparação com a adoção do estilo REST. Assim, o estudo mais detalhado de outros modos gRPC é um ponto de possível desenvolvimento futuro.

- Um dos tópicos abordados durante a avaliação da solução, mais concretamente na secção 6.4, foi o impacto que o tamanho da mensagem apresentou no desempenho do protótipo que usa gRPC. Aqui encontra-se outra das limitações do trabalho, pois, pelos resultados e análise apresentada, é difícil de compreender qual o limite de tamanho exato que provoca a perda de desempenho e se a mesma é resolvível através de outras opções de implementação recorrendo a gPRC (por exemplo aumento do limite máximo da mensagem, uso de outro modo gRPC, entre outras opções). Como desenvolvimento futuro seria interessante estudar com maior detalhe, através de testes de desempenho mais pormenorizados e recorrendo a outros modos gRPC, o real impacto que a dimensão de mensagens tem no desempenho de uma aplicação gRPC.
- Durante a secção 5.2.5 foi abordado brevemente o desenvolvimento de um componente *frontend* que tenha como objetivo a comunicação com um (ou vários) serviços gRPC. Na secção 7.1.5, concluiu-se que o suporte para o desenvolvimento de uma aplicação deste tipo encontra-se ainda numa fase embrionária. O aprofundar deste ponto através do desenvolvimento de uma aplicação *frontend* que comunica com gRPC, expondo com maior detalhe toda a configuração e especificidades que o mesmo envolve, é um possível tema futuro de estudo, principalmente se parte das limitações expostas forem limadas.
- Apesar de não serem necessariamente limitações deste trabalho, visto que não foram propostos para o mesmo, existem um conjunto de temas que seria relevante desenvolver como extensão do trabalho apresentado, e que ajudariam a obter uma maior compreensão sobre o uso de gRPC em ambientes de produção:
    - Comunicação segura
    - *Load Balancing*
    - *Observability* (métricas, *logs*, rastreabilidade, etc.)
    - Outras áreas relevantes para o desenvolvimento de uma aplicação baseada em microsserviços

7.5 Apreciação final

No geral, o trabalho exposto ao longo deste documento permitiu ao estudante adquirir conhecimentos teóricos e práticos sobre uma tecnologia emergente, como é o caso da *framework* gRPC, possibilitando a sua comparação com uma abordagem mais tradicional, neste caso a adoção do estilo REST.

O estudo realizado, numa perspetiva global, cumpriu com os objetivos propostos inicialmente, oferecendo uma introdução e análise à *framework* gRPC. Não obstante, e como discutido na secção 7.4, o trabalho não oferece todas as respostas, deixando ainda vários tópicos de possível investigação futura.

Referências

Apache JMeter - User's Manual: Glossary (2019). Available at: https://jmeter.apache.org/usermanual/glossary.html (Accessed: February 22, 2022).

Aroraa, G., Kale, L. and Manish, K. (2017) *Building Microservices with .NET Core 2.0 : Transitioning monolithic architectures using microservices with .NET Core 2.0 using C# 7.0.* Packt Publishing.

Azure Architecture Center | Microsoft Docs (2022). Available at: https://docs.microsoft.com/en-us/azure/architecture/ (Accessed: April 11, 2022).

Belliveau, P., Griffin, A. and Somermeyer, S. (2004) *The PDMA toolbook 1 for new product development*. John Wiley & Sons.

Beták, M. (2016) "Document Oriented REST," *Diplomová práce, Masarykova univerzita, Fakulta informatiky, Brno* [Preprint].

Brandhorst, J. (2019) *The state of gRPC in the browser*, *gRPC Blog*. Available at: https://grpc.io/blog/state-of-grpc-web/ (Accessed: June 20, 2022).

Brito, G., Mombach, T. and Valente, M.T. (2019) "Migrating to GraphQL: A Practical Assessment," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 140–150. https://doi.org/10.1109/SANER.2019.8667986.

Brito, G. and Valente, M.T. (2020) "REST vs GraphQL: A Controlled Experiment," in *2020 IEEE International Conference on Software Architecture (ICSA)*, pp. 81–91. https://doi.org/10.1109/ICSA47634.2020.00016.

Buck, A. (2021) *Publisher-Subscriber pattern - Azure Architecture Center | Microsoft Docs*. Available at: https://docs.microsoft.com/en-us/azure/architecture/patterns/publisher-subscriber (Accessed: January 6, 2022).

Case studies | Cloud Native Computing Foundation (2022). Available at: https://www.cncf.io/case-studies/?_sft_lf-project=grpc (Accessed: February 5, 2022).

Chalin, P. and Ranney, A. (2021) *Supported languages | gRPC*. Available at: https://grpc.io/docs/languages/ (Accessed: June 10, 2022).

Cortellessa, V., di Marco, A. and Inverardi, P. (2011) *Model-based software performance analysis*. Springer.

De, S. *et al.* (2016) *Evolve the Monolith to Microservices with Java and Node*.

Design a microservices architecture - Azure Architecture Center (2022). Available at: https://docs.microsoft.com/en-us/azure/architecture/microservices/design/ (Accessed: April 11, 2022).

"Developer Survey 2020 Report" (2020).

Dewulf, K. (2013) "Sustainable product innovation: the importance of the front-end stage in the innovation process," *Advances in industrial design engineering*, pp. 139–166.

Doglio, F., Doglio and Corrigan (2018) *REST API Development with Node. js*. Springer.

Domain analysis for microservices - Azure Architecture Center (2022). Available at: https://docs.microsoft.com/en-us/azure/architecture/microservices/model/domain-analysis (Accessed: April 13, 2022).

Doyle, K., Ferguson, K. and McKenzie, C. (2021) *What is REST (REpresentational State Transfer)?*, *TechTarget*. Available at: https://www.techtarget.com/searchapparchitecture/definition/REST-REpresentational-State-Transfer (Accessed: June 10, 2022).

Dragoni Nicola and Giallorenzo, S. and L.A.L. and M.M. and M.F. and M.R. and S.L. (2017) "Microservices: Yesterday, Today, and Tomorrow," in B. Mazzara Manuel and Meyer (ed.) *Present and Ulterior Software Engineering*. Cham: Springer International Publishing, pp. 195–216. https://doi.org/10.1007/978-3-319-67425-4_12.

Einav, Y. (2019) *Amazon Found Every 100ms of Latency Cost them 1% in Sales - GigaSpaces*. Available at: https://www.gigaspaces.com/blog/amazon-found-every-100ms-of-latency-cost-them-1-in-sales/ (Accessed: January 6, 2022).

Feng, X., Shen, J. and Fan, Y. (2009) "REST: An alternative to RPC for Web services architecture," in *2009 First International Conference on Future Information Networks*, pp. 7–10. https://doi.org/10.1109/ICFIN.2009.5339611.

Fielding, R.T. (2000) "REST : Architectural Styles and the Design of Network-based Software Architectures," *Doctoral dissertation, University of California* [Preprint]. Available at: https://ci.nii.ac.jp/naid/20000913235/en/.

Google (2021) *About gRPC | gRPC*. Available at: https://grpc.io/about/ (Accessed: December 17, 2021).

grpc/grpc-swift: The Swift language implementation of gRPC. (2022). Available at: https://github.com/grpc/grpc-swift (Accessed: June 10, 2022).

Halili, E.H. (2008) *Apache JMeter*. Packt Publishing Birmingham.

Himschoot, P. (2021) *Microsoft Blazor: Building Web Applications in .NET*. Third Edition. Edited by J. Gennick, L. Berendson, and J. Balzano. Berkeley, CA: Apress. https://doi.org/10.1007/978-1-4842-7845-1.

Husai, J., Taylor, P. and Paulson, M. (2015) *Netflix Releases Falcor Developer Preview | by Netflix Technology Blog | Netflix TechBlog*. Available at: https://netflixtechblog.com/netflix-releases-falcor-developer-preview-aefc033df7a7 (Accessed: January 30, 2022).

Identify microservice boundaries | Azure Architecture Center (2022). Available at: https://docs.microsoft.com/en-us/azure/architecture/microservices/model/microservice-boundaries (Accessed: April 21, 2022).

Indrasiri, K. and Kuruppu, D. (2020) *GRPC : up and running: building cloud native applications with Go and Java for docker and kubernetes*. First Edition. Edited by R. Shaw et al. O'Reilly Media, Inc.

Indrasiri, K. and Siriwardena, P. (2018) "Microservices for the Enterprise," *Apress, Berkeley* [Preprint].

joyrex2001/grpc-perl: Perl 5 implementation of gRPC using the official gRPC shared library. (2022). Available at: https://github.com/joyrex2001/grpc-perl (Accessed: June 10, 2022).

Koen, P. *et al.* (2001) "Providing Clarity and A Common Language to the 'Fuzzy Front End,'" *Research-Technology Management*, 44(2), pp. 46–55. https://doi.org/10.1080/08956308.2001.11671418.

de la Torre, C., Wagner, B. and Rousos, M. (2021) *.NET Microservices: Architecture for Containerized .NET Applications*. 6.0. Edited by M. Pope and S. Hoag. Microsoft Corporation.

Language Guide (proto3) | Protocol Buffers | Google Developers (2021). Available at: https://developers.google.com/protocol-buffers/docs/proto3 (Accessed: January 13, 2022).

Language Guide (proto3) | Protocol Buffers | Google Developers (2022). Available at: https://developers.google.com/protocol-buffers/docs/proto3 (Accessed: June 16, 2022).

Larrucea, X. *et al.* (2018) "Microservices," *IEEE Software*, 35(3), pp. 96–100. https://doi.org/10.1109/MS.2018.2141030.

Lavrakas, P.J. (2008) *Encyclopedia of Survey Research Methods*. SAGE Publications. Available at: https://books.google.pt/books?id=2sr0CAAAQBAJ.

Loukides, M. and Swoyer, S. (2020) *Microservices Adoption in 2020 – O'Reilly*. Available at: https://www.oreilly.com/radar/microservices-adoption-in-2020/ (Accessed: February 4, 2022).

Magnoni, L. (2015) "Modern messaging for distributed sytems," in *Journal of Physics: Conference Series*, p. 12038.

McLarty, M. (2019) *API interaction types in a microservice architecture: queries, commands, and events | MuleSoft Blog*. Available at: https://blogs.mulesoft.com/dev-

guides/microservices/microservices-api-interaction-queries-commands-events/ (Accessed: January 5, 2022).

Miles, L.D. (2015) *Techniques of value analysis and engineering*. Miles Value Foundation.

Mourougan, S. and Sethuraman, K. (2017) "Hypothesis development and testing," *IOSR Journal of Business and Management (IOSR-JBM)*, 9(5), pp. 34–40.

Neuman, S. (2015) "Building microservices: Designing fine-grained systems," *Oreilly & Associates Inc* [Preprint].

Newton-King, J. (2022a) *Performance best practices with gRPC*, *Microsoft Docs*. Available at: https://docs.microsoft.com/en-us/aspnet/core/grpc/performance?view=aspnetcore-6.0 (Accessed: June 23, 2022).

Newton-King, J. (2022b) *Use gRPC in browser apps*, *Microsoft Docs*. Available at: https://docs.microsoft.com/en-us/aspnet/core/grpc/browser?view=aspnetcore-6.0 (Accessed: June 19, 2022).

Osterwalder, A. *et al.* (2015) *Value proposition design: How to create products and services customers want*. John Wiley & Sons.

Osterwalder, A. and Pigneur, Y. (2003) "Modeling value propositions in e-Business," in *ACM International Conference Proceeding Series*, pp. 429–436. https://doi.org/10.1145/948005.948061.

Palcic, I. and Lalic, B. (2009) "Analytical Hierarchy Process as a tool for selecting and evaluating projects.," *International Journal of Simulation Modelling (IJSIMM)*, 8(1).

Peffers, K. *et al.* (2007) "A design science research methodology for information systems research," *Journal of management information systems*, 24(3), pp. 45–77.

Peon, R. and Ruellan, H. (2015) "HPACK: Header compression for HTTP/2," *Internet Requests for Comments, RFC Editor, RFC*, 7541.

Porcello, E. and Banks, A. (2018) *Learning GraphQL: Declarative Data Fetching for Modern Web Apps*. " O'Reilly Media, Inc."

Rich, N. and Holweg, M. (2000) "Value analysis," *Value engineering* [Preprint].

Richardson, C. (2018) *Microservices patterns: with examples in Java*. Simon and Schuster.

Richardson, C. (2022) *Database per service*. Available at: https://microservices.io/patterns/data/database-per-service.html (Accessed: February 2, 2022).

Rodríguez Carlos and Baez, M. and D.F. and C.F. and T.J.C. and C.L. and P.G. (2016) "REST APIs: A Large-Scale Analysis of Compliance with Principles and Best Practices," in P. and P.C. Bozzon

Alessandro and Cudre-Maroux (ed.) *Web Engineering*. Cham: Springer International Publishing, pp. 21–39.

Saaty, T.L. (1990) “How to make a decision: the analytic hierarchy process,” *European journal of operational research*, 48(1), pp. 9–26.

de Saxcé, H., Oprescu, I. and Chen, Y. (2015) “Is HTTP/2 really faster than HTTP/1.1?,” in *2015 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pp. 293–299. https://doi.org/10.1109/INFCOMW.2015.7179400.

Seabra, M., Nazário, M. and Pinto, G. (2019) “REST or GraphQL?: A Performance Comparative Study,” in *SBCARS ’19: Proceedings of the XIII Brazilian Symposium on Software Components, Architectures, and Reuse*, pp. 123–132. https://doi.org/10.1145/3357141.3357149.

Sharma, S. (2021) *Modern API Development with Spring and Spring Boot: Design highly scalable and maintainable APIs with REST, gRPC, GraphQL, and the reactive paradigm*. Packt Publishing. Available at: https://books.google.pt/books?id=3wYxEAAAQBAJ.

“State of APIs Developer Survey 2021 Report” (2021).

Stefanic, M. (2021) “Developing the guidelines for migration from RESTful microservices to gRPC.”

stepancheg/grpc-rust: Rust implementation of gRPC (2022). Available at: https://github.com/stepancheg/grpc-rust (Accessed: June 10, 2022).

Talwar, V. (2016) *gRPC: a true internet-scale RPC framework is now 1.0 and ready for production deployments | Google Cloud Blog*. Available at: https://cloud.google.com/blog/products/gcp/grpc-a-true-internet-scale-rpc-framework-is-now-1-and-ready-for-production-deployments (Accessed: January 13, 2022).

Tibrewal, Y. *et al.* (2020) *Status codes and their use in gRPC*, *GitHub*. Available at: https://github.com/grpc/grpc/blob/master/doc/statuscodes.md (Accessed: January 20, 2022).

Using tactical DDD to design microservices | Azure Architecture Center (2022). Available at: https://docs.microsoft.com/en-us/azure/architecture/microservices/model/tactical-ddd (Accessed: April 20, 2022).

Varanasi, B. and Belida, S. (2015) *Spring Rest*. Apress.

Vasudevan, K. (2017) *Design First or Code First: What’s the Best Approach to API Development?*, *SWAGGER BLOG*.

Vettor, R. and Smith, S. (2021) *Architecting Cloud Native .NET Applications for Azure*. 1.0.2. Edited by M. Wenzel and D. Pine. Microsoft Corporation. Available at: https://www.microsoft.com.

Wenzel, M., Parente, J. and Anil, N. (2021) *Microservices architecture | Microsoft Docs*. Available at: https://docs.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/microservices-architecture (Accessed: December 17, 2021).

Westeinde, K. (2019) *Deconstructing the Monolith*. Available at: https://shopify.engineering/deconstructing-monolith-designing-software-maximizes-developer-productivity (Accessed: December 30, 2021).

Wijnants, M. *et al.* (2018) "HTTP/2 Prioritization and Its Impact on Web Performance," in *Proceedings of the 2018 World Wide Web Conference*. Republic and Canton of Geneva, CHE: International World Wide Web Conferences Steering Committee (WWW '18), pp. 1755–1764. https://doi.org/10.1145/3178876.3186181.

Zimmermann, T. *et al.* (2017) "How HTTP/2 pushes the web: An empirical study of HTTP/2 server push," in *2017 IFIP Networking Conference (IFIP Networking) and Workshops*, pp. 1–9. https://doi.org/10.23919/IFIPNetworking.2017.8264830.