Universidade de São Paulo
Instituto de Matemática e Estatística
Bacharelado em Ciência da Computação

Análise de *trade-offs* arquiteturais avaliando técnicas de integração entre microsserviços

Lucas Moretto da Silva

Monografia Final

mac 499 — Trabalho de Formatura Supervisionado

Supervisor: Prof. Dr. Alfredo Goldman Vel Lejbman
Cossupervisor: Me. João Francisco Lino Daniel

São Paulo
2022

Agradecimentos

Primeira, gostaria de agradecer a minha família por todo apoio que me dão. As lições e aprendizados adquiridos durante meus anos no IME. E também ao professor Alfredo e ao João por toda ajuda e apoio no desenvolvimento deste trabalho.

Resumo

Lucas Moretto da Silva. **Análise de *trade-offs* arquiteturais avaliando técnicas de integração entre microsserviços**. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2022.

O mundo de software atual está focado cada vez mais no desenvolvimento de aplicações distribuídas. Nesse contexto, arquitetura de microsserviços é um tema cada vez mais em evidência. Esse estilo de arquitetura traz diversas vantagens para os desenvolvedores e para a aplicação, mas, carrega consigo diversas responsabilidades que não podem ser ignoradas. A integração e comunicação entre microsserviços é uma dessas responsabilidades e, se bem desenvolvida, proporciona a independência e autonomia do sistema. Além disso, tendo em vista a complexidade de uma aplicação estruturada em microsserviços, podemos dizer que padrões de arquitetura são essenciais para o desenvolvimento bem-sucedido de sistemas baseados nesse estilo. Este trabalho avalia dois estilos de comunicação entre microsserviços – síncrono com a implementação REST, e assíncrono com AMQP –, e avalia os *trade-offs* na estruturação de uma aplicação. Para a análise, foram considerados os seguintes atributos estruturais de microsserviços: tamanho do serviço, compartilhamento de bases de dados entre módulos, nível de acoplamento, desempenho e heterogeneidade de tecnologias disponíveis para o desenvolvimento. Sendo assim, implementou-se um recorte de uma aplicação hipotética chamada Pingr para mostrar a influência (negativa, neutra ou positiva) exercida por um padrão sobre os atributos mencionados. Os resultados obtidos foram: ambos os padrões têm influência neutra sobre os atributos 'tamanho do serviço', 'compartilhamento de bases de dados entre módulos' e 'heterogeneidade de tecnologias disponíveis para desenvolvimento' (levando em conta o levantamento de linguagens de programação apresentado neste trabalho); o padrão de comunicação síncrono tem influência negativa sobre os atributos 'nível de acoplamento' e 'desempenho', enquanto o padrão assíncrono tem influência positiva.

Palavras-chave: Microsserviços. Integração. REST. AMQP. Análise de trade-offs.

Abstract

Lucas Moretto da Silva. **Analysis of architectural trade-offs evaluating integration techniques between microservices**. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2022.

Today's software world is increasingly focused on developing distributed applications. In this context, microservices architecture is a topic that is increasingly in evidence. This architectural style brings several advantages to developers and to the application, but carries with it several responsibilities that cannot be ignored. The integration and communication between microservices is one of these responsibilities and, if well developed, provides the independence and autonomy of the system. Furthermore, given the complexity of an application structured in microservices, we can say that architectural patterns are essential for the successful development of systems based on this style. This work evaluates two styles of communication between microservices – synchronous with the REST implementation, and asynchronous with AMQP –, and evaluates the trade-offs in structuring an application. For the analysis, the following structural attributes of microservices were considered: service size, database sharing between modules, service coupling level, performance and heterogeneity of technologies available for development. Therefore, a clipping of a hypothetical application called Pingr was implemented to show the influence (negative, neutral or positive) exerted by a pattern on the mentioned attributes. The results obtained were: both patterns have a neutral influence on the attributes 'service size', 'databases sharing between modules' and 'heterogeneity of technologies available for development' (taking into account the survey of programming languages presented in this work); the synchronous communication pattern has a negative influence on the 'coupling level' and 'performance' attributes, while the asynchronous pattern has a positive influence.

Keywords: Microservices. Integration. REST. AMQP. Trade-off analysis.

Lista de abreviaturas

ADS	Grau de dependência do serviço (*Absolute Dependence of the Service*)
AIS	Grau de importância do serviço (*Absolute Importance of the Service*)
AMQP	Protocolo avançado de enfileiramento de mensagens (*Advanced Message Queuing Protocol*)
API	Interface de Programação de Aplicações (*Application Programming Interface*)
gRPC	Chamada de procedimento remoto da Google (*Google Remote Procedure Call*)
HTTP	Protocolo de Transferência de Hipertexto (*Hypertext Transfer Protocol*)
IME	Instituto de Matemática e Estatística
JDK	Kit de Desenvolvimento Java (*Java Development Kit*)
MSA	Arquitetura de microsserviços (*Microservices Architecture*)
REST	Transferência Representacional de Estado (*Representational State Transfer*)
SOAP	Protocolo Simples de Acesso a Objetos (*Simple Object Access Protocol*)
UML	Linguagem Unificada de Modelagem (*Unified Modeling Language*)
URL	Localizador Uniforme de Recursos (*Uniform Resource Locator*)
USP	Universidade de São Paulo

Lista de figuras

Lista de tabelas

Sumário

Capítulo 1

Introdução

1.1 Motivação

O conceito de microsserviço foi criado por Fowler e Lewis em 2014 (Fowler e Lewis, 2014), e desde então, o interesse nesse estilo arquitetural cresceu e foi amplamente adotado na indústria de software. O estilo de arquitetura de microsserviços (MSA) é uma abordagem que visa permitir o desenvolvimento de software complexo "como um conjunto de pequenos serviços, cada um executando seu próprio processo e se comunicando com mecanismos leves" (Fowler e Lewis, 2014). No entanto, desenvolver sistemas baseados em microsserviços é uma tarefa complexa. Dentre os principais atributos estruturais, inerentes a essa arquitetura, que contribuem para o aumento da complexidade são identificar o tamanho dos serviços, definir o nível de acoplamento dos serviços e gerenciar a integração e comunicação entre eles.

A arquitetura de microsserviços ainda é um estilo de desenvolvimento muito recente. Os padrões e tecnologias atuais não solucionam todos os problemas da arquitetura, e estes, quando existem, também são recentes e necessitam de novas técnicas. Tendo em vista a complexidade de uma aplicação estruturada em microsserviços, podemos dizer que padrões de arquitetura são essenciais para o desenvolvimento bem-sucedido de sistemas baseados nesse estilo (Marquez e Astudillo, 2018).

Embora os microsserviços possam existir e funcionar de forma independente, em muitas situações precisam interagir e se comunicar com outros microsserviços para realizar algum processo. Isso requer o desenvolvimento de APIs funcionais, simples e bem definidas que atuem como canal de comunicação entre vários serviços que constituem a aplicação.

Além disso, como uma mesma aplicação costuma ser composta por diversos microsserviços, a comunicação entre eles pode ficar bastante intensa para a rede e consumir muitos recursos. Questões como latência e acoplamento entre microsserviços podem tornar a solução adotada ainda mais complexa.

1.2 Objetivos

O objetivo geral deste trabalho consiste em apresentar argumentos e resultados que auxiliem a decisão de engenheiros de software quando estes estiverem escolhendo o padrão de comunicação que será utilizado no desenvolvimento de um sistema com arquitetura de microsserviços.

1.3 Metodologia

Nesta seção, apresentamos a metodologia adotada neste trabalho. Inicialmente, começamos definindo qual seria o objetivo da análise que seria realizada. Delimitamos o escopo da pesquisa como sendo a identificação das influências que os padrões de comunicação síncrono e assíncrono têm sobre uma aplicação estruturada em uma arquitetura de microsserviços. Para possibilitar a análise decidimos desenvolver uma mesma aplicação sob os dois paradigmas de comunicação mencionados, onde optamos por utilizar a implementação REST no padrão síncrono, e AMQP no assíncrono.

A partir disso, começamos a explorar a literatura para estabelecer uma compreensão bem fundamentada do que é uma arquitetura em microsserviços, seus atributos estruturais e os padrões de desenvolvimento e integração entre serviços. Também procuramos levantar fontes que apresentassem métodos bem definidos para avaliar *trade-offs* arquiteturais em MSA.

Além de responder algumas perguntas sobre MSA, a pesquisa inicial nos ajudou a identificar alguns atributos estruturais de microsserviços que sofrem alterações dependendo da solução de arquitetura utilizada. Dentre eles, selecionamos alguns para serem aplicados à análise de *trade-offs* deste trabalho:

- Tamanho do serviço/módulo
- Compartilhamento de bases de dados entre módulos
- Nível de acoplamento entre serviços
- Desempenho
- Heterogeneidade de tecnologias disponíveis para desenvolvimento

Com o objetivo e escopo da análise bem definidos, procuramos por uma ideia de aplicação que pudesse ser implementada em tempo hábil sob os dois padrões arquiteturais previamente estabelecidos e que apresentasse alguma projeção de valor real em seu contexto. Assim, optou-se por desenvolver um recorte de um sistema hipotético chamado Pingr (Verão - IME USP, 2021), que representa uma rede social no formato de microblog.

Por fim, com as duas soluções do sistema implementadas e utilizando o método para análise de *trade-offs* de al, 2020b em conjunto com o modelo de caracterização definido em al, 2020a, levantamos métricas que possibilitaram avaliar as influências geradas pelos padrões de integração nos atributos de software previamente estabelecidos.

1.4 Resultados obtidos

Neste trabalho, conduzimos uma análise para identificar a influência (negativa, neutra ou positiva) que os padrões de comunicação síncrono e assíncrono têm sobre atributos estruturais: tamanho dos serviços/módulos, compartilhamento de bases de dados entre módulos, nível de acoplamento entre serviços, desempenho das operações, heterogeneidade de tecnologias disponíveis para desenvolvimento.

Os resultados de influência podem ser sumarizados na tabela 1.1:

Padrão de comunicação	Tam. do serviço	Comp. de bases de dados	Nível de acoplamento	Desempenho	Het. de tec.
Assíncrono	Neutra	Neutra	Positiva	Positiva	Neutra
Síncrono	Neutra	Neutra	Negativa	Negativa	Neutra

Tabela 1.1: *Influência exercida pelos padrões de comunicação nos atributos estruturais de microsserviços*

Acreditamos que esse resultado possa ser identificado nos mais diversos sistemas com MSA, sempre que a solução proposta esteja adequada com o modelo de comunicação utilizado na arquitetura. Também acreditamos que os resultados aqui apresentados possam ajudar a guiar arquitetos e engenheiros de software na tomada de decisão em seus projetos.

1.5 Organização do texto

Este trabalho está organizado da seguinte forma. O capítulo 2 apresenta uma visão geral do estilo de arquitetura dos microsserviços, suas vantagens e desvantagens, assim como alguns outros conceitos e modelos abordados ou utilizados para a análise. O capítulo 3 descreve o método de pesquisa e desenvolvimento do trabalho. O capítulo 4 descreve a aplicação implementada para análise. O capítulo 5 expõe a análise de *trade-offs*. O capítulo 6 apresenta nossas considerações finais.

Capítulo 2

Fundamentos

2.1 Arquitetura de microsserviços

A arquitetura de microsserviços (MSA) refere-se a um estilo arquitetural para o desenvolvimento de aplicações. Neste estilo arquitetural, o sistema é desenvolvido em diversas partes pequenas, cada qual executada em um processo isolado - essa parte recebe o nome de "microsserviço". Esse tem seu escopo definido através de um único domínio de responsabilidade. Nesse estilo de arquitetura, os microsserviços comunicam-se entre si por meio de interfaces simples e leves para resolver problemas de negócios, montar respostas provenientes de diferentes partes do sistema, etc.

A arquitetura de microsserviços foi desenvolvida para superar as dificuldades apresentadas por abordagens arquiteturais monolíticas. A arquitetura monolítica é caracterizada pelo desenvolvimento da aplicação em um único código-fonte que é executado em um único processo contendo todos os componentes de software de um aplicativo: interface de usuário, camada de negócios e interface de dados. Construir uma aplicação como uma única unidade traz diversas limitações, como inflexibilidade, falta de confiabilidade, dificuldade de dimensionamento, desenvolvimento lento e assim por diante. Foi para contornar esses problemas que a arquitetura de microsserviços foi criada.

A divisão de uma aplicação em microsserviços permite um desenvolvimento mais rápido, detecção e resolução de bugs de maneira mais eficiente, manutenção mais simples, flexibilidade e maior disponibilidade e escalabilidade.

2.1.1 Características de microsserviços

Autonomia

Cada serviço pertencente a uma arquitetura de microsserviços deve ser desenvolvido, implantado, operado e dimensionado sem afetar o funcionamento dos demais. Os componentes não precisam compartilhar nenhum código ou implementação uns com os outros. Qualquer comunicação entre microsserviços acontece diretamente por meio de APIs simples e bem definidas ou indiretamente por infraestruturas de mensageria do tipo "*dumb pipe*" .

Especificação

Cada serviço é projetado com um conjunto de recursos e se concentra na solução de um problema específico. Se os desenvolvedores contribuírem com mais código para um serviço ao longo do tempo e o serviço se tornar complexo, ele pode ser dividido em serviços menores.

Agilidade

Os microsserviços promovem a organização de um time de desenvolvimento em equipes pequenas e independentes que se apropriam de seus serviços. As equipes agem dentro de um contexto pequeno e bem compreendido, e têm o poder de trabalhar de forma mais independente e rápida, ajudando a reduzir o tempo de desenvolvimento.

Elasticidade

Os microsserviços permitem que cada serviço seja dimensionado independentemente para atender à demanda de recursos do aplicativo. Isso permite que as equipes dimensionem as necessidades de infraestrutura, meçam com precisão o custo de um recurso e mantenham a disponibilidade se um serviço tiver um pico de demanda e reduzam a oferta em momentos de baixa demanda.

Fácil implantação

Os microsserviços permitem integração e entrega contínuas, facilitando a experimentação de novas ideias e a reversão se algo não funcionar. O baixo custo da falha permite a experimentação, facilita a atualização do código e acelera o tempo de entrega de novas releases.

Liberdade de tecnologias

Dentro de uma arquitetura monolítica, assim que a *stack* de tecnologias da aplicação é definida, ela terá de ser carregada e preservada por meio de manutenções até o fim do desenvolvimento. Pode ser que a escolha das ferramentas utilizadas não beneficiem parte das funcionalidades do sistema, mas, como o monolito é um artefato único, a equipe de desenvolvimento fica presa a decisão feita no começo do projeto.

Numa arquitetura de microsserviços as equipes têm a liberdade de escolher a melhor ferramenta para resolver seus problemas ou desenvolver as funcionalidades da aplicação. Como consequência, as equipes que criam microsserviços podem sempre escolher a melhor ferramenta para cada trabalho.

Código reutilizável

Dividir o software em módulos pequenos e bem definidos permite que as equipes reutilizem funções para diversos propósitos. Um serviço escrito para uma determinada função pode ser usado como um bloco de construção para outro recurso.

Resiliência

A independência de cada serviço aumenta a resistência a falhas de uma aplicação. Em uma arquitetura monolítica, se um único componente falhar, pode causar a falha de todo

o sistema. Com os microsserviços, os componentes lidam com a falha total do serviço degradando a funcionalidade e não travando a aplicação inteira.

Existem diversas estratégias que podem ser utilizadas para garantir a tolerância a falhas em microsserviços. Por exemplo, pode-se utilizar uma estratégia de abertura e fechamento de circuito (*circuit breaking*) onde, caso ocorra falhas no sistema, "abre-se" o circuito, impedindo comunicação entre os serviços que falharam por um determinado período de tempo. O sistema continua monitorando o ponto de falha e, quando as coisas voltarem ao normal, o circuito é "fechado" para restabelecer a funcionalidade padrão.

Pode-se utilizar estratégias de *timeouts*, que definem que não se deve esperar por uma resposta de serviço por um período de tempo indefinido — lance uma exceção em vez de esperar muito. Isso garante que o sistema não fique preso em um estado intermediário, continuando a consumir recursos do aplicativo. Quando o tempo limite for atingido, a comunicação é liberada novamente.

Além disso, existem estratégias mais simples, como utilização de *retry*, que simplesmente indica para tentar novamente uma conexão assim que a falha acontece. Isso é muito útil em caso de problemas temporários com um dos microsserviços.

2.1.2 Desafios inerentes a arquitetura de microsserviços

Complexidade da solução

Na arquitetura de microsserviços uma aplicações possui muito mais componentes e camadas que funcionam em conjunto do que um monolito equivalente. Cada microsserviço é mais simples, mas o sistema como um todo é mais complexo de se desenvolver e manter.

Integração/Comunicação entre microsserviços

Embora os microsserviços possam existir e funcionar de forma independente, em muitas situações precisam interagir e se comunicar com outros microsserviços para realizar algum processo. Isso requer o desenvolvimento de APIs funcionais, simples e bem definidas que atuem como canal de comunicação entre vários serviços que constituem a aplicação.

Além disso, como existirão diversos microsserviços, a comunicação entre eles pode ficar bastante intensa para a rede e consumir muitos recursos. Outro problema que pode surgir é a questão da latência de resposta. Se a cadeia de dependências do serviço ficar muito longa (o serviço 1 chama 2, que chama 3...), a latência adicional pode se tornar um problema para a aplicação.

Logs distribuídos

Microsserviços são aplicações independentes e cada um deles geralmente possui um mecanismo de log distinto. Por consequência grandes volumes de dados de log são gerados de forma não centralizada e não estruturada, o que pode acarretar em problemas para organizar e manter os registros de uma mesma aplicação.

Dependências cíclicas entre serviços

Uma dependência cíclica refere-se ao acoplamento entre dois ou mais serviços na aplicação. Pode ser que durante o desenvolvimento existam serviços tão dependentes um do outro que podem deixar a solução mais complexa e dificultar a manutenção da aplicação.

Garantir consistência de dados

Cada microsserviço é responsável por seu próprio mecanismo de persistência de dados, com seus próprios banco de dados independentes, etc. Como resultado, manter a consistência dos dados entre diferentes microsserviços pode ser um desafio.

2.2 Padrões de comunicação em microsserviços

Serviços podem se comunicar por meio de diferentes formas, cada uma visando cenários e objetivos diferentes. Podemos classificar as formas de comunicação em três eixos: comunicação síncrona, comunicação assíncrona e híbrida.

2.2.1 Comunicação síncrona

Protocolo síncrono é aquele no qual os dados são transmitidos de um serviço a outro através de uma interação blocante. Geralmente essa comunicação é feita utilizando REST ou gRPC. Dizemos que é blocante pois, na comunicação entre dois serviços, o cliente envia uma requisição e aguarda uma resposta do serviço. Isso porque o protocolo utilizado (HTTP/HTTPS) é síncrono, então o código cliente só pode continuar sua execução quando receber a resposta do servidor HTTP.

2.2.2 Comunicação assíncrona

Na comunicação assíncrona, um cliente envia uma requisição, mas não para sua execução para esperar uma resposta. Geralmente esse tipo de comunicação é feita utilizando-se o protocolo AMQP (*Advanced Message Queuing Protocol*).

O AMQP foi criado como um protocolo aberto que permite a interoperabilidade de mensagens entre sistemas, independente do cliente produtor das mensagens e da plataforma utilizada. Além de definir o protocolo da camada de rede da aplicação, o AMQP também especifica uma arquitetura de alto nível para componentes intermediários das mensagens.

Nessa definição estão especificados um conjunto de recursos que devem ser disponibilizados por uma implementação de um servidor (chamado *broker*) compatível com AMQP (como RabbitMQ), regras de como as mensagens devem ser processadas, roteadas, armazenadas, etc.

O funcionamento de um sistema seguindo o protocolo AMQP ocorre da seguinte forma: um cliente chamado produtor envia uma mensagem para o *broker*. Estes distribuem cópias das mensagens para filas, dependendo das regras definidas pelo mesmo e da chave

fornecida na mensagem. Por fim, a mensagem é consumida por um cliente chamado consumidor.

2.2.3 Comunicação híbrida

Entre dois serviços podemos ter apenas um tipo de comunicação de cada vez (ou síncrona ou assíncrona). Porém, uma aplicação arquitetada seguindo o estilo de microsserviços pode ser composta por uma vasta coleção de componentes.

Sendo assim, uma aplicação baseada em microsserviços geralmente usa uma combinação desses estilos de comunicação, variando a utilização de acordo com os cenários e especificações de negócio.

2.3 Modelo para caracterização e evolução de sistemas com arquitetura baseada em serviços (CharM)

O CharM é um modelo que auxilia na caracterização e evolução da arquitetura de sistemas baseados em diretrizes de microsserviços (AL, 2020a; OLIVEIRA ROSA, 2021a; OLIVEIRA ROSA, 2021b).

Tal modelo ajuda a identificar padrões arquiteturais que influenciam um determinado conjunto de atributos de qualidade. Para isso, ele apresenta quatro dimensões que medem características estruturais da arquitetura:

- **Tamanho de módulos:** esta dimensão caracteriza a composição dos módulos considerando serviços e operações. O objetivo é identificar quais critérios devem ser adotados para limitar o escopo dos serviços e para agrupar serviços em módulos;
- **Compartilhamento de bases de dados entre módulos:** esta dimensão caracteriza a distribuição de base de dados entre os módulos, apontando se uma base de dados é utilizada por um ou mais módulos. O objetivo é auxiliar na escolha da estratégia mais adequada para compartilhamento das bases de dados do sistema;
- **Acoplamento síncrono entre os serviços:** esta dimensão caracteriza o grau de dependência síncrona entre os serviços do sistema, onde, durante as interações, os serviços ficam bloqueados e aguardando por uma resposta. Sua finalidade é guiar a construção do esquema de comunicação síncrona do sistema verificando o grau de importância e dependência de cada um de seus módulos;
- **Acoplamento assíncrono entre os serviços:** esta dimensão caracteriza o grau de dependência assíncrona entre os serviços do sistema, onde, durante as interações, os serviços não ficam bloqueados esperando por uma resposta. Sua finalidade é guiar a construção do esquema de comunicação assíncrona do sistema verificando o grau de importância e dependência de cada um de seus módulos.

Essas dimensões avaliam os mesmos atributos estruturais de microsserviços que este trabalho estuda. Além do mais, o modelo define métricas comparativas para cada uma das dimensões e, em conjunto com o trabalho de AL, 2020b, pode ser utilizado para analisar a

influência (negativa, neutra ou positiva) que um determinado padrão de arquitetura causa nos atributos estipulados.

Utilizaremos o CharM e suas métricas no capítulo 5 durante a análise dos *trade-offs* gerados quando comparados os atributos tamanho do serviço, nível de acoplamento e compartilhamento de bases de dados entre módulos.

Capítulo 3

Metodologia

3.1 Descrição do método de pesquisa

De forma geral, podemos enumerar os passos seguidos para o desenvolvimento deste trabalho na seguinte ordem:

1. Definimos o objetivo da análise, delimitando o escopo da pesquisa, os estilos e padrões de arquitetura que serão estudados e que tipos de *trade-offs* serão analisados;
2. Selecionamos as fontes e bibliografia relacionadas ao tipo de arquitetura investigada;
3. Selecionamos os atributos de software para análise de *trade-offs*;
4. Verificamos se os atributos de software selecionados são aplicáveis sobre os padrões de arquitetura levantados para análise;
5. Desenvolvemos aplicações para mostrar a influência (negativa, neutra ou positiva) exercida por um padrão particular nos atributos de qualidade de software;
6. Para cada padrão selecionado, verificamos os *trade-offs*.

Assim, estabelecemos que a análise teve por objetivo identificar a influência que os padrões de comunicação síncrono e assíncrono têm sobre uma aplicação estruturada em uma arquitetura de microsserviços. Como forma de integração síncrona será utilizado o protocolo HTTP (através de REST) e a assíncrona será AMQP (através do RabbitMQ). Para possibilitar a análise foi desenvolvida uma mesma aplicação sob os dois paradigmas de comunicação mencionados.

Exploramos a literatura para estabelecer uma compreensão bem fundamentada sobre MSA, seus padrões e particularidades. Procuramos por referências no Google usando palavras-chave como *"microservice"*, *"communication pattern"*, *"architectural pattern"*, *"messaging protocol"*, *"synchronous communication"*, *"asynchronous communication"*. Além disso, algumas fontes previamente conhecidas pelo autor do trabalho também foram selecionadas. O critério usado para a seleção da fonte foi sua disponibilidade para aquisição e qualidade da descrição. Assim, as bibliografias utilizadas foram:

- *"Building microservices: designing fine-grained systems"*, Newman, 2015;

- *"Microservices Patterns"*, Richardson, 2018;
- *"Microservices"*, Fowler e Lewis, 2014;
- *"Software Architecture Patterns"*, Richards, 2015;
- *"Communication in a microservice architecture"*, Wagner e Rousos, 2022;
- *"Deployment and communication patterns in microservice architectures: A systematic literature review"*, Aksakalli *et al.*, 2021;
- *"Integrating Microservices"*, Indrasiri e Siriwardena, 2018.

Com base nas fontes selecionadas e outros trabalhos que estudam desafios relacionados a microsserviços (Bonér, 2016; Fowler e Lewis, 2014; Hassan e Bahsoon, 2016; Newman, 2015; Jacopo Soldani e Heuvel, 2018), identificamos que os principais atributos estruturais que sofrem alterações dependendo da solução de arquitetura utilizada são: identificação do tamanho dos serviços e módulos, gerenciamento de consistência de dados (compartilhamento de banco de dados entre serviços) e definição do nível de acoplamento entre serviços.

Além disso, como abordado na seção 2.1.2, a liberdade tecnológica é uma das vantagens inerentes à arquitetura de microsserviços (Fowler e Lewis, 2014). É interessante entender como a escolha de uma padrão de comunicação pode impactar na heterogeneidade de tecnologias disponíveis para o desenvolvimento de um sistema.

Por fim, diferentes formas de comunicação entre microsserviços possibilitam a criação de diferentes soluções para um mesmo problema (Fowler e Lewis, 2014; Bhojwani, 2018). Diferentes soluções podem acarretar em benefícios ou malefícios para o desempenho de uma aplicação.

Por conta desses pontos, selecionamos os seguintes atributos estruturais de microsserviços para aplicar a análise de *trade-offs*:

- Tamanho do serviço/módulo;
- Compartilhamento de bases de dados entre módulos;
- Nível de acoplamento entre serviços;
- Desempenho das operações;
- Heterogeneidade de tecnologias disponíveis para desenvolvimento.

O desenvolvimento da aplicação e as análises de *trade-offs* são abordadas em detalhes nos capítulos 4 e 5, respectivamente.

Capítulo 4

Aplicação desenvolvida

4.1 Contextualização do escopo

A aplicação desenvolvida será baseada no contexto e especificação do sistema Pingr[1]. De acordo com sua documentação, o Pingr é um sistema hipotético que foi criado como projeto para o curso **Arquitetura Ágil de Software**, ministrado durante o Programa de Verão - IME-USP - 2021[2]. Posteriormente foi utilizado como projeto em MAC0475 - Laboratório de Sistemas Computacionais Complexos oferecida pelo Departamento de Ciência da Computação - IME-USP.

O sistema representa uma rede social no formato de microblog, onde os usuários podem postar atualizações/mensagens limitadas a 140 caracteres. O domínio do sistema utilizado abrange toda a lógica de negócio e operacional que a empresa necessita para operacionalizar as interações entre usuários da rede.

Para o contexto desse trabalho utilizaremos como referência a proposta de arquitetura disponibilizada pelos professores responsáveis pelo curso. Essa proposta nos é interessante por utilizar o padrão arquitetural de microsserviços que é necessário para nossa análise.

Como definido anteriormente, este trabalho tem por objetivo analisar as técnicas de integração entre os serviços e os *trade-offs* arquiteturais gerados por suas utilizações. Desta forma, será feita a implementação da lógica de negócio de cada serviço, mas daremos mais enfoque nas interfaces de comunicação (recebimento e envio de mensagens) utilizando REST e AMQP como diferentes técnicas de integração.

[1] https://pingr-complexos.netlify.app/

[2] https://www.ime.usp.br/~verao/index.php

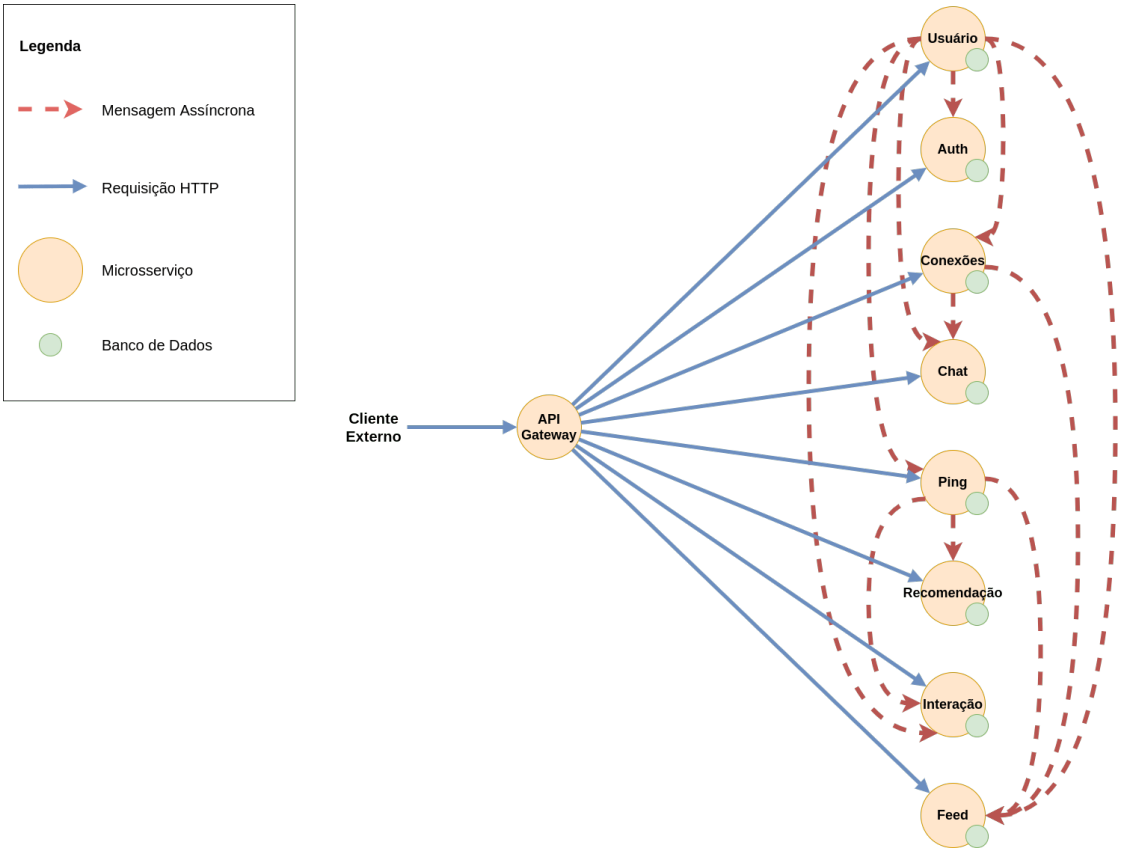

Figura 4.1: *Visão de microsserviços panorâmica do sistema Pingr. (Figura retirada de* Verão - IME USP, 2021*)*

4.1.1 Recorte

Para as análises deste trabalho implementaremos um recorte do sistema Pingr. Esse focará na funcionalidade de envio de mensagem entre usuários. Para isso, serão desenvolvidos microsserviços que se responsabilizarão pelas conexões entre usuários (estabelecimento de amizade) e chat (troca direta de mensagens entre usuários amigos).

4.1.2 Funcionalidades gerais

De forma geral, as funcionalidades de conexão e chat podem ser descritas pelos diagramas 4.2 e 4.3. As diferenças de fluxo causadas pelo escolha da forma de integração (síncrona ou assíncrona) serão discutidas nas seções posteriores.

Em detalhes, para que um usuário possa enviar uma mensagem para outro, é necessários a existência de uma conexão (amizade) entre eles. Caso essa conexão exista, então um usuário pode trocar mensagens com o outro através do sistema. Caso contrário, então essa conexão deve ser estabelecida para que ambos consigam se comunicar pelo sistema.

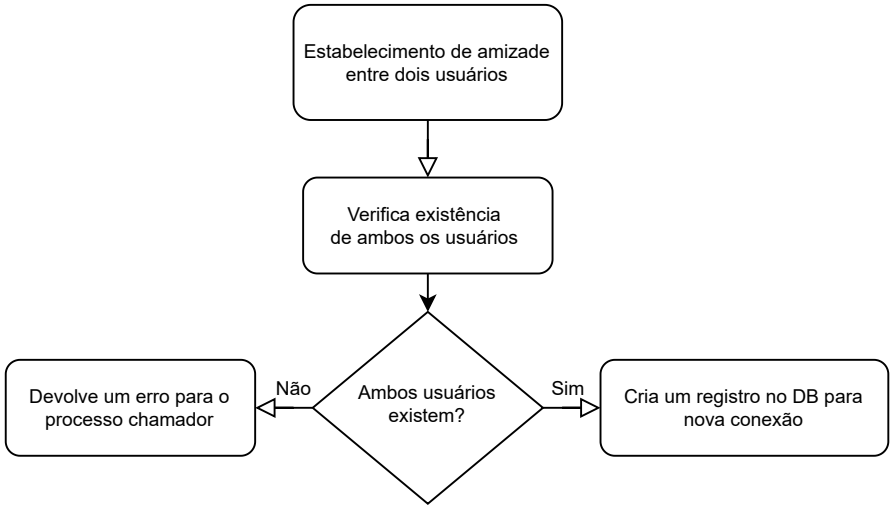

Figura 4.2: *Processo para estabelecimento de amizade entre dois usuários.*

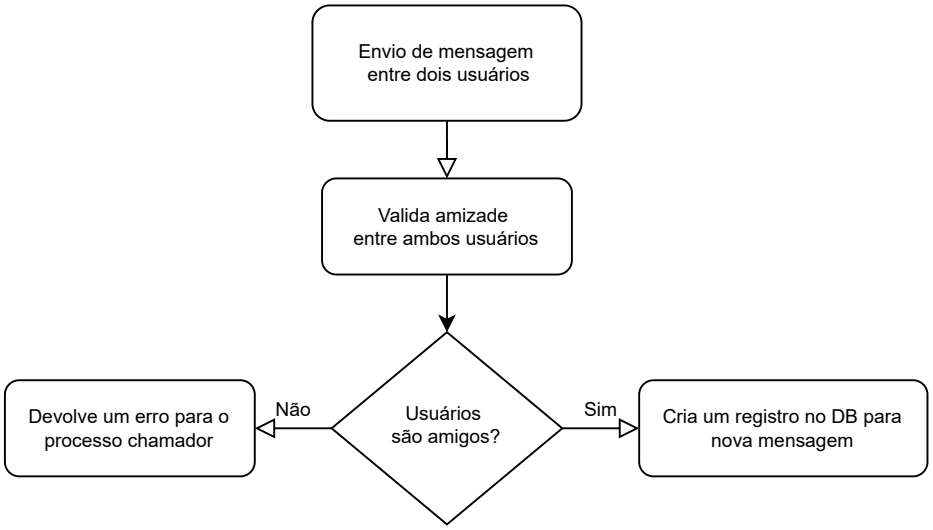

Figura 4.3: *Processo para envio de mensagem entre dois usuários.*

4.1.3 Microsserviços

As funcionalidades descritas são realizadas por dois microsserviços:

- *Conexões*: gerencia a relação entre usuários, permitindo a solicitação e resolução de solicitação de amizade, além de visualizar e gerenciar a lista de amigos;
- *Chat*: gerencia o conteúdo privativo, permitindo criação de novas conversas e leitura e envio de novas mensagens entre usuários.

4.1.4 Tecnologias utilizadas

Para a implementação dos serviços, utilizou-se a linguagem de programação *Kotlin*[3] na versão JDK 1.8, em conjunto com o framework *Spring Boot*[4] na versão 2.7.3. Os bancos de dados foram criados em PostgreSQL.

4.2 Solução assíncrona

4.2.1 Arquitetura

Seguindo a arquitetura de microsserviços, esta solução apresenta dois microsserviços (*async-ms-chat* e *async-ms-connection*) que se comunicam de forma assíncrona através de mensagens no RabbitMQ. Cada microsserviço possui o seu próprio banco de dados e é responsável por armazenar os dados que são necessários para seu funcionamento.

O código desta solução pode ser acessado em https://github.com/LucasMorettoSilva/pingr-async.

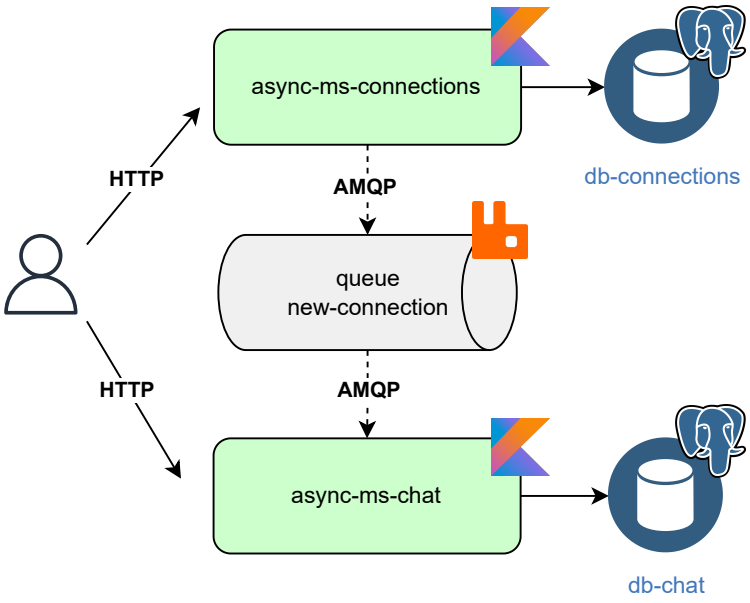

Figura 4.4: *Arquitetura da solução assíncrona*

4.2.2 Microsserviço *async-ms-connection*

Este microsserviço gerencia a relação entre usuários, permitindo a solicitação e resolução de solicitação de amizade, além de visualizar e gerenciar a lista de amigos de um usuário. Isso é possível pois esse módulo conta com um banco de dados próprio chamado *db-connections*. Tal banco possui duas tabelas:

- *users*: representa um usuário em si;

[3] https://kotlinlang.org/docs/home.html

[4] https://spring.io/projects/spring-boot

- *friendships*: representa relações de amizade entre usuários.

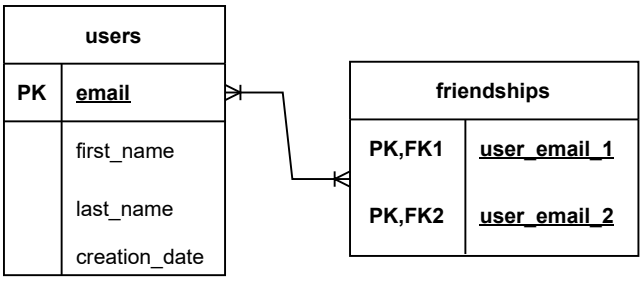

Figura 4.5: *Diagrama de entidades no modelo UML do banco de dados db-connections na solução assíncrona*

O microsserviço expõe uma API Rest através do *FriendshipController*. Por meio dela podemos realizar operações para obter todas as relações de amizade existentes no banco de dados, criar uma nova relação de amizade entre usuários, verificar todas as amizades de um determinado usuário, etc.

Esse módulo também possui um produtor/publicador de mensagens AMQP chamado *EstablishFriendshipAmqpProducer*. Esse é responsável por publicar um evento numa fila do *RabbitMQ* sempre que uma nova amizade é estabelecida. Dessa forma, os serviços interessados nessa operação podem simplesmente escutar/consumir dessa fila. Isso é feito pelo microsserviço *async-ms-chat* e, por conta disso, nessa arquitetura temos uma menor dependência entre os dois microsserviços quando tentamos enviar uma mensagem de um usuário para outro.

As demais classes do microsserviço foram desenvolvidas de forma a terem comportamentos bem definidos e encapsulados. Para entender o funcionamento do microsserviço de forma mais geral pode-se visualizar o UML da figura 4.6.

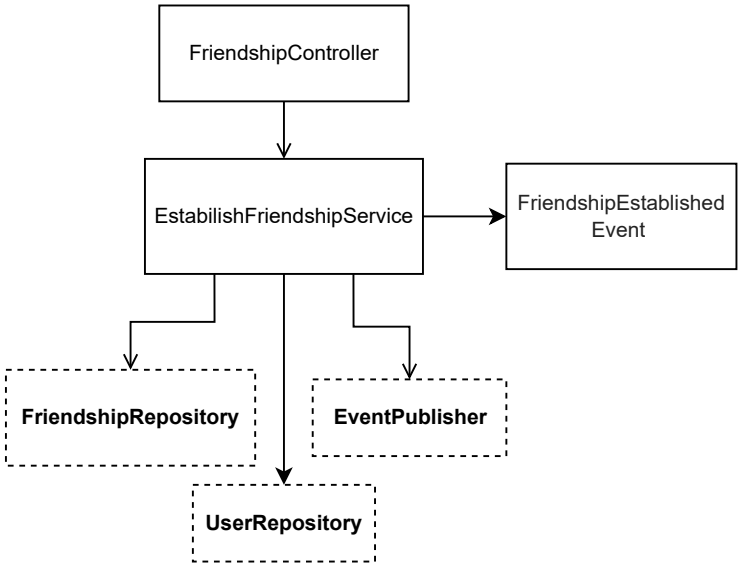

Figura 4.6: *UML das principais classes do microsserviço async-ms-connection*

- *FriendshipController*: responsável por expor a API pública do microsserviço;
- *EstabilishFriendshipService*: modela o comportamento de criação de amizade entre dois usuários;
- *UserRepository*: define as transações relacionadas à entidade Usuário;
- *FriendshipRepository*: define as transações relacionadas à entidade Amizade;
- *EventPublisher*: define as operações de publicação de eventos;
- *FriendshipEstablishedEvent*: representa o evento de estabelecimento de amizade.

4.2.3 Microsserviço *async-ms-chat*

Este microsserviço gerencia o conteúdo privativo, permitindo criação de novas conversas e leitura e envio de novas mensagens entre usuários amigos.

Esse módulo conta com um banco de dados próprio chamado *db-chat*. Tal banco possui quatro tabelas:

- *chats*: representa um chat em si;
- *chats_users*: representa relações entre chat e usuários, ou seja, quais usuários pertencem a um determinado chat;
- *messages*: representa uma mensagem enviada em um chat;
- *friendships*: representa relações de amizade entre usuários.

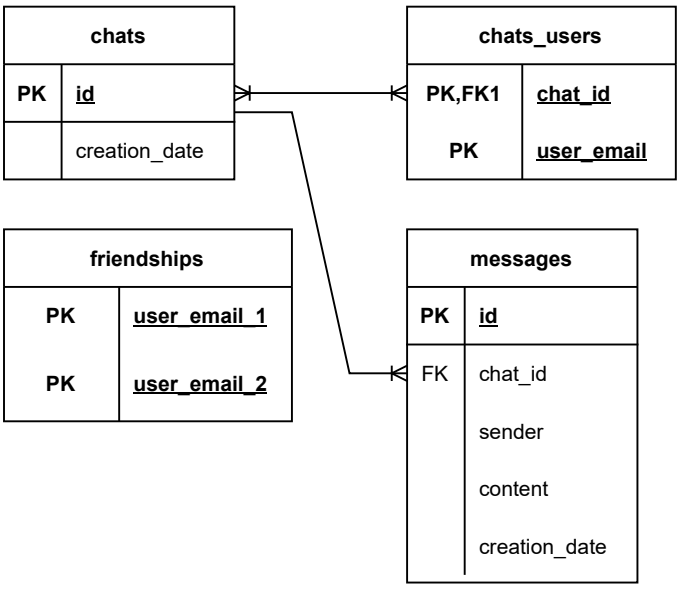

Figura 4.7: *Diagrama de entidades no modelo UML do banco de dados db-chat na solução assíncrona*

O microsserviço expõe uma API REST através do *ChatController*. Por meio dela podemos realizar operações para enviar mensagens entre usuários, obter as informações de um determinado chat, etc.

Esse módulo também possui um consumidor de mensagens AMQP chamado *EstablishFriendshipAmqpConsumer*. Ele é responsável por consumir eventos da fila de amizade do RabbitMQ. Sendo assim, sempre que uma nova amizade é criada, o microsserviço recebe os dados pertinentes e os armazena na tabela *friendships*. Com isso, eliminamos a dependência com o serviço de conexões, pois esse módulo conterá todas as informações necessárias para o envio de mensagens e criação de chats quando comparado com a solução síncrona vista na próxima seção.

As demais classes do microsserviço foram desenvolvidas de forma a terem comportamentos bem definidos e encapsulados. Para entender o funcionamento do microsserviço de forma mais geral pode-se visualizar o UML da figura 4.8.

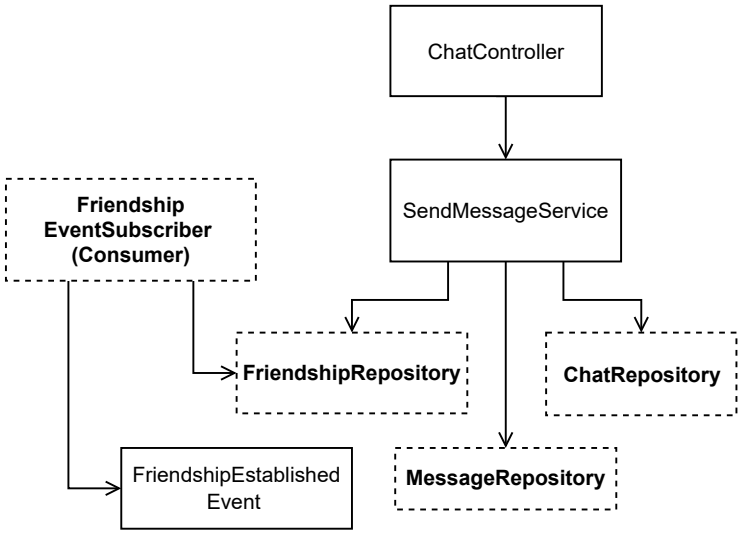

Figura 4.8: *UML das principais classes do microsserviço async-ms-chat*

- *ChatController*: responsável por expor a API pública do microsserviço;
- *SendMessageService*: modela o comportamento do envio de mensagem entre dois usuários;
- *ChatRepository*: define as transações relacionadas à entidade Chat;
- *FriendshipRepository*: define as transações relacionadas à entidade Amizade;
- *MessageRepository*: define as transações relacionadas à entidade Mensagem;
- *FriendshipEventSubscriber*: define as operações de leitura de eventos;
- *FriendshipEstablishedEvent*: representa o evento de estabelecimento de amizade.

4.2.4 Diagrama de sequência

Nessa arquitetura assíncrona, podemos representar a sequência de processos para criação de amizade e envio de mensagens entre usuários através do diagrama na figura 4.9.

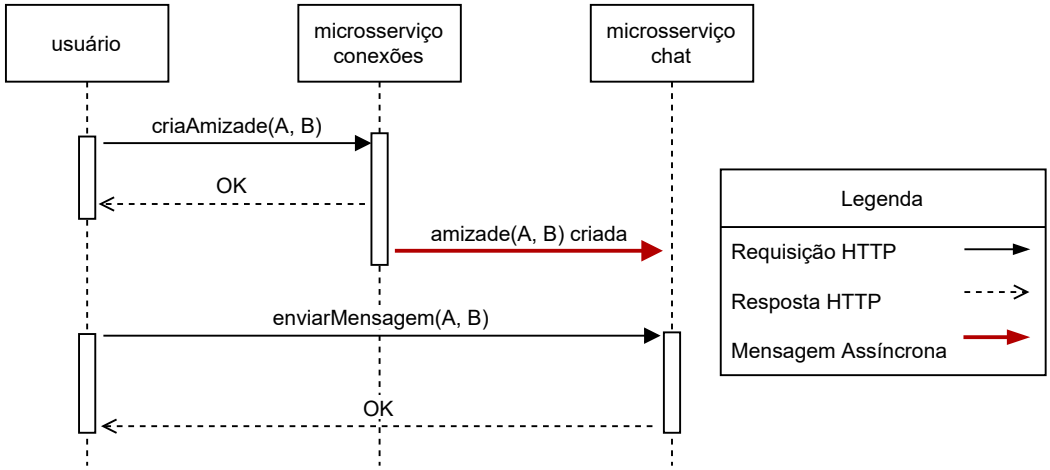

Figura 4.9: *Diagrama de sequência com funcionamento da arquitetura assíncrona*

Note que para estabelecimento de amizade temos os seguintes processos:

1. O usuário envia uma requisição para *async-ms-connection*;
2. O microsserviço *async-ms-connection* armazena essa nova relação em seu banco de dados;
3. O microsserviço *async-ms-connection* publica um evento com os dados na nova amizade estabelecida na fila *new-connection*;
4. O microsserviço *async-ms-chat*, ao consumir o evento dessa fila, armazena os dados da amizade em seu banco de dados.

Note que para o envio de mensagens entre usuários temos os seguintes processos:

1. O usuário enviar uma requisição para *async-ms-chat*;
2. O microsserviço *async-ms-chat* verifica em seu banco de dados a existência da amizade e, caso exista, armazena a mensagem de texto no banco.

Note que não existe uma relação de dependência entre os microsserviços *async-ms-connection* e *async-ms-chat* quando comparado com a arquitetura síncrona. Note também que, por conta disso, a operação de envio de mensagem torna-se muito mais simples de ser implementada e totalmente independente do serviço de conexões. Então, caso *async-ms-connection* esteja indisponível ou inacessível, o serviço de chat ainda pode funcionar normalmente. O que não ocorre na solução síncrona.

4.3 Solução síncrona

4.3.1 Arquitetura

Seguindo a arquitetura de microsserviços, esta solução apresenta dois microsserviços (*sync-ms-chat* e *sync-ms-connection*) que se comunicam de forma síncrona através de

requisições HTTPs. Cada microsserviço possui o seu próprio banco de dados e é responsável por armazenar os dados que são necessários para seu funcionamento.

O código desta solução pode ser acessado em https://github.com/LucasMorettoSilva/pingr-sync.

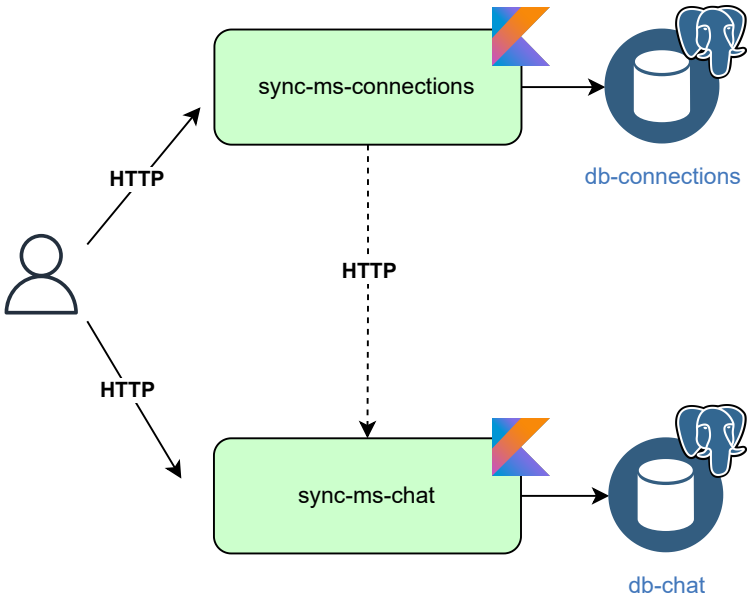

Figura 4.10: *Arquitetura da solução síncrona*

4.3.2 Microsserviço *sync-ms-connection*

Este microsserviço gerencia a relação entre usuários, permitindo a solicitação e resolução de solicitação de amizade, além de visualizar e gerenciar a lista de amigos de um usuário. Isso é possível pois esse módulo conta com um banco de dados próprio chamado *db-connection*. Tal banco possui duas tabelas:

- *users*: representa um usuário em si;
- *friendships*: representa relações de amizade entre usuários.

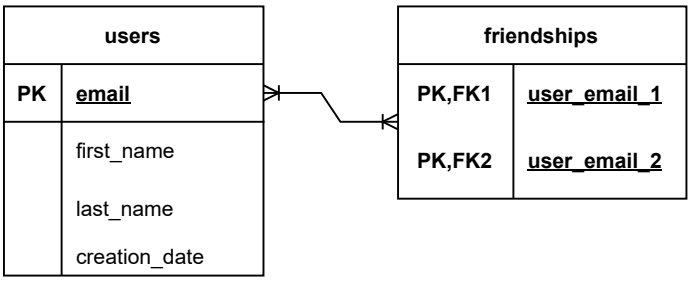

Figura 4.11: *Diagrama de entidades no modelo UML do banco de dados db-connections na solução síncrona*

O microsserviço expõe uma API REST através do *FriendshipController*. Por meio dela podemos realizar operações para obter todas as relações de amizade existentes no banco

de dados, criar uma nova relação de amizade entre usuários, verificar todas as amizades de um determinado usuário, etc.

Diferentemente da solução assíncrona, ao criar uma nova relação de amizade o microsserviço apenas armazena essa relação em seu banco de dados. Caso algum outro microsserviço precise averiguar se uma determinada amizade existe, então ele terá de realizar um requisição HTTP para a API deste microsserviço.

As demais classes do microsserviço foram desenvolvidas de forma a terem comportamentos bem definidos e encapsulados. Para entender o funcionamento do microsserviço de forma mais geral pode-se visualizar o UML na figura 4.12.

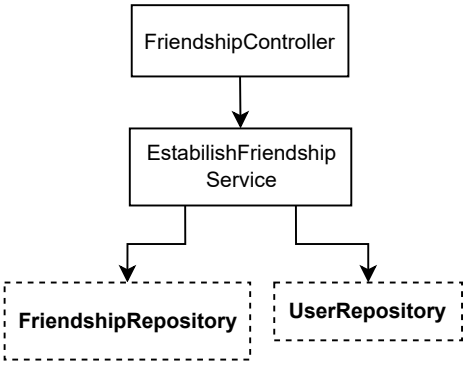

Figura 4.12: *UML das principais classes do microsserviço sync-ms-connection*

- *FriendshipController*: responsável por expor a API pública do microsserviço;
- *EstabilishFriendshipService*: modela o comportamento de criação de amizade entre dois usuários;
- *UserRepository*: define as transações relacionadas à entidade Usuário;
- *FriendshipRepository*: define as transações relacionadas à entidade Amizade.

4.3.3 Microsserviço *sync-ms-chat*

Este microsserviço gerencia o conteúdo privativo, permitindo criação de novas conversas e leitura e envio de novas mensagens entre usuários amigos. Quando necessário o microsserviço verifica a existência de uma determinada amizade através de uma requisição HTTP para o microsserviço *sync-ms-connection*.

Esse módulo também conta com um banco de dados próprio chamado *db-chat*. Tal banco possui três tabelas:

- *chats*: representa um chat em si;
- *chats_users*: representa relações entre chat e usuários, ou seja, quais usuários pertencem a um determinado chat;
- *messages*: representa uma mensagem enviada em um chat.

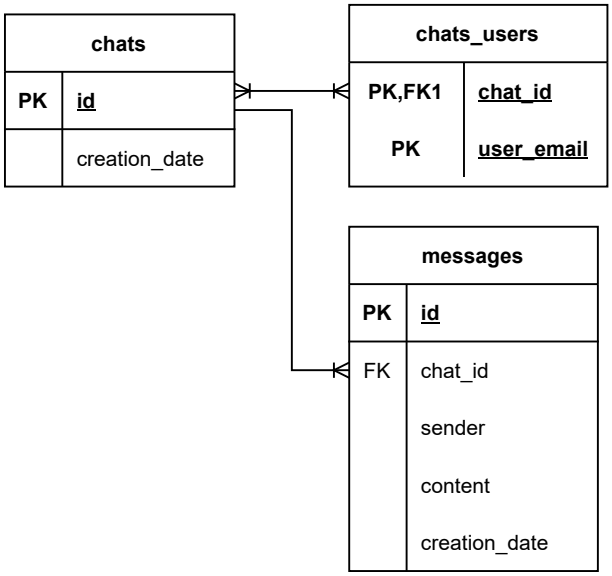

Figura 4.13: *Diagrama de entidades no modelo UML do banco de dados db-chat na solução síncrona*

O microsserviço expõe uma API REST através do *ChatController*. Por meio dela podemos realizar operações para enviar mensagens entre usuários, obter as informações de um determinado chat, etc.

Diferentemente da solução assíncrona, ao tentar enviar uma mensagem de um usuário para outro, este microsserviço precisa consultar o *sync-ms-connection* para verificar se os usuários em questão são amigos. Isso torna seu funcionamento dependente do microsserviço de conexões.

As demais classes do microsserviço foram desenvolvidas de forma a terem comportamentos bem definidos e encapsulados. Para entender o funcionamento do microsserviço de forma mais geral pode-se visualizar o UML na figura 4.14.

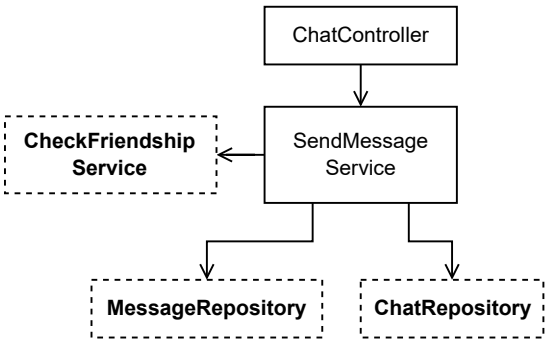

Figura 4.14: *UML das principais classes do microsserviço sync-ms-chat*

- *ChatController*: responsável por expor a API pública do microsserviço;
- *SendMessageService*: modela o comportamento do envio de mensagem entre dois usuários;
- *ChatRepository*: define as transações relacionadas à entidade Chat;

- *MessageRepository*: define as transações relacionadas à entidade Mensagem;
- *CheckFriendshipService*: define as operações para comunicação com *sync-ms-connections* para verificar existência de amizade.

4.3.4 Diagrama de sequência

Nessa arquitetura síncrona, podemos representar a sequência de processos para criação de amizade e envio de mensagens entre usuários através do diagrama na figura 4.15.

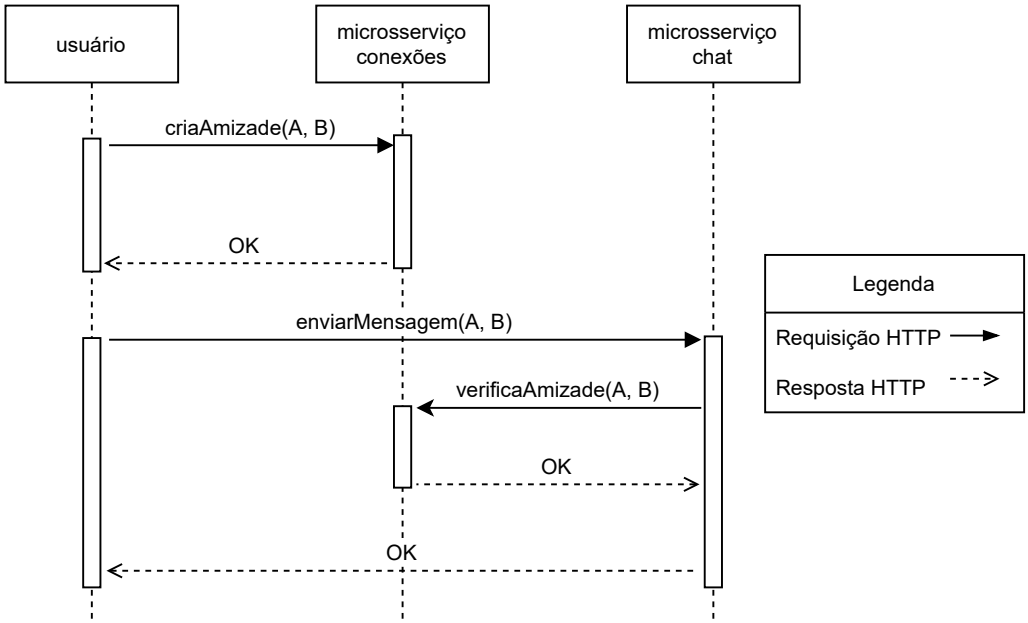

Figura 4.15: *Diagrama de sequência com funcionamento da arquitetura síncrona*

Note que para estabelecimento de amizade temos os seguintes processos:

1. O usuário envia uma requisição para *sync-ms-connection*;
2. O microsserviço de conexões armazena essa nova relação em seu banco de dados.

Note que para o envio de mensagens entre usuários temos os seguintes processos:

1. O usuário enviar uma requisição para *sync-ms-chat*;
2. O microsserviço *sync-ms-chat* faz uma requisição ao *sync-ms-connection* para verificar a existência da amizade;
3. O microsserviço de conexões verifica em seu banco de dados a existência da amizade e devolve uma resposta de acordo com o resultado;
4. O microsserviço de chat armazena, ou não, a mensagem de texto com base na resposta de *sync-ms-connection*.

É possível notar que a operação de estabelecimento de amizade é mais simples do que na arquitetura assíncrona. Porém, essa simplicidade cria uma relação de dependência

forte entre os microsserviços *sync-ms-connection* e *sync-ms-chat* quando comparado com a outra arquitetura. Note também que, por conta dessa dependência, a operação de envio de mensagem torna-se mais complexa, com um total de 4 processos quando comparado com a solução assíncrona, onde temos apenas 2 processos.

4.4 Métricas de código

Para estabelecer um visão geral sobre o código desenvolvido em ambes as soluções utilizamos a ferramenta cloc[5]. Por meio dela podemos obter métricas da quantidade de linhas de código do projeto, linhas de comentário, arquivos, etc.

As métricas para *async-ms-chat* podem ser vistas na tabela 4.1.

Language	files	blank	comment	code
Kotlin	32	207	0	761
XML	6	0	0	186
Bourne Shell	1	28	109	103
DOS Batch	1	21	2	68
Gradle	2	7	0	47
YAML	2	3	0	44
Dockerfile	1	25	26	23
Properties	2	0	1	6
SQL	1	0	0	2
SUM:	48	291	138	1240

Tabela 4.1: *Métricas de código para o microsserviço async-ms-chat*

[5] https://github.com/AlDanial/cloc

As métricas para *async-ms-connection* podem ser vistas na tabela 4.2.

Language	files	blank	comment	code
Kotlin	20	103	0	398
XML	7	0	0	283
Bourne Shell	1	28	109	103
DOS Batch	1	21	2	68
Gradle	2	7	0	47
YAML	2	3	0	43
Dockerfile	1	25	26	23
Properties	2	0	1	6
SQL	1	0	0	2
SUM:	37	187	138	973

Tabela 4.2: *Métricas de código para o microsserviço async-ms-connection*

As métricas para *sync-ms-chat* podem ser vistas na tabela 4.3.

Language	files	blank	comment	code
Kotlin	26	173	0	650
XML	6	0	0	202
Bourne Shell	1	28	109	103
DOS Batch	1	21	2	68
Gradle	2	7	0	46
YAML	3	4	0	46
Dockerfile	1	25	26	23
Properties	2	0	1	6
SQL	1	0	0	2
SUM:	43	258	138	1146

Tabela 4.3: *Métricas de código para o microsserviço sync-ms-chat*

As métricas para *sync-ms-connection* podem ser vistas na tabela 4.4.

Language	files	blank	comment	code
Kotlin	15	85	0	332
XML	6	0	0	233
Bourne Shell	1	28	109	103
DOS Batch	1	21	2	68
Gradle	2	7	0	46
YAML	2	2	0	29
Dockerfile	1	25	26	23
Properties	2	0	1	6
SQL	1	0	0	2
SUM:	31	168	138	842

Tabela 4.4: *Métricas de código para o microsserviço sync-ms-connection*

Comparando as métricas podemos notar que o serviço de chat possui um pouco mais de código na arquitetura assíncrona do que na síncrona. Isso acontece por conta das soluções implementadas. Como explicado nas seções anteriores, na solução assíncrona o serviço de chat já possui as informações sobre as amizades criadas em seu próprio banco de dados. Por conta disso, é necessário implementar classes de entidades, repositórios e serviços para que ele consiga manipular esses dados.

Também conseguimos notar que o serviço de conexões possui um pouco mais de código na solução assíncrona do que na síncrona. Isso acontece pois na primeira, o serviço precisa de algumas classes de serviço e entidades a mais para publicar mensagens na fila do RabbitMQ.

Capítulo 5

Análise de *trade-offs*

Conforme definido na seção 3.1, o objetivo desta análise é identificar a influência que os padrões de comunicação síncrono e assíncrono exercem nos atributos tamanho do módulo/serviço, compartilhamento de banco de dados entre módulos/serviços, nível de acoplamento do sistema, desempenho das operações e heterogeneidade de tecnologias disponíveis para o desenvolvimento. Este capítulo apresenta as análises de *trade-offs* sob cada um desses atributos.

5.1 Tamanho do serviço

O atributo tamanho do serviço engloba aspectos relacionados ao tamanho dos serviços e módulos. Utilizando as diretrizes do CharM as métricas adotadas nesse dimensão são: número de serviços por módulo e número de operações por serviço.

No escopo do trabalho o objetivo é analisar se cada padrão de comunicação (síncrono ou assíncrono) utilizado na solução do sistema tem influência no aumento ou diminuição dos serviços e tamanho dos módulos.

Levando-se em consideração o módulo de chat, ambas as soluções apresentam apenas 1 serviço por módulo. Sendo que cada microsserviço realiza apenas 3 operações, sendo elas: criação de novas conversas, envio de novas mensagens em uma conversa, leitura de mensagens de uma conversa. As métricas do CharM podem ser sumarizadas na tabela 5.1.

Serviço	Número de serviços por módulo	Número de operações
async-ms-chat	1	3
sync-ms-chat	1	3

Tabela 5.1: *Métricas do CharM para atributo tamanho do serviço no módulo de chat*

Levando-se em consideração o módulo de conexão/amizade, ambas as soluções apresentam apenas 1 serviço por módulo. Sendo que cada microsserviço realiza apenas 2 operações,

Serviço	Número de serviços por módulo	Número de operações
async-ms-connection	1	2
sync-ms-connection	1	2

Tabela 5.2: *Métricas do CharM para atributo tamanho do serviço no módulo de conexões*

sendo elas: criação de novas amizades e verificação da existência de uma amizade. As métricas do CharM podem ser sumarizadas na tabela 5.2.

Levando em consideração as métricas obtidas por serviço, podemos apresentar as métricas gerais do sistema através da tabela 5.3.

Tipo do sistema	Número de módulos	Número de serviços	Serviços por módulo	Total de operações
Assíncrono	2	2	1	5
Síncrono	2	2	1	5

Tabela 5.3: *Métricas do CharM para atributo tamanho do serviço no sistema Pingr*

Analisando a influência gerada por cada padrão nas soluções podemos concluir que não há *trade-offs* gerados pela escolha de um padrão de comunicação ao invés do outro. O atributo tamanho do serviço mantém-se neutro quando comparamos as soluções.

5.2 Compartilhamento de bases de dados entre módulos

Esta dimensão caracteriza a distribuição de base de dados entre os módulos, apontando se uma base de dados é utilizada por um ou mais módulos. O objetivo é entender se a escolha da estratégia de comunicação afeta o compartilhamento das bases de dados do sistema. As métricas adotadas nesta dimensão são: número de bases de dados por módulo e número de módulos que compartilham uma mesma base de dados.

Levando-se em consideração o módulo de chat, ambas as soluções apresentam apenas 1 base de dados por módulo. Sendo que essa base de dados é exclusiva do serviço de chat. As métricas do CharM podem ser sumarizadas na tabela 5.4.

Serviço	Bases de dados exclusivas	Bases de dados compartilhados entre serviços	Bases com operações apenas de escrita	Bases com operações apenas de leitura	Bases com operações de escrita e leitura
async-ms-chat	1	0	0	0	1
sync-ms-chat	1	0	0	0	1

Tabela 5.4: *Métricas do CharM para atributo compartilhamento de bases de dados no módulo de chat*

Levando-se em consideração o módulo de conexão/amizade, ambas as soluções apresentam apenas 1 base de dados por módulo. Sendo que essa base de dados é exclusiva do

serviço de conexão. As métricas do CharM podem ser sumarizadas na tabela 5.5.

Serviço	Bases de dados exclusivas	Bases de dados compartilhados entre serviços	Bases com operações apenas de escrita	Bases com operações apenas de leitura	Bases com operações de escrita e leitura
async-ms-connection	1	0	0	0	1
sync-ms-connection	1	0	0	0	1

Tabela 5.5: *Métricas do CharM para atributo compartilhamento de bases de dados no módulo de conexões*

Levando em consideração as métricas obtidas por serviço, podemos apresentar as métricas gerais do sistema através da tabela 5.6.

Tipo do sistema	Total de bases de dados	Bases de dados exclusivas	Bases de dados compartilhadas
Assíncrono	2	2	0
Síncrono	2	2	0

Tabela 5.6: *Métricas do CharM para atributo compartilhamento de bases de dados no sistema Pingr*

Analisando a influência gerada por cada padrão nas soluções podemos concluir que não há *trade-offs* gerados pela escolha de uma forma de comunicação sobre a outra. O atributo compartilhamento de bases de dados entre módulos mantém-se neutro quando comparamos as soluções.

5.3 Nível de acoplamento

Esta dimensão caracteriza o grau de dependência/número de conexões entre os serviços. O objetivo é analisar se cada padrão de comunicação influencia no aumento ou diminuição do nível de acoplamento entre serviços.

Um sistema real pode conter serviços que comunicam-se de forma síncrona, enquanto outros comunicam-se de forma assíncrona. Por conta disso, o CharM separa métricas de acoplamento em dois tipos, síncronas ou assíncronas. Dessa forma, o arquiteto consegue identificar o nível de acoplamento síncrono e o nível de acoplamento assíncrono de um mesmo sistema.

Como esse trabalho implementa duas soluções, uma puramente síncrona e outra puramente assíncrona, simplificaremos as métricas coletadas por solução.

Para a solução síncrona, esta dimensão caracteriza o grau de dependência síncrona entre os serviços do sistema. Em uma interação síncrona, um serviço x faz uma solicitação a um serviço y e fica bloqueado, aguardando uma resposta em tempo hábil (Richardson, 2018). As métricas adotadas nesta solução são a AIS (*Absolute Importance of the Service*, ou grau de importância do serviço), que indica o número de serviços clientes que invocam as

operações do serviço, e ADS (*Absolute Dependence of the Service*, ou grau de dependência do serviço), que indica o número de serviços que o dado serviço é dependente.

De forma parecida, para a solução assíncrona, esta dimensão caracteriza o grau de dependência assíncrona entre os serviços do sistema. Diferente da síncrona, na interação assíncrona o serviço x que enviou uma solicitação a um serviço y, não fica bloqueado aguardando a resposta, que pode demorar ou não chegar (Pahl e Jamshidi, 2016). As métricas coletadas também são a AIS, que indica o número de diferentes tipos de mensagens publicadas pelo serviço, e ADS, que indica o número de diferentes tipos de mensagens consumidas pelo serviço.

Levando-se em consideração o módulo de chat, podemos notar que as métricas do CharM mantêm-se as mesmas quando comparamos as duas soluções. Elas podem ser sumarizadas na tabela 5.7.

Serviço	Grau de importância	Grau de dependência
async-ms-chat	0	1
sync-ms-chat	0	1

Tabela 5.7: *Métricas do CharM para atributo nível de acoplamento no módulo de chat*

Levando-se em consideração o módulo de conexão/amizade, podemos notar que as métricas do CharM também mantêm-se as mesmas quando comparamos as duas soluções. Elas podem ser sumarizadas na tabela 5.8.

Serviço	Grau de importância	Grau de dependência
async-ms-connection	1	0
sync-ms-connection	1	0

Tabela 5.8: *Métricas do CharM para atributo nível de acoplamento no módulo de conexões*

Em ambas as soluções desenvolvidas, o serviço de chat necessita de informações provenientes do serviço de conexão. A diferença entre elas é a forma como essas informações são disponibilizadas. Na solução síncrona, o serviço de conexões precisa ser diretamente requisitado, enquanto que na solução assíncrona, não existe dependência direta entre os serviços.

Analisando a influência gerada por cada padrão nas soluções podemos concluir que ambos os sistemas desenvolvidos possuem certo nível de acoplamento. A diferença é se esse acoplamento é direto, impactando o funcionamento do sistema, ou não.

Na solução síncrona temos acoplamento síncrono no sistema. Isso significa que os serviços passam a executar suas funções com tempo de espera ou bloqueio por parte de outro. Em particular, o serviço *sync-ms-chat* é bloqueado por *sync-ms-connection* sempre que precisa enviar uma nova mensagem de um usuário a outro. Isso também gera um sistema pouco resiliente, visto que o serviço de chat não pode executar a função de envio de mensagem quando sync-ms-connection não estiver disponível. Demonstrando, dessa forma, que o acoplamento entre eles é muito forte.

Na solução assíncrona temos acoplamento assíncrono no sistema, sendo que *async-ms-chat* depende das mensagens publicadas por *async-ms-connection*. É importante ressaltar que essa dependência não ocorre diretamente entre os serviços, mas sim da garantia de que as mensagens sejam publicadas e estejam disponíveis na fila/tópico para os serviços interessados. Isso significa que existe dependência com o *broker* de mensagens (que deve ser bastante resiliente). Também significa que ambos os serviços podem executar de forma resiliente independente do estado do outro (se um cair, o outro continua executando sua função plenamente).

Em resumo, na solução síncrona temos um grau de dependência forte entre os serviços de chat e conexão, sendo que o chat fica bloqueado ou pode parar de funcionar completamente caso o serviço de conexões esteja lento ou indisponível. Enquanto que na solução assíncrona não temos dependência direta entre os serviços, onde ambos podem funcionar plenamente independente um do outro.

Em termos de *trade-offs*, optar pela solução assíncrona diminui o nível de acoplamento direto do sistema e também torna-o mais resiliente e mais tolerante a falhas.

5.4 Desempenho

Esta dimensão tem como objetivo metrificar a latência de resposta dos serviços. Identificando o tempo que cada solução leva para completar o mesmo conjunto de operações.

5.4.1 Lógica de negócio

Dado os serviços implementados e a lógica de negócio por trás de seu funcionamento, escolhemos a operação de envio de mensagens entre usuários para ser analisada nos testes de performance.

A operação de envio de mensagem possui uma diferença significativa em seu funcionamento quando comparamos ambas as soluções implementadas. É importante ressaltar que o que proporcionou essa diferenciação na operação foi a forma de integração entre microsserviços utilizada pela arquitetura da solução. Por conta disso, essa funcionalidade é ideal para uma análise de *trade-offs*.

Como explicado nos capítulos anteriores, quando um usuário inicia uma conversa com outro, para os serviços com comunicação síncrona, o módulo de chat precisa consultar o módulo de conexões para verificar a existência de amizade entre eles antes de permitir o envio de mensagens. Enquanto isso, na solução assíncrona, o módulo de chat já possui as informações necessárias para a validação da amizade e envio de mensagens.

5.4.2 Cenários de teste

Os critérios estabelecidos para os teste foram:

- Os usuários envolvidos no envio de mensagens já possuem amizade estabelecida.

Os cenários de teste em si foram:

- Os serviços implementados foram executados na mesma máquina virtual;
- Os serviços implementados foram executados em máquinas virtuais distintas.

Assim, temos um total de 4 cenários de teste, sendo 2 para a solução síncrona e outros 2 para a solução assíncrona.

5.4.3 Métricas em observação

Para os testes de desempenho escolheu-se observar a métrica de **latência de resposta da operação**. Para isso, a medição foi feita pelo intervalo de tempo que o serviço de chat leva para processar uma requisição de envio de mensagem. Para facilitar a análise desta dimensão foram criados alguns scripts em Python. Seus funcionamentos estão descrito no apêndice A.

5.4.4 Resultados

As estatísticas resultantes podem ser observadas na tabela 5.9:

Experimento	Total de medições	Média das latências (segundos)	Desvio padrão das latências	Intervalo de confiança
async-diff-vm	50	0.018850	0.001411	[0.018459, 0.019241]
async-single-vm	50	0.019238	0.000823	[0.019009, 0.019466]
sync-diff-vm	50	0.024048	0.000930	[0.023791, 0.024306]
sync-single-vm	50	0.023098	0.001304	[0.022736, 0.023459]

Tabela 5.9: *Estatísticas de latência de resposta dos módulos durante os testes de performance da operação de envio de mensagem entre usuários*

O gráfico na figura 5.1 apresenta as latências obtidas por experimento em cada cenário de teste.

Figura 5.1: *Gráfico de latência de resposta por experimento*

O box plot na figura 5.2 apresenta variação dos dados de cada experimento.

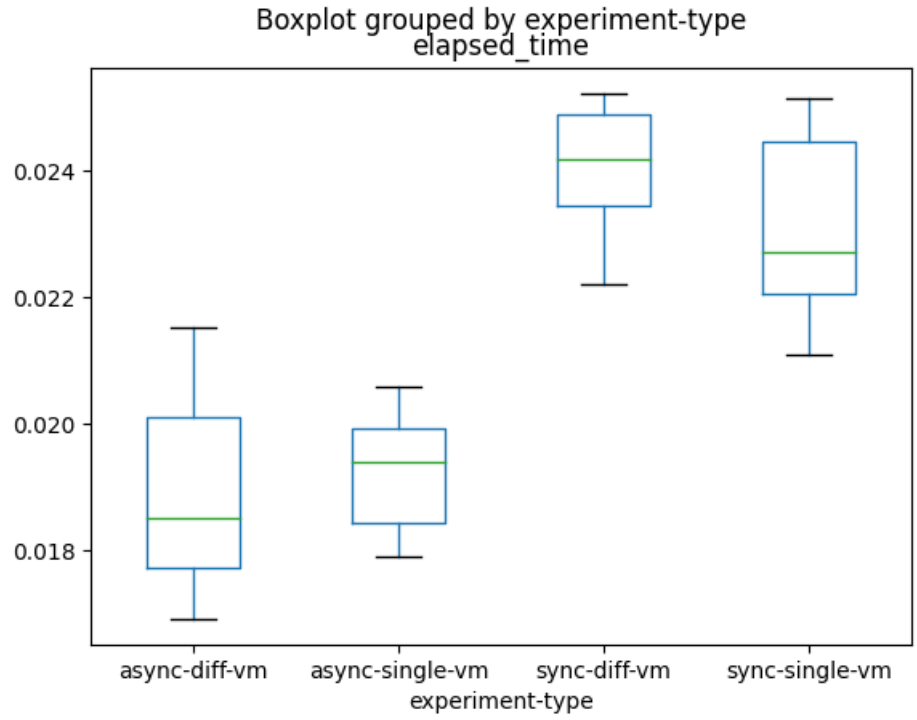

Figura 5.2: *Gráfico Box Plot para latência de respostas por experimento*

Durante os testes observou-se que em média as 20 primeiras requisições/mensagens enviadas pelos microsserviços tinham uma latência ligeiramente maior. Após isso, os tempos tornavam-se menores e mais estáveis, como evidenciado nos gráficos anteriores. Este trabalho não se aprofundou na razão dessa necessidade de *warm-up* dos microsserviços.

É importante ressaltar que a análise de *trade-offs* do atributo desempenho está fortemente atrelada a estratégia utilizada para a operação sendo analisada. Tal estratégia foi possibilitada pela forma de comunicação de cada solução desenvolvida. As soluções implementadas foram as mais simples e naturais vislumbradas por cada forma de integração utilizada.

Através dos resultados obtidos nota-se que a solução assíncrona é ligeiramente mais rápida do que a solução síncrona. Isso já era esperado devido ao fato de que, nesta solução, o serviço de chat não precisar consultar o módulo de conexões para operar o envio de mensagens.

Em termos de *trade-offs*, podemos dizer que, quando bem utilizada, a comunicação por mensageria possibilita o desenvolvimento de um sistema mais rápido e eficiente. Enquanto que a solução síncrona, de forma natural, apresenta um funcionamento mais lento e menos eficiente.

5.5 Heterogeneidade de tecnologias disponíveis para cada solução

Esta dimensão tem por objetivo verificar a quantidade de linguagens de programação que a documentação das técnicas/ferramentas de comunicação indicam que permitem integração. Para isso, foi realizado um levantamento de bibliotecas existentes que oferecem meios de se comunicar em HTTP ou AMQP para algumas linguagens de programação utilizadas no mercado.

Para as linguagens escolhidas no levantamento foi utilizada a pesquisa *"Top Programming Languages 2022"* da IEEE Spectrum (Spectrum, 2022). Esse levantamento inclui algumas linguagens como HTML, Verilog, VHDL, etc. que não representam realmente linguagens de programação ou não estariam no escopo desse trabalho, mas estas foram mantidas na análise para permanecer em acordo com a pesquisa.

Como estamos focando no escopo de desenvolvimento de microsserviços, também utilizou-se as pesquisas *"DevEcosystem 2021"* da Jetbrains (Jetbrains, 2021) e *"Primary programming languages among microservices developers worldwide in 2021"* da Statista (Statista, 2021), para garantir que o levantamento cobrisse as linguagens mais utilizadas durante o desenvolvimento desse tipo de arquitetura. As linguagens que aparecem no topo da tabela 5.10, com background azul, são as presentes nas duas últimas pesquisas, as demais vieram do levantamento da IEEE.

A tabela 5.10 lista as linguagens utilizadas no levantamento e as bibliotecas existentes em cada uma delas com suporte a uma determinada forma de comunicação. Dentre as 54 linguagens listadas, 44 possuem bibliotecas com suporte a AMQP, enquanto que 49

possuem suporte a HTTP.

Linguagem	AMQP	HTTP
Java	*RabbitMQ Java Client* s.d.	*Apache HttpComponents* s.d.
Javascript	*amqp-node* s.d.	*Axios* s.d.
Python	*pika - PyPI* s.d.	*Requests* s.d.
SQL	*RabbitMQ stored procedures* s.d.	*SQL httprequest* s.d.
PHP	*php-amqplib* s.d.	*Yii 2 HTTP client* s.d.
Typescript	*amqp-client.js* s.d.	*Axios* s.d.
HTML	✗	✗
C#	*.NET/CSharp RabbitMQ Client* s.d.	*HTTP .NET Client* s.d.
Go	*AMQP Go client* s.d.	*Go REST Client* s.d.
Kotlin	*RabbitMQ Java Client* s.d.	*Apache HttpComponents* s.d.
Shell	*amqpcat* s.d.	*curl* s.d.
C++	*RabbitMQ C++ client* s.d.	*LibHTTP* s.d.
C	*RabbitMQ C client* s.d.	*LibHTTP* s.d.
R	*messageQueue-package* s.d.	*CRAN - Package crul* s.d.
Ruby	*RabbitMQ Ruby client* s.d.	*Faraday* s.d.
Scala	*Scala AMQP client* s.d.	*Scala HTTP client* s.d.
Matlab	*MATLAB RabbitMQ Interface* s.d.	*MATLAB HTTP interface* s.d.
SAS	*SAS AMQP Package* s.d.	*SAS HTTP client* s.d.
Assembly	✗	✗
Rust	*AMQP Rust Client* s.d.	*Rust HTTP client* s.d.
Perl	*Perl AMQP client* s.d.	*HTTP::Tiny* s.d.
Objective-C	*RabbitMQ for Objective-C/Swift* s.d.	*HTTP interface* s.d.
Dart	*dart-amqp* s.d.	*Dart HTTP Package* s.d.
Swift	*RabbitMQ for Objective-C/Swift* s.d.	*Swift HTTP Client* s.d.
Julia	*AMQPClient.jl* s.d.	*JuliaWeb/HTTP.jl* s.d.
Visual Basic	*.NET/CSharp RabbitMQ Client* s.d.	*libcurl.vb* s.d.
Groovy on Grails	*RabbitMQ Grails Plugin* s.d.	*httpbuilder* s.d.
Lua	*Lua RabbitMQ client* s.d.	*Lua HTTP Client* s.d.
Haskell	*Haskell AMQP client* s.d.	*Haskell HTTP Client* s.d.
COBOL	*RabbitMQ from COBOL* s.d.	*HTTP from COBOL* s.d.
Elixir	*RabbitMQ Elixir client* s.d.	*elixir-mint* s.d.
F#	*F-RabbitMQ* s.d.	*FSharp HTTP library* s.d.
Lisp	*RabbitMQ interface for Lisp* s.d.	*Lisp HTTP library* s.d.
Delphi/Pascal	*habari-rabbitmq* s.d.	*RADStudio HTTP library* s.d.
Clojure	*Clojure AMQP client* s.d.	*Clojure HTTP library* s.d.
Prolog	✗	*Prolog HTTP Package* s.d.
OCaml	*OCaml Amqp client* s.d.	*OCaml HTTP Lib* s.d.
Erlang	*Erlang RabbitMQ Client* s.d.	*Erlang HTTP Client* s.d.
JRuby	*RabbitMQ JRuby Client* s.d.	*jruby-httpclient* s.d.
Crystal	*RabbitMQ Crystal Client* s.d.	*Crystal Http Client* s.d.
Elm	✗	*ELM Http Package* s.d.
Raku	*P6-Net-AMQP* s.d.	*HTTP-Easy* s.d.
Arduino	*Arduino RabbitMQ Lib* s.d.	*Arduino HTTP Client* s.d.

LabView	*LabbitMQ Toolkit* s.d.	*HTTP for Labview* s.d.
Verilog	✗	✗
VHDL	✗	✗
DLang	*rabbitmq-d* s.d.	*Struct HTTP* s.d.
Cuda	✗	✗
Ada	*AdaBinding* s.d.	*Utility Library* s.d.
Scheme	✗	*Gerbil HTTP* s.d.
ABAP	✗	*HTTP for SAP* s.d.
WebAssembly	✗	*wasi-experimental-http* s.d.
CoffeScript	*amqp-coffee* s.d.	*basic-http-client* s.d.
Unity3D	*AMQP Unity 3D client* s.d.	*HTTP client for Unity3D* s.d.

Tabela 5.10: *Linguages de programação e suas bibliotecas com suporte aos modelos de comunicação assíncrono através de AMQP e síncrono através de HTTP*

Pode-se notar que dentre as linguagens mais utilizadas para o desenvolvimento de microsserviços, todas, com exceção de HTML, possuem alguma biblioteca que oferece suporte para comunicação via AMQP ou HTTP.

Levando em consideração o amplo espectro de linguagens, temos que 10 linguagens não possuem bibliotecas com suporte de comunicação via AMQP e 5 não possuem bibliotecas com suporte de comunicação via HTTP. Dentre as 10 sem suporte a AMQP, temos presente as 5 sem suporte a HTTP. As outras 5 restantes possuem suporte a HTTP.

Assim, em questão de *trade-offs*:

- Se considerarmos as linguagens mais utilizadas para desenvolvimento de microsserviços, ambas as soluções, síncrona ou assíncrona, seriam equivalentes. Ou seja, atuam de forma neutra no atributo "heterogeneidade de tecnologias disponíveis";
- Se considerarmos o amplo espectro das linguagens mais utilizadas no mercado, nos anos de 2021 e 2022, então a escolha de uma solução assíncrona pode comprometer de forma negativa as tecnologias que podem ser empregadas na arquitetura do sistema.

Capítulo 6

Conclusão

Neste trabalho, buscamos elencar argumentos suportados por resultados práticos que sirvam de auxílio a engenheiros e arquitetos de software na hora de escolher o modelo de comunicação em aplicações de microsserviços. Para isso, foi implementado um recorte do sistema Pingr para mostrar a influência (negativa, neutra ou positiva) exercida por um padrão de comunicação (síncrono ou assíncrono) entre microsserviços nos seguintes atributos de software: tamanho dos serviços/módulos desenvolvidos, compartilhamento de bases de dados entre módulos, nível de acoplamento entre serviços, performance das operações, heterogeneidade de tecnologias disponíveis para aplicação da solução.

Após aplicar o método proposto por AL, 2020b em conjunto com o modelo de caracterização CharM descrito em AL, 2020a e analisar os resultados gerados, acreditamos que obtivemos *insights* interessantes. Assim, levando em conta os *trade-offs* observados no capítulo anterior, podemos sumarizar as análises de cada atributo na tabela 6.1. A coluna "Resultado final" representa uma visão ampla sobre as dimensões analisadas.

Padrão de comunicação	Tam. do serviço	Comp. de bases de dados	Nível de acoplamento	Desempenho	Het. de tec.	Resultado final
Assíncrono	Neutro	Neutro	Positivo	Positivo	Neutro	Positivo
Síncrono	Neutro	Neutro	Negativo	Negativo	Neutro	Negativo

Tabela 6.1: *Influência exercida pelos padrões de comunicação nos atributos de software*

No geral, podemos notar que o padrão de comunicação assíncrono mostrou-se melhor do que o padrão síncrono ao implementarmos o sistema proposto seguindo uma arquitetura de microsserviços.

Acreditamos que esse resultado possa ser identificado nos mais diversos sistemas com MSA, sempre que a solução proposta esteja adequada com o modelo de comunicação utilizado na arquitetura. Também acreditamos que os resultados aqui apresentados possam ajudar a guiar arquitetos e engenheiros de software na tomada de decisão em seus projetos.

Como continuidade deste trabalho, seria interessante comparar outras alternativas de integração entre microsserviços, como gRPC, SOAP, etc. Além disso, pode-se adicionar

novos atributos qualitativos para a análise de *trade-offs*, como por exemplo, segurança do sistema, agilidade de desenvolvimento, etc. Também seria interessante explorar o comportamento de aplicações que transmitem mensagens/requisições maiores durante a comunicação entre serviços. Dessa forma, poderia ser analisada como o tamanho da mensagem interfere na latência da comunicação entre módulos.

Outro aspecto que pode ser explorado é a identificação da razão da necessidade de *warm-up* pelos microsserviços desenvolvidos, como indicado em 5.4. Durante os testes de performance das aplicações observou-se que as primeiras requisições/mensagens enviadas pelos microsserviços tinham uma latência maior do que as demais. Conforme os microsserviços enviavam novas mensagens as latências de resposta tornavam-se menores e mais estáveis. Este trabalho não se aprofundou no fato ocorrido, porém reproduzir e analisar este comportamento pode gerar discussões interessantes em um trabalho futuro.

Apêndice A

Scripts para teste de desempenho

Como explicado na seção 5.4, escolheu-se observar a métrica de **latência de resposta** da operação de envio de mensagem entre usuários para comparar os *trade-offs* gerados pela escolha de um padrão de comunicação. A medição foi feita pelo intervalo de tempo que o serviço de chat leva para processar uma requisição de envio de mensagem. Para facilitar a análise desta dimensão foram criados alguns scripts em Python. Estes podem ser acessados em https://github.com/LucasMorettoSilva/pingr-perf.

A.1 Script *perf.py*

O script *perf.py* foi criado para automatizar o processo de testes. Ele é responsável por realizar a requisição ao módulo de chat e metrificar a latência da operação. O script executa um total de 70 experimentos por cenário e gera um arquivo com as latências observadas por experimento.

Cada experimento consiste em enviar uma requisição `POST` para o *endpoint* `/api/chats/messages` no serviço de chat, com o seguinte corpo em `JSON`:

```
{
    "senderEmail": "user2@email.com",
    "recipientEmail": "user1@email.com",
    "message": "hey, how are you?"
}
```

Dependendo do padrão de arquitetura da solução que está sob teste, o sistema executará uma sequência de operações distintas. Quando essa for finalizada, o script armazenará o tempo decorrido da operação em um arquivo *.csv*, onde a primeira coluna da uma linha representa o número do experimento e a segunda a latência de resposta em segundos.

A.2 Script *merge.py*

Ao final dos testes de desempenho, os resultados obtidos pelo script *perf.py* estarão em quatro arquivos distintos, um para cada cenário de teste.

O script *merge.py* é responsável por juntar os dados de cada arquivo de resultado num único arquivo, agrupando-os por um identificador comum. Os resultados escritos neste arquivo final são diferenciados pela primeira coluna da linha, que representa o cenário de teste daquele resultado. Estes podem ser "*async-diff-vm*", "*async-single-vm*", "*sync-diff-vm*"e "*sync-single-vm*".

A.3 Script *plot.py*

Este script simplesmente coleta as informações do arquivo único produzido por *merge.py* e gera as estatísticas de média, desvio padrão e intervalo de confiança (95%), exibindo os dados de forma gráfica para uma comparação visual.

Referências

[*.NET/CSharp RabbitMQ Client* s.d.] *.NET/CSharp RabbitMQ Client Library - RabbitMQ*. URL: https://www.rabbitmq.com/dotnet.html (citado na pg. 37).

[Aksakalli *et al.* 2021] Karabey Aksakalli, Turgay Celik, Ahmet Burak Can e Bedir Tekinerdogan. "Deployment and communication patterns in microservice architectures: a systematic literature review". Em: *Journal of Systems and Software* 180 (2021), pg. 111014 (citado na pg. 12).

[*Apache HttpComponents* s.d.] *Apache HttpComponents – HttpClient Overview*. URL: https://hc.apache.org/httpcomponents-client-4.5.x/index.html (citado na pg. 37).

[*RabbitMQ from COBOL* s.d.] *Assorted Ramblings: Using RabbitMQ from COBOL*. URL: https://assortedrambles.blogspot.com/2013/04/using-rabbitmq-from-cobol_9584.html (citado na pg. 37).

[Bhojwani 2018] Rajesh Bhojwani. *Design Patterns for Microservice-To-Microservice Communication*. 2018. URL: https://dzone.com/articles/design-patterns-for-microservice-communication (acesso em 11/11/2022) (citado na pg. 12).

[Bonér 2016] Jonas Bonér. "Reactive microservices architecture: design principles for distributed systems". Em: *O'Reilly Media* (2016) (citado na pg. 12).

[*RabbitMQ Ruby client* s.d.] *Bunny, a dead easy to use RabbitMQ Ruby client*. URL: http://rubybunny.info/ (citado na pg. 37).

[*basic-http-client* s.d.] *CoffeeScript Cookbook » Basic HTTP Client*. URL: https://coffeescript-cookbook.github.io/chapters/networking/basic-http-client (citado na pg. 38).

[*CRAN - Package crul* s.d.] *CRAN - Package crul*. URL: https://cran.r-project.org/web/packages/crul/ (citado na pg. 37).

[*curl* s.d.] *curl*. URL: https://curl.se/ (citado na pg. 37).

[*dart-amqp* s.d.] *dart-amqp - Dart Package*. URL: https://pub.dev/packages/dart_amqp (citado na pg. 37).

[*Erlang HTTP Client* s.d.] *Erlang HTTP Client*. URL: https://www.erlang.org/doc/apps/inets/http_client.html (citado na pg. 37).

[*Erlang RabbitMQ Client* s.d.] *Erlang RabbitMQ Client library - RabbitMQ*. URL: https://www.rabbitmq.com/erlang-client-user-guide.html (citado na pg. 37).

[Fowler e Lewis 2014] Martin Fowler e James Lewis. *Microservices*. 2014. URL: https://martinfowler.com/articles/microservices.html (acesso em 18/09/2022) (citado nas pgs. 1, 12).

[*RabbitMQ C++ client* s.d.] *GitHub - akalend/amqpcpp: rabbitcpp is a C++ library for Message Queue Server RabbitMQ*. URL: https://github.com/akalend/amqpcpp (citado na pg. 37).

[*RabbitMQ C client* s.d.] *GitHub - alanxz/rabbitmq-c: RabbitMQ C client*. URL: https://github.com/alanxz/rabbitmq-c (citado na pg. 37).

[*amqp-node* s.d.] *GitHub - amqp-node/amqplib: AMQP 0-9-1 library and client for Node.JS*. URL: https://github.com/amqp-node/amqplib (citado na pg. 37).

[*AMQP Rust Client* s.d.] *GitHub - amqp-rs/lapin: AMQP client library in Rust, with a clean, futures based API*. URL: https://github.com/sozu-proxy/lapin (citado na pg. 37).

[*OCaml Amqp client* s.d.] *GitHub - andersfugmann/amqp-client: OCaml Amqp client library for Async and Lwt*. URL: https://github.com/andersfugmann/amqp-client (citado na pg. 37).

[*Axios* s.d.] *GitHub - axios/axios: Promise based HTTP client for the browser and node.js*. URL: https://github.com/axios/axios (citado na pg. 37).

[*RabbitMQ Crystal Client* s.d.] *GitHub - cloudamqp/amqp-client.cr: An AMQP 0-9-1 client for Crystal*. URL: https://github.com/cloudamqp/amqp-client.cr (citado na pg. 37).

[*amqp-client.js* s.d.] *GitHub - cloudamqp/amqp-client.js: AMQP 0-9-1 TypeScript client both for Node.js and browsers (using WebSocket)*. URL: https://github.com/cloudamqp/amqp-client.js (citado na pg. 37).

[*amqpcat* s.d.] *GitHub - cloudamqp/amqpcat: CLI tool for publishing to and consuming from AMQP servers*. URL: https://github.com/cloudamqp/amqpcat (citado na pg. 37).

[*Perl AMQP client* s.d.] *GitHub - cooldaemon/RabbitFoot: An asynchronous and multi channel Perl AMQP client*. URL: https://github.com/cooldaemon/RabbitFoot (citado na pg. 37).

[*AMQP Unity 3D client* s.d.] *GitHub - CymaticLabs/Unity3D.Amqp: AMQP client library for Unity 3D supporting RabbitMQ*. URL: https://github.com/CymaticLabs/Unity3D.Amqp (citado na pg. 38).

[*Clojure HTTP library* s.d.] *GitHub - dakrone/clj-http: An idiomatic clojure http client wrapping the apache client. Officially supported version.* URL: https://github.com/dakrone/clj-http (citado na pg. 37).

[*wasi-experimental-http* s.d.] *GitHub - deislabs/wasi-experimental-http: Experimental outbound HTTP support for WebAssembly and WASI.* URL: https://github.com/deislabs/wasi-experimental-http (citado na pg. 38).

[*amqp-coffee* s.d.] *GitHub - dropbox/amqp-coffee: An AMQP 0.9.1 client for Node.js.* URL: https://github.com/dropbox/amqp-coffee (citado na pg. 38).

[*elixir-mint* s.d.] *GitHub - elixir-mint/mint: Functional HTTP client for Elixir with support for HTTP/1 and HTTP/2.* URL: https://github.com/elixir-mint/mint (citado na pg. 37).

[*FSharp HTTP library* s.d.] *GitHub - fsprojects/FsHttp: A lightweight FSharp HTTP library by @ronaldschlenker.* URL: https://github.com/fsprojects/FsHttp (citado na pg. 37).

[*Lisp HTTP library* s.d.] *GitHub - fukamachi/dexador: A fast HTTP client for Common Lisp.* URL: https://github.com/fukamachi/dexador (citado na pg. 37).

[*Go REST Client* s.d.] *GitHub - go-resty/resty: Simple HTTP and REST client library for Go.* URL: https://github.com/go-resty/resty (citado na pg. 37).

[*Haskell AMQP client* s.d.] *GitHub - hreinhardt/amqp: Haskell AMQP client library.* URL: https://github.com/hreinhardt/amqp (citado na pg. 37).

[*RabbitMQ JRuby Client* s.d.] *GitHub - jerryluk/rabbitmq-jruby-client: RabbitMQ JRuby Client using Java RabbitMQ client.* URL: https://github.com/jerryluk/rabbitmq-jruby-client (citado na pg. 37).

[*httpbuilder* s.d.] *GitHub - jgritman/httpbuilder.* URL: https://github.com/jgritman/httpbuilder (citado na pg. 37).

[*AMQPClient.jl* s.d.] *GitHub - JuliaComputing/AMQPClient.jl: A Julia AMQP (Advanced Message Queuing Protocol) / RabbitMQ Client.* URL: https://github.com/JuliaComputing/AMQPClient.jl (citado na pg. 37).

[*JuliaWeb/HTTP.jl* s.d.] *GitHub - JuliaWeb/HTTP.jl: HTTP for Julia.* URL: https://github.com/JuliaWeb/HTTP.jl (citado na pg. 37).

[*RabbitMQ interface for Lisp* s.d.] *GitHub - lokedhs/cl-rabbit: RabbitMQ interface to Common Lisp.* URL: https://github.com/lokedhs/cl-rabbit (citado na pg. 37).

[*Faraday* s.d.] *GitHub - lostisland/faraday: Simple, but flexible HTTP client library, with support for multiple backends.* URL: https://github.com/lostisland/faraday (citado na pg. 37).

[*Lua HTTP Client* s.d.] *GitHub - lusis/lua-httpclient: A unified http/s client library for lua*. URL: https://github.com/lusis/lua-httpclient (citado na pg. 37).

[*MATLAB RabbitMQ Interface* s.d.] *GitHub - mathworks-ref-arch/matlab-rabbitmq: MATLAB Interface for RabbitMQ*. URL: https://github.com/mathworks-ref-arch/matlab-rabbitmq (citado na pg. 37).

[*OCaml HTTP Lib* s.d.] *GitHub - mirage/ocaml-cohttp: An OCaml library for HTTP clients and servers using Lwt or Async*. URL: https://github.com/mirage/ocaml-cohttp (citado na pg. 37).

[*php-amqplib* s.d.] *GitHub - php-amqplib/php-amqplib: The most widely used PHP client for RabbitMQ*. URL: https://github.com/php-amqplib/php-amqplib (citado na pg. 37).

[*RabbitMQ Elixir client* s.d.] *GitHub - pma/amqp: Idiomatic Elixir client for RabbitMQ*. URL: https://github.com/pma/amqp (citado na pg. 37).

[*RabbitMQ stored procedures* s.d.] *GitHub - pmq/rabbitmq-oracle-stored-procedures: RabbitMQ stored procedures for Oracle DB*. URL: https://github.com/pmq/rabbitmq-oracle-stored-procedures (citado na pg. 37).

[*AMQP Go client* s.d.] *GitHub - rabbitmq/amqp091-go: An AMQP 0-9-1 Go client maintained by the RabbitMQ team*. URL: https://github.com/rabbitmq/amqp091-go (citado na pg. 37).

[*RabbitMQ for Objective-C/Swift* s.d.] *GitHub - rabbitmq/rabbitmq-objc-client: RabbitMQ client for Objective-C and Swift*. URL: https://github.com/rabbitmq/rabbitmq-objc-client/ (citado na pg. 37).

[*HTTP-Easy* s.d.] *GitHub - raku-community-modules/HTTP-Easy: Make HTTP servers (with PSGI support) easily with Raku*. URL: https://github.com/raku-community-modules/HTTP-Easy (citado na pg. 37).

[*HTTP .NET Client* s.d.] *GitHub - restsharp/RestSharp: Simple REST and HTTP API Client for .NET*. URL: https://github.com/restsharp/RestSharp (citado na pg. 37).

[*P6-Net-AMQP* s.d.] *GitHub - retupmoca/P6-Net-AMQP*. URL: https://github.com/retupmoca/P6-Net-AMQP (citado na pg. 37).

[*HTTP client for Unity3D* s.d.] *GitHub - satanas/unity-simple-http: A dead simple HTTP client for Unity3D*. URL: https://github.com/satanas/unity-simple-http (citado na pg. 38).

[*Rust HTTP client* s.d.] *GitHub - seanmonstar/reqwest: An easy and powerful Rust HTTP Client*. URL: https://github.com/seanmonstar/reqwest (citado na pg. 37).

[*Scala AMQP client* s.d.] *GitHub - sstone/amqp-client: Simple fault-tolerant AMQP client written in Scala*. URL: https://github.com/sstone/amqp-client (citado na pg. 37).

[*rabbitmq-d* s.d.] *GitHub - symmetryinvestments/rabbitmq-d: Bindings for librabbitmq-c for the D programming language*. URL: https://github.com/symmetryinvestments/rabbitmq-d (citado na pg. 38).

[*Lua RabbitMQ client* s.d.] *GitHub - wingify/lua-resty-rabbitmqstomp: Opinionated Lua RabbitMQ client library*. URL: https://github.com/wingify/lua-resty-rabbitmqstomp (citado na pg. 37).

[*Yii 2 HTTP client* s.d.] *GitHub - yiisoft/yii2-httpclient: Yii 2 HTTP client*. URL: https://github.com/yiisoft/yii2-httpclient (citado na pg. 37).

[*AdaBinding* s.d.] *GTI-IA. Grupo de Tecnología Informática-Inteligencia Artificial*. URL: http://www.gti-ia.upv.es/sma/tools/AdaBinding/index.php (citado na pg. 38).

[*habari-rabbitmq* s.d.] *Habari Client for RabbitMQ - Habarisoft*. URL: http://www.habarisoft.com/habari_rabbitmq.html (citado na pg. 37).

[Hassan e Bahsoon 2016] Sara Hassan e Rami Bahsoon. "Microservices and their design tradeoffs: a self-adaptive roadmap". Em: *2016 IEEE International Conference on Services Computing (SCC 2016)* (2016) (citado na pg. 12).

[*HTTP from COBOL* s.d.] *How to make http requests from COBOL - Stack Overflow*. URL: https://stackoverflow.com/questions/26367026/how-to-make-http-requests-from-cobol (citado na pg. 37).

[*Utility Library* s.d.] *HTTP - Ada Utility Library*. URL: https://ada-util.readthedocs.io/en/latest/Util_Http/ (citado na pg. 38).

[*Dart HTTP Package* s.d.] *http - Dart Package*. URL: https://pub.dev/packages/http (citado na pg. 37).

[*ELM Http Package* s.d.] *http 2.0.0*. URL: https://package.elm-lang.org/packages/elm/http/latest/ (citado na pg. 37).

[*Gerbil HTTP* s.d.] *HTTP requests - Gerbil Scheme*. URL: https://cons.io/reference/requests.html (citado na pg. 38).

[*Haskell HTTP Client* s.d.] *http-client: An HTTP client engine*. URL: https://hackage.haskell.org/package/http-client (citado na pg. 37).

[*Crystal Http Client* s.d.] *HTTP::Client - Crystal 1.6.1*. URL: https://crystal-lang.org/api/1.6.1/HTTP/Client.html (citado na pg. 37).

[*HTTP::Tiny* s.d.] *HTTP::Tiny - A small, simple, correct HTTP/1.1 client - Perldoc Browser*. URL: https://perldoc.perl.org/HTTP::Tiny (citado na pg. 37).

[*Arduino HTTP Client* s.d.] *HttpClient - Arduino Reference*. URL: https://www.arduino.cc/reference/en/libraries/httpclient/ (citado na pg. 37).

[*Swift HTTP Client* s.d.] *HTTPClient – Swift Package Index*. URL: https://swiftpackageindex.com/uhooi/swift-http-client (citado na pg. 37).

[Indrasiri e Siriwardena 2018] Kasun Indrasiri e Prabath Siriwardena. "Integrating microservices". Em: *Microservices for the Enterprise*. Springer, 2018, pgs. 167–217 (citado na pg. 12).

[Jacopo Soldani e Heuvel 2018] Damian Andrew Tamburri Jacopo Soldani e Willem-Jan Van Den Heuvel. "The pains and gains of microservices: a systematic grey literature review". Em: *Journal of Systems and Software 146 (2018)* (2018) (citado na pg. 12).

[Jetbrains 2021] Jetbrains. *DevEcosystem 2021*. 2021. URL: https://www.jetbrains.com/lp/devecosystem-2021/microservices/ (acesso em 04/10/2022) (citado na pg. 36).

[*jruby-httpclient* s.d.] *jruby-httpclient - RubyGems.org - your community gem host*. URL: https://rubygems.org/gems/jruby-httpclient/versions/1.1.1-java (citado na pg. 37).

[*LabbitMQ Toolkit* s.d.] *LabbitMQ Toolkit for LabVIEW - Download - VIPM by JKI*. URL: https://www.vipm.io/package/distrio_labbitmq/ (citado na pg. 38).

[*Clojure AMQP client* s.d.] *Langohr, an idiomatic Clojure RabbitMQ client - Clojure AMQP 0.9.1 client*. URL: http://clojurerabbitmq.info/ (citado na pg. 37).

[*libcurl.vb* s.d.] *libcurl.vb download - SourceForge.net*. URL: https://sourceforge.net/projects/libcurl-vb/ (citado na pg. 37).

[*LibHTTP* s.d.] *LibHTTP – Open Source HTTP Library in C – Cross platform HTTP and HTTPS library*. URL: https://www.libhttp.org/ (citado na pg. 37).

[*HTTP interface* s.d.] *Making HTTP and HTTPS Requests*. URL: https://developer.apple.com/library/archive/documentation/NetworkingInternetWeb/Conceptual/NetworkingOverview/WorkingWithHTTPAndHTTPSRequests/WorkingWithHTTPAndHTTPSRequests.html (citado na pg. 37).

[Marquez e Astudillo 2018] Gaston Marquez e Hernan Astudillo. "Actual use of architectural patterns in microservices-based open source projects". Em: *25th Asia-Pacific Software Engineering Conference (APSEC)* (2018) (citado na pg. 1).

[*messageQueue-package* s.d.] *messageQueue-package: Allow R to communicate with message queues*. URL: https://rdrr.io/rforge/messageQueue/man/messageQueue-package.html (citado na pg. 37).

[Newman 2015] Sam Newman. "Building microservices: designing fine-grained systems". Em: *O'Reilly Media* (2015) (citado nas pgs. 11, 12).

[Oliveira Rosa 2021a] Thatiane de Oliveira Rosa. *CharM Presentation*. 2021. URL: https://www.youtube.com/watch?v=VQRqG9hLBSQ (acesso em 11/11/2022) (citado na pg. 9).

[Oliveira Rosa 2021b] Thatiane de Oliveira Rosa. *Demonstration of application of the CharM*. 2021. URL: https://www.youtube.com/watch?v=bK9Yg9jmQXY (acesso em 11/11/2022) (citado na pg. 9).

[Pahl e Jamshidi 2016] Claus Pahl e Pooyan Jamshidi. "Microservices: a systematic mapping study". Em: *6th International Conference on Cloud Computing and Services Science (CLOSER 2016)* (2016) (citado na pg. 32).

[*pika - PyPI* s.d.] *pika - PyPI*. URL: http://pypi.python.org/pypi/pika (citado na pg. 37).

[*RabbitMQ Java Client* s.d.] *RabbitMQ Java Client Library - RabbitMQ*. URL: https://www.rabbitmq.com/java-client.html (citado na pg. 37).

[*RabbitMQ Grails Plugin* s.d.] *RabbitMQ Native Grails Plugin Documentation*. URL: https://plugins.grails.org/plugin/budjb/rabbitmq-native (citado na pg. 37).

[*Arduino RabbitMQ Lib* s.d.] *RabbitMQ with Arduino Uno - Stack Overflow*. URL: https://stackoverflow.com/questions/44865473/rabbitmq-with-arduino-uno (citado na pg. 37).

[*Requests* s.d.] *Requests: HTTP for Humans — Requests 2.28.1 documentation*. URL: https://requests.readthedocs.io/ (citado na pg. 37).

[Richards 2015] Mark Richards. "Software architecture patterns". Em: *O'Reilly Media* (2015) (citado na pg. 12).

[Richardson 2018] Chris Richardson. "Microservices patterns". Em: *Manning Publications Co* (2018) (citado nas pgs. 12, 31).

[*HTTP for SAP* s.d.] *SAP Help Portal*. URL: https://help.sap.com/docs/SAP_NETWEAVER_750/753088fc00704d0a80e7fbd6803c8adb/48c7c53bda5e31ebe10000000a42189b.html?version=7.5.8 (citado na pg. 38).

[*SAS AMQP Package* s.d.] *SAS Help Center*. URL: https://documentation.sas.com/doc/en/espcdc/6.1/espca/p1k84sxazgi57dn19ij9yz9tu6u6.htm (citado na pg. 37).

[*SAS HTTP client* s.d.] *SAS Help Center*. URL: https://documentation.sas.com/doc/en/pgmsascdc/9.4_3.5/proc/n0bdg5vmrpyi7jn1pbgbje2atoov.htm (citado na pg. 37).

[*Prolog HTTP Package* s.d.] *section('packages/http.html')*. URL: https://www.swi-prolog.org/pldoc (citado na pg. 37).

[*HTTP for Labview* s.d.] *Solved: Labview POST, PUT HTTP method - NI Community*. URL: https://forums.ni.com/t5/LabVIEW-Web-Development/Labview-POST-PUT-HTTP-method/td-p/3663058 (citado na pg. 38).

[Spectrum 2022] IEEE Spectrum. *Top Programming Languages 2022*. 2022. URL: https://spectrum.ieee.org/top-programming-languages-2022 (acesso em 04/10/2022) (citado na pg. 36).

[*SQL httprequest* s.d.] *SQL httprequest*. URL: https://sqlhttp.net/documentation/http/httprequest/ (citado na pg. 37).

[Statista 2021] Statista. *Primary programming languages among microservices developers worldwide in 2021*. 2021. URL: https://www.statista.com/statistics/1273806/microservice-developers-programming-language/ (acesso em 04/10/2022) (citado na pg. 36).

[*Struct HTTP* s.d.] *Struct HTTP - D Programming Language*. URL: https://dlang.org/library/std/net/curl/http.html (citado na pg. 38).

[*Scala HTTP client* s.d.] *sttp: the Scala HTTP client you always wanted! — sttp 3 documentation*. URL: https://sttp.softwaremill.com/en/latest/ (citado na pg. 37).

[al 2020a] Thatiane de Oliveira Rosa, Eduardo Martins Guerra, and Alfredo Goldman. "Modelo para caracterização e evolução de sistemas com arquitetura baseada em serviços". Em: (2020) (citado nas pgs. 2, 9, 39).

[al 2020b] Thatiane de Oliveira Rosa, João Francisco Lino Daniel, Eduardo Martins Guerra, and Alfredo Goldman. "A method for architectural trade-off analysis based on patterns: evaluating microservices structural attributes". Em: *European Conference on Pattern Languages of Programs 2020 (EuroPLoP 20)* (2020). DOI: 10.1145/3424771.3424809 (citado nas pgs. 2, 9, 39).

[*MATLAB HTTP interface* s.d.] *Use HTTP with MATLAB - MATLAB - Simulink*. URL: https://www.mathworks.com/help/matlab/http-interface.html (citado na pg. 37).

[*RADStudio HTTP library* s.d.] *Using an HTTP Client*. URL: https://docwiki.embarcadero.com/RADStudio/Alexandria/en/Using_an_HTTP_Client (citado na pg. 37).

[*F-RabbitMQ* s.d.] *Utilizing RabbitMQ with FSharp - codesuji*. URL: https://www.codesuji.com/2019/10/24/F-RabbitMQ/ (citado na pg. 37).

[Verão - IME USP 2021] Programa de Verão - IME USP. *Pingr Complexos*. 2021. URL: https://pingr-complexos.netlify.app/ (acesso em 18/09/2022) (citado nas pgs. 2, 14).

[Wagner e Rousos 2022] Bill Wagner e Mike Rousos. *Communication in a microservice architecture*. 2022. URL: https://learn.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/communication-in-microservice-architecture (acesso em 11/11/2022) (citado na pg. 12).