UNIVERSIDADE FEEVALE

RONEI DOS SANTOS BERLEZI

ANÁLISE COMPARATIVA SOBRE MODELOS DE PERSISTÊNCIA DE DADOS EM APLICAÇÕES BASEADAS NA ARQUITETURA DE MICROSSERVIÇOS

Novo Hamburgo

2019

RONEI DOS SANTOS BERLEZI

ANÁLISE COMPARATIVA SOBRE MODELOS DE PERSISTÊNCIA DE DADOS EM APLICAÇÕES BASEADAS NA ARQUITETURA DE MICROSSERVIÇOS

Trabalho de Conclusão de Curso
apresentado como requisito parcial
à obtenção do grau de Bacharel em
Ciência da Computação pela
Universidade Feevale

Orientador: Dr. Rodrigo Rafael Villarreal Goulart

Novo Hamburgo
2019

AGRADECIMENTOS

Gostaria de agradecer a todos os que, de alguma maneira, contribuíram para a realização desse trabalho de conclusão, em especial:

A minha família e minha esposa, também ao meu cachorro Bóris, que esteve presente comigo durante toda a elaboração do projeto.

RESUMO

Com o crescente aumento da complexidade dos sistemas corporativos, surgiu um novo termo, Microsserviços. Este termo surgiu nos últimos anos para descrever uma maneira específica de desenvolver aplicações com o isolamento de responsabilidades sobre o sistema como um todo. Microsserviços, por ser uma arquitetura de pequenos projetos com um só foco, traz inúmeras vantagens, como independência de tecnologia para construir os módulos, possibilidade de mudança rápida em um único serviço sem necessidade de retestar e gerar um novo *deploy* para toda a aplicação, alta escalabilidade, etc. No contexto de banco de dados em sistemas monolíticos, transações utilizam o conjunto de propriedades ACID (Atomicidade, Consistência, Isolamento e Durabilidade) para garantir a integridade dos dados. Já na arquitetura de microsserviços, os serviços devem estar desacoplados e, portanto, cada um pode ter seu próprio repositório de dados, apesar de algumas implementações utilizarem banco de dados relacional para isso, a grande maioria das aplicações baseadas em microsserviços utilizam bancos NoSQL. A implementação de transações que utilizam dados de múltiplos serviços acaba aumentando a complexidade do sistema, ao mesmo tempo que muitos dos bancos de dados não suportam protocolos de transações distribuídas. Sendo assim, este trabalho tem como propósito elucidar os conceitos envolvidos na implementação de uma arquitetura baseada em microsserviços, levando em consideração principalmente o comportamento dos dados sobre um sistema de armazenamento de dados distribuído, levantando pontos como consistência e escalabilidade neste contexto, assim como a avaliação de desempenho entre dois dos modelos analisados. Os resultados foram obtidos realizando uma sequência de testes sobre o modelo *Event Sourcing* e Saga, variando parâmetros como o número de requisições simultâneas e quantidades de instâncias de cada microsserviço. O modelo *Event Sourcing* apresentou melhor desempenho em todos os cenários, também melhorando proporcionalmente o desempenho em relação ao acréscimo de novas instâncias, enquanto o modelo Saga teve desempenho inferior e o aumento de desempenho ocorreu somente no primeiro acréscimo de instâncias.

Palavras-chave: Microsserviços. *Eventuate*. *Sagas*. *Event Sourcing*.

ABSTRACT

With the increasing complexity of corporate systems, a new term, Microservices, has emerged. This term has emerged in recent years to describe a specific way to develop applications with the isolation of responsibilities over the system as a whole. This architecture has limitations, because each time a deploy is performed, the entire application must be restarted. Microservices, because it is an architecture of small projects with a single focus, brings innumerable advantages, such as independence of technology to build the modules, possibility of rapid change in a single service without need to rest and generate a new deploy for the whole application, high scalability, etc. In the context of a database in monolithic systems, transactions use the set of ACID properties (Atomicity, Consistency, Isolation and Durability) to guarantee the integrity of the data. In microservice architecture, the services must be decoupled and, therefore, each should have its own data repository, although some implementations use relational database for this, the vast majority of the applications based on microservices use NoSQL databases. Implementing transactions that use data from multiple services ends up increasing system complexity, while many databases do not support distributed transactions. Therefore, this work aims to analyze the methods to persist data within a system based on the microservice architecture, as well a performance comparison between two of these methods. Results were obtained by running a set of tests against the Event Sourcing and Saga patterns, with different value parameters for concurrent number of requests and number of microservices instances. Event Sourcing resulted to be more performatic in all scenarios, also had a better performance by increasing the number of microservices instances, while Saga had a lower performance and the performance was increased only during the first increasing of instances.

Keywords: Miroservices. Eventuate. Sagas. Event Sourcing.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

ACID	Atomicidade, Consistência, Isolamento e Durabilidade
API	Application Program Interface
HTML	Hyper Text Markup Language
HTTP	Hyper Text Transfer Protocol
JSON	JavaScript Object Notification
NoSQL	Not Only SQL
REST	Representational State Transfer
SOA	Service Oriented Architecture
SOAP	Service Oriented Architecture Protocol
SQL	Structured Query Language
XML	eXtensible Markup Language

SUMÁRIO

1. INTRODUÇÃO

Vivemos em uma constante mudança de tecnologias e abordagens no âmbito do desenvolvimento de aplicações, fazendo com que reavaliemos como construímos e entregamos soluções aos nossos clientes. Um exemplo disso é a adoção de várias empresas pela utilização da arquitetura baseada em microsserviços. Esta arquitetura é composta por pequenos serviços autônomos que trabalham juntos, definidos como pequenos módulos responsáveis por fazer uma única coisa bem feita, de forma que alterações não afetam os demais, e ao mesmo tempo permite a geração de um novo *deploy* por si próprio sem a necessidade de alteração nos consumidores (NEWMAN, 2015).

Microsserviços ganharam popularidade na indústria nos últimos anos. Esta arquitetura pode ser considerada um refinamento e simplificação de *Service-Oriented Architecture (SOA)* (AMARAL et al., 2016).

SOA emergiu para combater os desafios de grandes aplicações monolíticas através da quebra do sistema em uma série de serviços que trabalham de forma colaborativa para fornecer um conjunto de funcionalidades. A ideia em si é muito sensata, mas apesar dos esforços da comunidade, houve uma falta de consenso em como aplicar o SOA. Muitos dos problemas envolviam protocolos de comunicação, como o SOAP (*Service-Oriented Architecture Protocol*), e principalmente a falta de orientação sobre a granularidade de serviços (NEWMAN, 2015).

A arquitetura de microsserviços veio como uma variação de SOA, este conceito começou a ser adotado pela comunidade em 2014 como uma resposta direta a alguns dos desafios de escalabilidade, tanto tecnica quanto organizacional de grandes aplicações monolíticas (CARNELL, 2017).

Os benefícios apresentados pela arquitetura de microsserviços incluem bases de códigos simplificadas para serviços individuais, habilidade de atualizar e escalar serviços de forma isolada, possibilidade de escrita de serviços em diferentes linguagens e a utilização de diferentes camadas de dados (AMARAL et al., 2016).

Um dos maiores desafios de sistemas baseados em microsserviços é a divisão dos serviços. Cada um deles deve ser simples o suficiente a ponto de possuir poucas

responsabilidades. Esta fragmentação da aplicação pode transformar o gerenciamento de dados no gargalo do sistema (MESSINA et al., 2013).

Existem dois padrões predominantes na implementação de banco de dados, por serviço e compartilhado (KALSKE, 2017). A utilização de um banco de dados compartilhado como forma de integração entre os serviços pode fazer sentido em um primeiro momento, é uma forma rápida e fácil de implementação, mas ao utilizar esta técnica estamos permitindo que recursos externos sejam ligados diretamente ao modelo. Ao modificar a modelagem de dados, deve ser feita uma análise em todos os serviços, afim de evitar que outros consumidores não quebrem (NEWMAN, 2015).

Deve-se evitar a comunicação indireta entre microsserviços pela utilização de banco de dados compartilhado. Uma das ramificações deste princípio é que usualmente cada microsserviço irá armazenar seus dados em um banco isolado (BROWN; WOOLF, 2016). Com a utilização do padrão de banco de dados por serviço, o acesso a informações de outros contextos ocorre por forma de chamadas aos serviços responsáveis. Ao descentralizar os dados, existem implicações durante o gerenciamento das atualizações. Transações garantem que toda a informação seja armazenada no banco, ou nenhuma. Esta abordagem é comumente utilizada em arquiteturas monolíticas. Transações distribuídas são notoriamente difíceis de implementar e por consequência as arquiteturas de microsserviços enfatizam a coordenação sem transação entre os serviços, com reconhecimento explícito de consistência eventual e de que os problemas serão resolvidos por operações de compensação (LEWIS; FOWLER, 2014).

O presente trabalho tem como objetivo elucidar os conceitos envolvidos na implementação de uma arquitetura baseada em microsserviços, levando em consideração principalmente o comportamento dos dados sobre um sistema de armazenamento de dados distribuído, levantando pontos como consistência e escalabilidade neste contexto. Os modelos *Saga* orquestrada e *Event Sourcing* são utilizados neste estudo, a fim de clarificar os conceitos por trás de cada um deles, comparando o desempenho de cada um em um ambiente clusterizado.

Este trabalho está divido em 7 capítulos. O primeiro capítulo aborda a motivação para o surgimento da arquitetura de microsserviços. O segundo capítulo aborda as principais características e vantagens envolvidas na arquitetura de microsserviços. O terceiro capítulo apresenta os padrões arquiteturais mais utilizados dentro deste modelo, tanto no que se refere à comunicação entre serviços quanto ao armazenamento de dados. No quarto capítulo, são explorados as possíveis maneiras de escalar uma aplicação, fazendo uma avaliação sobre o *Scale Cube*. No capítulo 5 são apresentados os principais conceitos e padrões utilizados para o

armazenamento de dados e implementação para garantia de consistência em dados distribuídos, realizando uma análise sobre o teorema CAP, padrão SAGA, transações distribuídas e Event Sourcing. No capítulo 6, é apresentado um cenário de teste para comparação entre o modelo *Event Sourcing* e SAGA. Este capítulo consiste da explicação dos experimentos e resultados de cada implementação em dois ambientes distintos. Por fim, o capítulo 7 apresenta os resultados obtidos durante a execução deste trabalho.

A relevância desta pesquisa contribui, diretamente, para ser utilizada em estudos futuros com relação a transações com dados distribuídos na arquitetura de microsserviços, e serve de base para outros trabalhos que venham a abordar este tema, sendo que este trabalho sintetiza diversas características de funcionamento da arquitetura de microsserviços e a natureza de dados distribuídos em alguns padrões para implementação de regras de negócio.

2. MICROSSERVIÇOS

A computação em nuvem introduziu o conceito de não utilizar computadores “reais”, mas sim utilizar o poder de computação de forma dinâmica, controlado por um sistema automatizado, portanto aumentando e diminuindo a capacidade dos recursos de acordo com a demanda. Esta mudança no paradigma de desenvolvimento levou empresas a largarem o modelo tradicional de desenvolver grandes aplicações monolíticas e adotarem o modelo onde pequenos times são responsáveis pela construção de pequenos módulos do sistema, estes módulos com responsabilidades limitadas, chamados de Microsserviços.

O surgimento deste modelo decorre, ao mesmo tempo, devido ao aumento da complexidade das aplicações modernas, principalmente no que tange à quantidade de clientes fazendo requisições a um mesmo sistema, como em aplicações da Netflix, Amazon, Spotify, Uber e etc. Surgem então novas propostas de arquiteturas para que tornem factível o atendimento desta crescente demanda. É apresentado então o modelo de arquitetura baseado em microsserviços.

Este capítulo tem como objetivo apresentar os conceitos que definem a arquitetura de microsserviços, assim como levantar pontos crucias que expliquem as características, componentes e desafios decorrentes deste modelo.

2.1. Arquitetura de Microsserviços

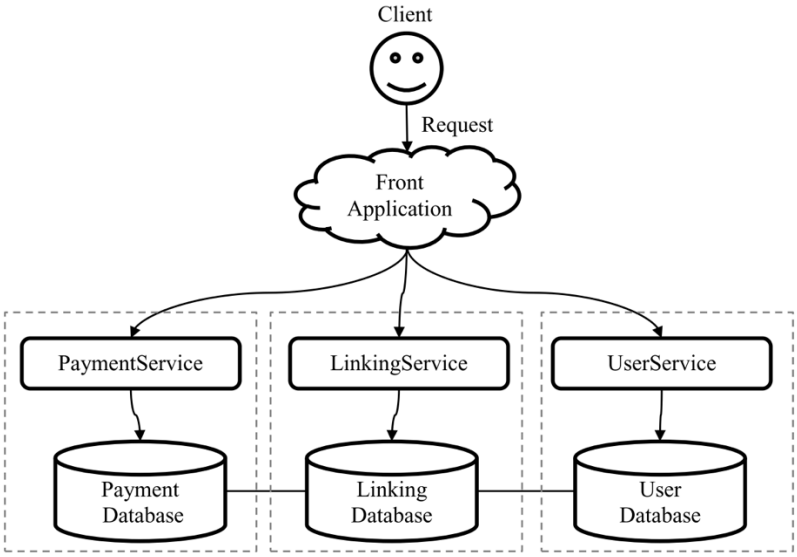
De acordo com Lewis e Fowler (2014), o padrão arquitetural de microsserviços é uma abordagem para o desenvolvimento de uma única aplicação como um conjunto de pequenos serviços, sendo que cada uma possui seu próprio processo e se comunica com os demais através de mecanismos leves, frequentemente utilizando uma *Application Program Interface* (API) sobre o *Hyper Text Transfer Protocol* (HTTP). Estes serviços são construídos em torno de capacidades de negócios e são implantáveis isoladamente através de mecanismos de *deploy* automatizado. Há um gerenciamento mínimo centralizado sobre estes serviços, os quais podem ser escritos em diferentes linguagens de programação e usar diferentes tecnologias de armazenamento de dados.

Microsserviços se baseiam na decomposição funcional, frequentemente através do contexto de *Domain-Driven Design*. Eles são caracterizados por interfaces bem definidas e publicados explicitamente. Cada serviço é completamente autônomo, consequentemente,

mudanças feitas na implementação de um serviço não causam impacto nos demais, pois a comunicação é dada somente através de interfaces. A decomposição funcional da aplicação é a chave para o desenvolvimento de uma arquitetura baseada em microsserviços. Isso garante menor acoplamento - interfaces REST (*Representational State Transfer*) - e uma alta coesão - múltiplos serviços podem compor uns com os outros para definir serviços ou aplicação de maior complexidade - Decomposição funcional permite, por exemplo, agilidade, flexibilidade e escalabilidade (PAHL; JAMSHIDI, 2016).

A Figura 1 demonstra o exemplo de uma proposta de aplicação baseada em microsserviços, composta de serviços como de pagamento, usuário, etc.

Figura 1 - Arquitetura de Microsserviços



Fonte: Risthein (2015)

2.2. Arquitetura Monolítica

Uma arquitetura monolítica é uma aplicação que usa uma única base de código para implementar um sistema e diferentes interfaces, como o REST (*Representational State Transfer)*, páginas HTML e APIs. A abordagem monolítica é considerada como o modelo padrão para iniciar a o desenvolvimento de aplicações. Um único repositório de código facilita o desenvolvimento, implementação e escalabilidade da aplicação, desde que o tamanho deste código seja relativamente pequeno. A escolha pela utilização deste modelo é vantajosa no início do projeto devido aos atributos mencionados acima e também porque não há nenhuma distribuição de código o qual aumentaria a complexidade (KALSKE, 2017).

Segundo Lewis e Fowler (2014), para entender melhor microsserviços, é importante comparar com a arquitetura monolítica. As aplicações corporativas são construídas

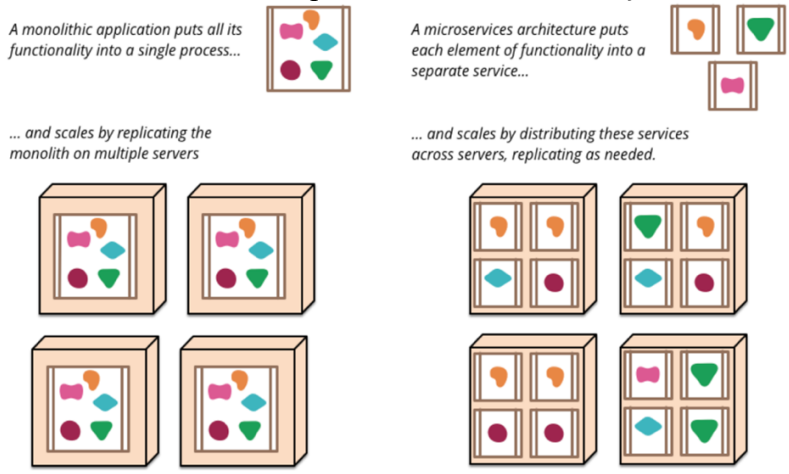
frequentemente de três partes principais: uma interface de usuário – consistindo de páginas HTML e Javascript que são executadas no navegador do usuário – um banco de dados – consistindo de diversas tabelas inseridas normalmente em um único banco de dados relacional – e uma aplicação rodando no servidor. A aplicação no servidor será responsável por lidar com as requisições HTTP, executar a lógica de domínio, buscar e atualizar dados do banco de dados e popular as páginas HTML que serão enviadas para o navegador. Esta aplicação no servidor é monolítica – um único executável. Qualquer mudança no sistema requer a compilação da aplicação inteira e o *redeploy* desta nova versão.

A utilização de um servidor monolítico é um modo natural no desenvolvimento de um sistema como este. Toda a lógica para tratar requisições é executa em um único processo, permitindo o uso de recursos da linguagem utilizada para a divisão da aplicação em classes, pacotes ou módulos. Então é possível utilizar um *pipeline* para a executar os testes e realizar o *deploy* da aplicação em produção. É possível escalar a aplicação horizontalmente, de forma a executar instâncias réplicas atrás de um *load-balancer* (LEWIS; FOWLER, 2014).

Este modelo tradicional de desenvolvimento de aplicações pode ser de sucesso, mas equipes de desenvolvimento estão se frustrando com ele, especialmente com o crescente número de aplicações sendo implementadas na nuvem. A aplicação é altamente acoplada, o que significa que mudanças requerem um *build* no sistema como um todo, e um novo *deploy*. Escalar este modelo significa replicar a aplicação, o que acaba consumindo mais recursos do que escalar módulos ou serviços (LEWIS; FOWLER, 2014).

A Figura 2 mostra o consumo de recursos para escalar microsserviços e aplicações monolíticas.

Figura 2 - Escalabilidade das arquiteturas de microsserviços e monolíticas



Fonte: Lewis e Fowler (2014)

2.3. Principais Vantagens

Microsserviços oferecem várias vantagens, e diversas empresas adotam este padrão arquitetônico para resolver limitações e desafios encontrados durante e após o processo de desenvolvimento de seus sistemas, como, agilidade no processo de *deploy*, escalabilidade e disponibilidade da aplicação.

2.3.1. Liberdade de Tecnologia

De acordo com Newman (2015), uma das vantagens é a heterogeneidade de tecnologias que podem ser usadas para construir diferentes serviços. Sendo um sistema baseado em microsserviços, com múltiplos serviços colaborativos, é possível utilizar diferentes tecnologias para cada um deles. Isso permite que utilizemos a ferramenta mais adequada para cada tipo de trabalho, ao invés de ter que usar uma abordagem padronizada que não atenderá as necessidades de desenvolvimento em todos os cenários.

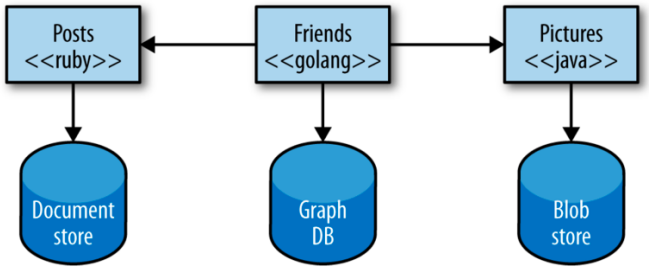
Microsserviços oferece esta liberdade para a escolha da tecnologia, pois eles se comunicam através da rede, e podem ser implementados em qualquer linguagem e plataforma, desde que a comunicação com os demais serviços seja possível (WOLFF, 2016).

Segundo Chen (2018), é de suma importâncias que se possa experimentar e adotar novas tecnologias de forma rápida, para assim manter os produtos competitivos. O impacto de novas tecnologias nos produtos pode ser tão importante como novas funcionalidades.

Se uma parte do sistema precisa melhorar o desempenho, podemos decidir por usar uma tecnologia diferente que melhor se adequa a este requisito de desempenho. Podemos decidir armazenar os dados em diferentes tipos de bancos de acordo com o serviço de nosso sistema. Por exemplo, para uma rede social, podemos armazenar as interações dos usuários em um banco de dados orientado a gráfos, de forma a refletir a natureza interconectada de um gráfo social. Posts realizados pelos usuários poderiam ser armazenados em um banco orientado ao armazenamento de documentos, surgindo então uma arquitetura heterogenia como a mostrada na Figura 3 (NEWMAN, 2015).

De acordo com Richardson (2015), as empresas devem evitar a completa anarquia e limitar os tipos de tecnologia. De qualquer maneira, esta liberdade significa que os desenvolvedores não são mais obrigados a usar tecnologias obsoletas que existiam antes do início do projeto. Serviços são relativamente pequenos, e agora se torna factível a reescrita de um serviço antigo utilizando uma tecnologia atual.

Figura 3 - Microsserviços e diferentes tecnologias



Fonte: Newman (2015)

2.3.2. Resiliência e Estabilidade

Segundo Wolff (2016), em um sistema baseado na arquitetura de microsserviços e bem projetado, a falha de um único serviço deve ter um impacto mínimo sobre a disponibilidade dos demais serviços que compõem o sistema. Como a natureza desta arquitetura é distribuída, o número de servidores por sistema é muito maior, e isso também aumenta a chance de falha. Deve ser evitado que falhas geradas em um serviço se propague para os demais, de forma a criar um efeito cascata.

Serviços desacoplados significam que uma aplicação não é mais apenas um "lamaçal", onde a degradação de uma parte da aplicação pode causar a falha da aplicação inteira. Falhas podem estar localizadas em uma pequena parte da aplicação e serem contidas antes que toda a aplicação sofra uma interrupção. Isso também permite que os serviços se degradem de forma não crítica em caso de erros não recuperáveis (CARNELL, 2017).

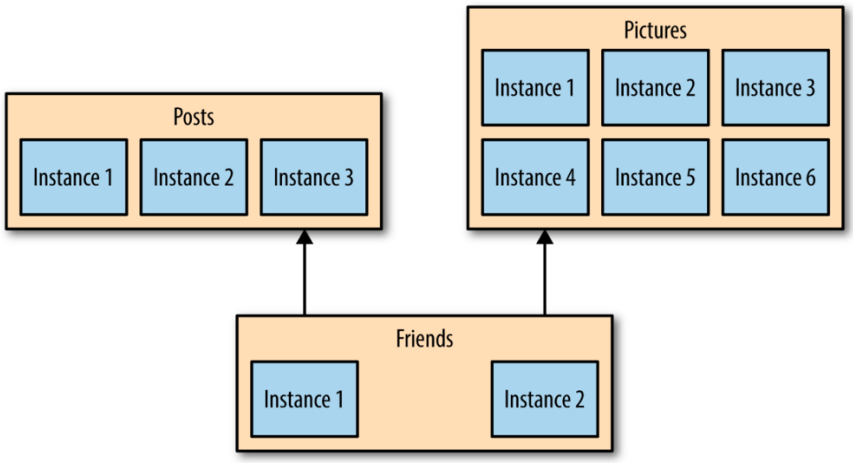
De acordo com Newman (2015), um conceito chave da engenharia de resiliência é o *bulkhead*, definido como um método para se isolar da falha. Nos navios, *bulkheads* são estruturas projetadas para caso haja um vazamento no casco, dessa maneira, pode-se fechar as portas do *bulkhead*, perdendo somente parte do barco, mantendo o resto do navio intacto.

Abordagens semelhantes são aplicáveis em software: todo o sistema pode ser separado em áreas individuais. Uma quebra de serviço ou um problema em determinada área não deve afetar as outras áreas. Por exemplo, pode-se ter diversas instâncias de um mesmo microsserviço para diferentes clientes. Se um cliente sobrecarrega o microsserviço, os demais não serão negativamente afetados. O mesmo é verdadeiro para recursos como conexões com bancos de dados ou *threads*. Quando diferentes partes de um microsserviço usam diferentes *pools* para estes recursos, uma parte não pode bloquear as outras partes, mesmo se ela usar todos os seus recursos (WOLFF, 2016).

2.3.3. Escalabilidade

Uma das principais vantagens deste modelo arquitetônico é a sua escalabilidade. Segundo Newman (2015), com uma grande aplicação monolítica, é necessário escalar tudo junto. Uma pequena parte do sistema fica limitada em desempenho devido à necessidade de escalar o sistema como uma nova instância da aplicação inteira. Com pequenos serviços, podemos escalar apenas os microsserviços que precisam de dimensionamento, o que nos permite executar outras partes do sistema com hardware menor e menos potente, como ilustrado na Figura 4.

Figura 4 - Escalando somente o necessário



Fonte: Newman (2015)

O capítulo 4 deste trabalho apresenta uma explicação mais detalhada sobre os diferentes métodos de escalabilidade, tanto da aplicação, quanto do armazenamento de dados.

2.3.4. Facilidade de *deploy*

No modelo tradicional de desenvolvimento de sistemas, múltiplos times trabalham em uma única base de código de uma grande aplicação monolítica. Quando um time faz uma mudança, não se pode realizar o *deploy* independente. A modificação deve então entrar na *branch* principal para realizar o *release*, e isso envolve coordenação entre os times, resolver conflitos de código, e depois disso, é necessário que a aplicação inteira seja retestada a fim de garantir que novos erros não foram inseridos (CHEN, 2018).

Segundo Dragoni et al. (2017), na arquitetura de microsserviços, cada serviço representa uma única capacidade de negócios que é entregue e atualizada independentemente. Achar um *bug* ou implementar pequenas melhorias não possui nenhum impacto nos demais serviços. Microsserviços é a primeira arquitetura desenvolvida após a era de entrega contínua

e, essencialmente, microsserviços são projetados para serem usados com entrega e integração contínua, tornando cada estágio da entrega automática. Através da utilização de sistemas de automação de *deploy* e ferramentas modernas de *containers*, é possível realizar o *deploy* de uma versão atualizada de um serviço para produção em questão de segundos, método o qual se prova benéfico em ambientes de negócios de mudança rápida.

De acordo com Limitee (2017), este modelo de arquitetura iniciou com o objetivo de realizar *deploy* de pequenas partes do sistema sem afetar o restante da aplicação. No entanto, isso evolui e começou a influenciar a maneira geral de como o software é arquitetado desde o início.

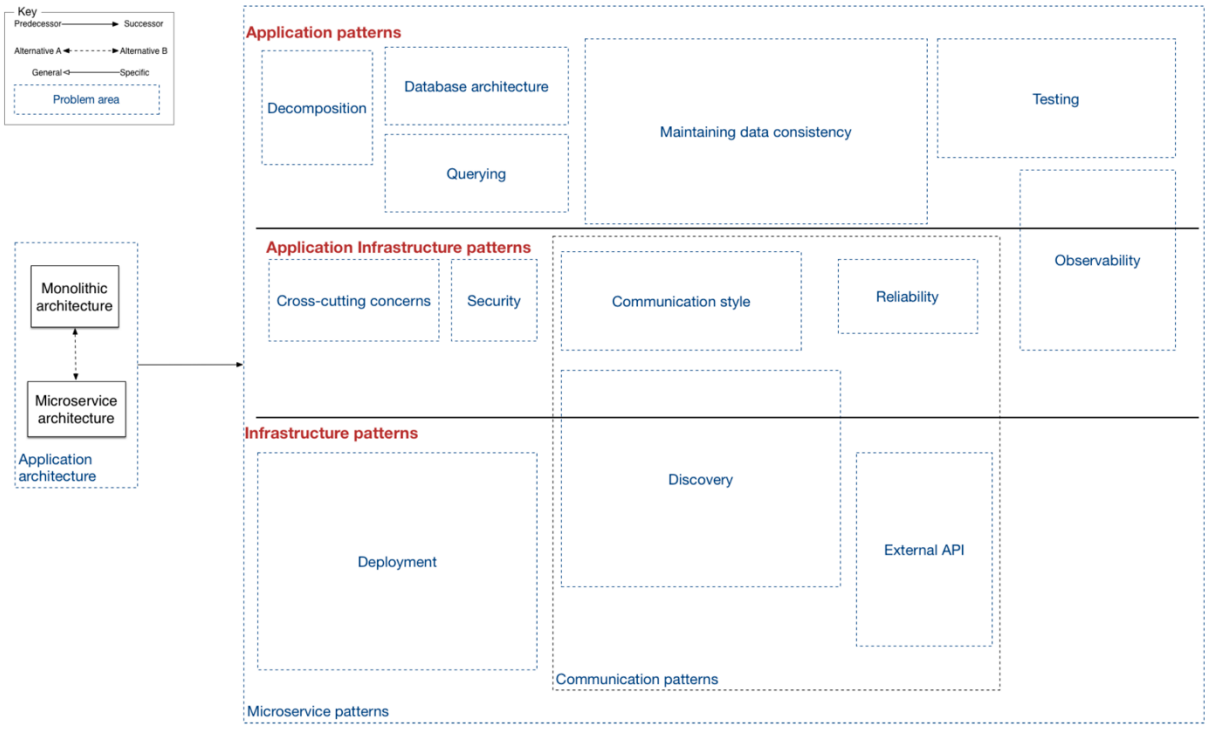
A componentização de microsserviços fornece a possibilidade de fazer mudanças em um componente e apenas realizar o *redeploy* do componente modificado, como o oposto frequentemente praticado em grandes aplicações monolíticas. Microsserviços encapsulam todos os recursos que eles precisam para funcionar. Portanto a arquitetura de microsserviços é um modelo mais flexível (LIMITEE, 2017).

3. PADRÕES ARQUITETURAIS DE MICROSSERVIÇOS

Em seu livro, Richardson (2018), mostra que para implementar uma arquitetura de microsserviços, diversas decisões arquiteturais devem ser tomadas, para isso, é proposto por ele uma linguagem de padrões, a qual descreve inúmeras abordagens para resolver problemas de *design* ou de arquitetura, através da definição de um conjunto de elementos de software colaborativos. Ele a descreve como uma coleção de padrões que ajudam a estruturar uma aplicação usando a arquitetura baseada em microsserviços.

A Figura 5 ilustra uma estrutura de alto nível sobre a linguagem de padrões proposta. Ela tende a ajudar na decisão se a arquitetura deve ser monolítica ou de microsserviços, em seguida, se a decisão é voltada para a baseada em microsserviços, a linguagem de padrões tem o intuito de ajudar a resolver diversos problemas de *design* oriundos deste modelo.

Figura 5 - Perspectiva de alto nível sobre a estrutura da linguagem de padrões



Fonte: Richardson (2018)

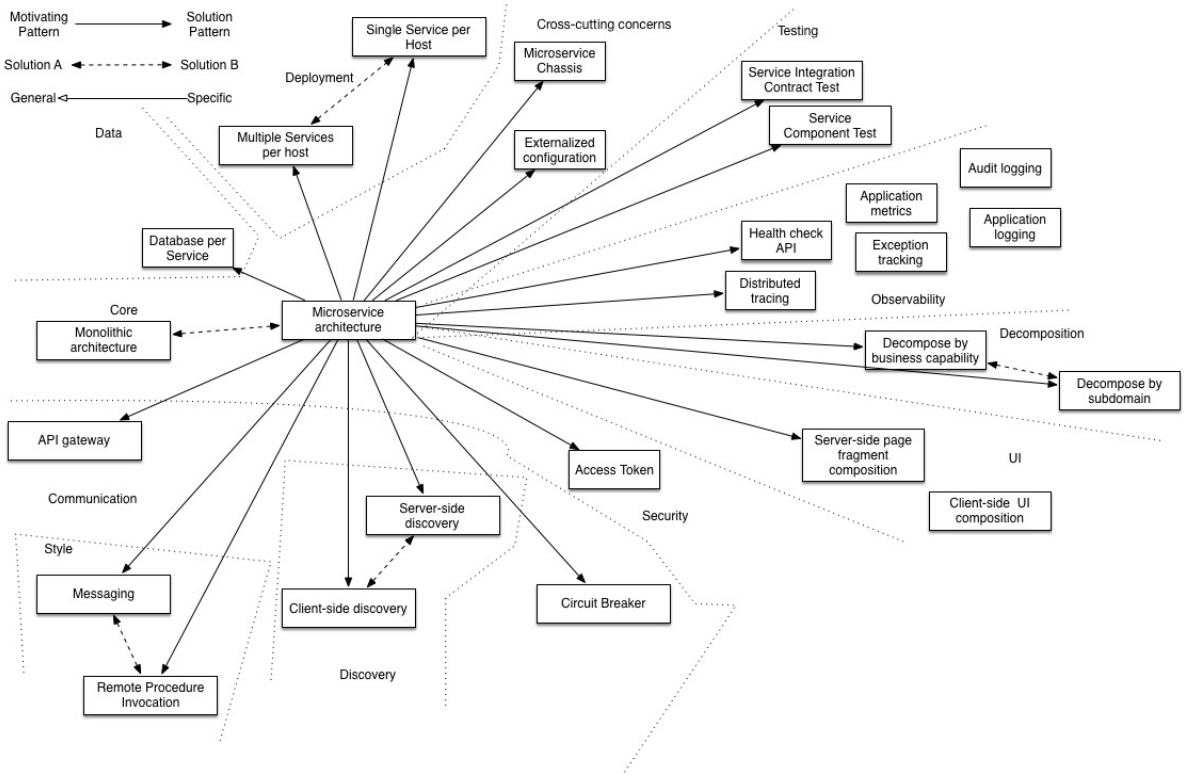
Na esquerda da Figura 5 são ilustrados os dois padrões de arquitetura, monolítica e microsserviços. O restante da imagem representa os grupos de padrões que são soluções para problemas oriundos da implementação do padrão arquitetural de microsserviços.

Segundo Richardson (2018), os padrões são agrupados baseados nos tipos de problemas que eles resolvem, podendo ser divididos em três camadas:

- Padrões de infraestrutura – Estes são padrões que resolvem problemas que são mais voltados para a infraestrutura e não estão relacionados com o desenvolvimento.
- Infraestrutura da aplicação – Estes são os padrões para problemas de infraestrutura que afetam o desenvolvimento.
- Padrões da aplicação – Estes são padrões que resolvem problemas relacionados com o desenvolvimento da aplicação.

A Figura 6 mostra os principais padrões arquiteturais de microsserviços dispostos dentro dos seus respectivos grupos.

Figura 6 - Padrões de microsserviços



Fonte: Richardson, 2018

3.1. Padrões de Comunicação entre Microsserviços

Segundo Brown e Woolf (2016), a arquitetura de microsserviços trabalha de forma a limitar estritamente a quantidade de opções para comunicação que um serviço pode

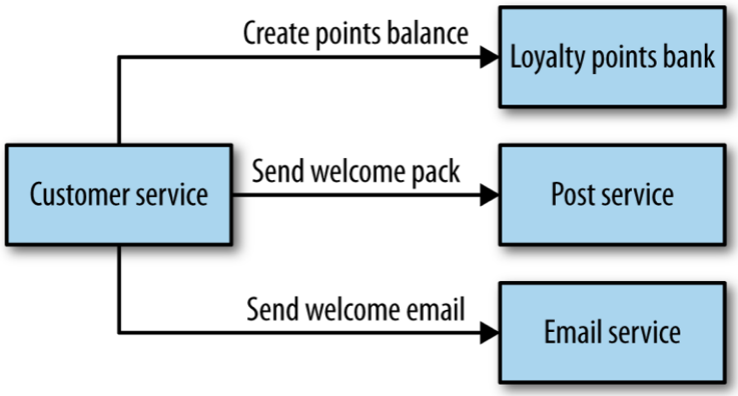
implementar, de forma a trazer maior simplicidade possível. Outra regra importante é evitar o acoplamento entre serviços causada pela comunicação implícita através do banco de dados – toda comunicação entre serviços deve ser feita através de uma API.

3.1.1. Orquestração e Coreografia

De acordo com Newman (2015), para modelar lógicas mais complexas, é necessário lidar com problemas como o gerenciamento de processos de negócios que vão além das fronteiras de um serviço individual. Com a arquitetura de microsserviços, esse limite é atingido mais cedo. Para implementar uma regra de negócio que necessite de outros serviços, existem dois estilos de arquiteturas que se pode seguir. Com orquestração, depende-se de um cérebro principal para guiar o processo, similar a um maestro de uma orquestra. Com coreografia, é informado a cada parte do sistema o seu trabalho, deixando que as mesmas cuidem dos detalhes, como dançarinas encontrando seu caminho e interagindo com os outros em um ballet.

O exemplo da Figura 7 demonstra o processo de criação de um usuário através de orquestração, onde o registro se inicia pelo serviço de clientes, que age como o cérebro principal. O serviço ficar responsável por interagir e acessar os demais serviços para o término do processo, podendo rastrear em que passo o processo se encontra.

Figura 7 - Criação de usuário via orquestração

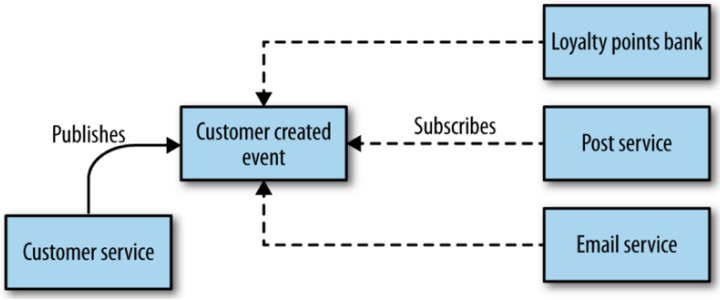


Fonte: Newman (2015)

Já ao utilizar a abordagem de coreografia, poderíamos ter somente o envio de um evento assíncrono pelo serviço Cliente, dizendo “Cliente criado”. Os demais serviços podem subscrever para este tipo de evento e reagir de acordo com o necessário, como ilustrado na Figura 8. Esta abordagem é mais desacoplada. Se outro serviço precisa ser incluso no processo

de criação de usuário, basta ele se subscrever para o evento e realizar o seu trabalho quando preciso (NEWMAN, 2015).

Figura 8 - Criação de usuário via coreografia



Fonte: Newman (2015)

3.1.2. Gateway de APIs

Microsserviços podem fornecer suas funcionalidades a outros serviços através de uma API. A criação de uma aplicação final baseada na composição de diferentes microsserviços requer um mecanismo de agregação. Um dos padrões frequentemente utilizados para isto é o API Gateway (TAIBI; LENARDUZZI; PAHL, 2018).

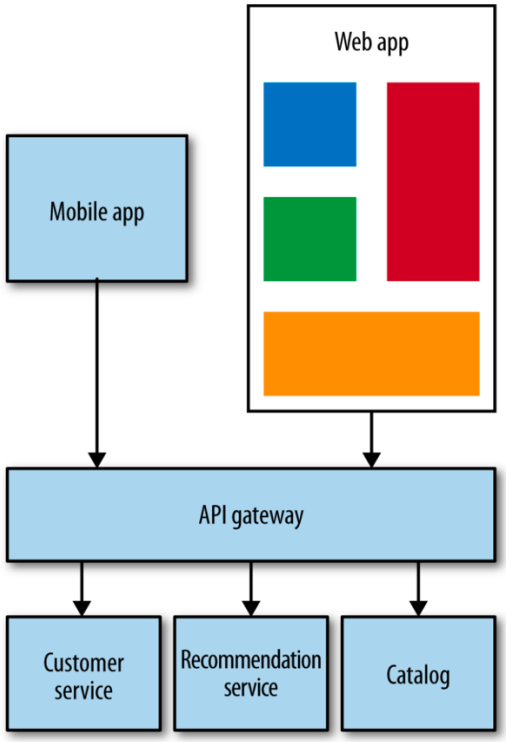
De acordo com Mozaffari (2015), a funcionalidade principal de uma aplicação é fornecida pelo conjunto de microsserviços, cada um tendo uma única responsabilidade. Um serviço então deve atuar como o API Gateway – definida por Richardson (2015) como um padrão para que um serviço haja como o único ponto de entrada de requisições para todos os tipos de clientes -, chamando os demais serviços individualmente e agregando a resposta de forma a ser consumida de maneira mais simples.

O *API-Gateway* fica entre os clientes e microsserviços, fornecendo uma API específica de acesso. Um dos objetivos é esconder a complexidade das tecnologias utilizadas (como por exemplo a conectividade com o *mainframe*) contra a complexidade de interface (DAYA et al., 2015).

Este padrão oferece uma interface mais simplificada para os clientes, tornando mais fácil de usar, entender e testar. Ainda reduz o número de requisições entre cliente e servidor, pois possibilita que clientes transformem múltiplas requisições em uma única requisição optimizada para um dado cliente, como um cliente *mobile* por exemplo. O benefício é que o dispositivo enfrentará a latência da rede uma única vez, usufruindo da infraestrutura interna do servidor para a comunicação entre o API Gateway e os demais serviços em uma rede com baixa latência e hardware mais poderoso (SANTIS et al., 2016).

A Figura 9 ilustra um API Gateway que recebe requisições de diferentes tipos de clientes, direcionando a requisição para os serviços que compõem a aplicação.

Figura 9 - API Gateway recebendo requisições



Fonte: Newman (2015)

Segundo Yöyen (2017), este serviço que age como API Gateway também é responsável por lidar com a autenticação de consumidores, autorização de acesso a recursos, e ainda age como um proxy para os demais serviços.

3.1.3. Descoberta de Serviço

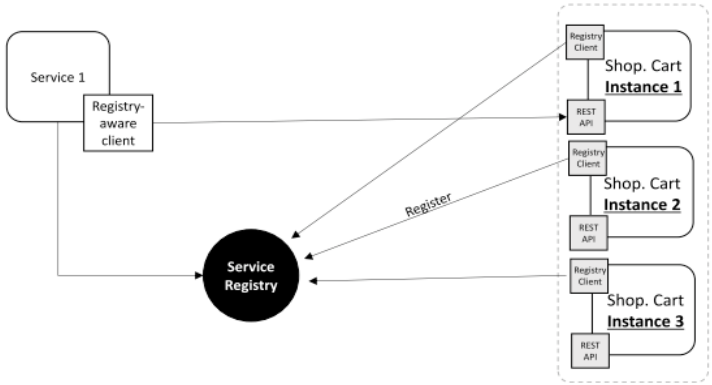
Ao criar uma aplicação com incontáveis serviços, é necessário rastrear onde estes componentes estão sendo executados para que seja possível se comunicar com eles. Estes serviços podem estar sendo executados em IPs ou portas temporárias, podem ter sido escalados, e podem até mesmo sido movidos para outros servidores desde a sua última execução. O padrão de descoberta de serviço ajuda a rastrear onde estes componentes estão sendo executados (SANTIS et al., 2016).

Segundo Santis et al. (2016), o padrão de descoberta de serviço é um registro que permite que um microsserviço se registre quando iniciado. Dessa maneira, o requisitante pode pesquisar sobre esse registro para descobrir quais instâncias de um serviço estão disponíveis antes de efetivamente chamá-lo.

A comunicação entre clientes e micro serviços deve ser dinâmica, para este propósito, o padrão de descoberta de serviço suporta a tradução de endereço DNS para endereços IP. São propostas duas implementações distintas, o *client-side* e o *server-side*. Na implementação do *client-side*, o cliente é responsável por escolher uma das instâncias do serviço através de uma busca prévia em um serviço de registro (Figura 10). Já no *server-side*, é adicionado um load-balancer como ponto de entrada da requisição, dessa meneira, o *load-balancer* fica responsável por realizar a busca no serviço de registro e possibilitando a comunicação direta entre cliente e a instância do micro serviço (Figura 11) (TAIBI; LENARDUZZI; PAHL, 2018).

A Figura 10 ilustra os serviços se registrando no módulo de registro assim que são iniciados, na sequência um serviço faz uma requisição para descobrir quais instâncias estão disponíveis e quais são seus endereços, de forma a acessá-los diretamente.

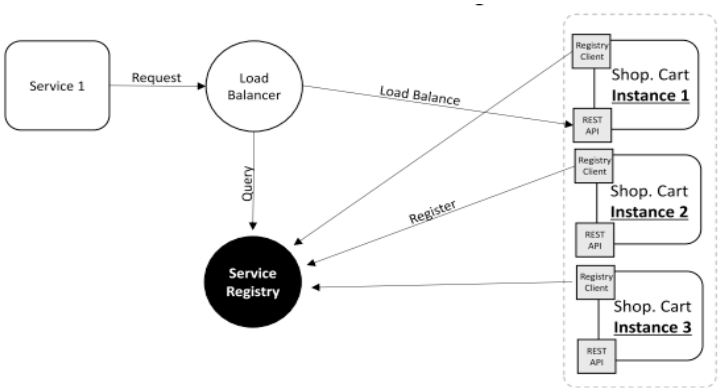
Figura 10 - Padrão de descoberta de serviços, *client-side*



Fonte: Taibi; Lenarduzzi; Pahl (2018)

O exemplo de arquitetura da Figura 11 demonstra o mesmo processo de registro dos serviços em um módulo de registro. A diferença dessa implementação está no módulo de balanceamento de carga que fica responsável pela busca em dos serviços disponíveis e posteriormente fornecendo o *endpoint* para o cliente.

Figura 11 - Padrão de descoberta de serviços, server-side



Fonte: Taibi; Lenarduzzi; Pahl (2018)

Mesmo assim, toda aplicação baseada em microsserviço deveria utilizar um sistema de descoberta de serviço. Ele forma a base de administração para um grande número de microsserviços e possibilita características adicionais com o balanceamento de carga. Para grandes aplicações, a descoberta de serviço é indispensável, pois o número de serviços aumenta com o passar do tempo, portanto, este padrão deve ser integrado na arquitetura logo no início do projeto (WOLFF, 2016).

3.2. Padrões de Armazenamento de Dados

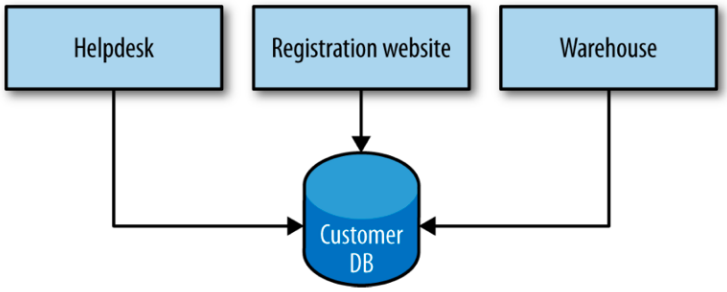
Um dos desafios de microsserviços é escolher o padrão mais adequado para o armazenamento de dados. Existem dois padrões predominantes de banco de dados, banco de dados por serviço e o banco de dados compartilhado (KALSKE, 2017).

As duas subseções a seguir apresentam detalhes sobre os padrões de bancos de dados utilizados na arquitetura de microsserviços.

3.2.1. Banco de dados compartilhado

No padrão do banco de dados compartilhado, que é mostrado na **Figura 12**, todos os microsserviços acessam um único repositório de dados, sem nenhuma abordagem para a isolação dos dados. Uma das principais vantagens deste modelo, é a simplicidade de migração de aplicações monolíticas, pois *schemas* podem ser reutilizados sem nenhuma mudança necessária, ou seja, a camada de acesso aos dados continua parecida, sem mudanças drásticas (TAIBI; LENARDUZZI; PAHL, 2018).

Figura 12 - Banco de dados compartilhado



Fonte: Newman, 2015

Ao utilizar um único banco de dados para toda a aplicação, é possível buscar, modificar e atualizar diversos registros em uma única transação, dando a segurança de ter a mesma perspectiva da informação todo o tempo (SANTIS et al., 2016).

Este modelo apresenta alguns problemas, um deles é a limitação de uma única tecnologia de banco de dados para toda a aplicação. Em alguns momentos, algumas informações seriam mais adequadas ficarem em um banco não relacional e outras em um banco relacional.

De acordo com Wolff (2016), a utilização de um banco de dados compartilhado é a violação de uma importante regra, a qual diz que os componentes devem poder mudar a representação interna dos dados sem afetar os demais serviços. Abaixo é elencado por ele os motivos pelos quais este padrão viola esta regra.

- A representação dos dados não pode ser facilmente modificada, pois várias aplicações acessam os dados. Uma mudança poderia fazer com que outras aplicações parem de funcionar. Isso significa que alterações devem ser coordenadas entre todas as aplicações.
- Se torna impossível fazer rápidas mudanças nas aplicações quando são exigidas mudanças no banco de dados.
- Com o avanço da complexidade da aplicação, se torna mais difícil enxugar o banco de dados – por exemplo, remover colunas que não são mais necessárias – porque não se sabe se outros serviços continuam a usar esta coluna.

Um dos princípios descritos por Fowler (2014) e referenciado a seguir é que se deve evitar a comunicação indireta entre microsserviços através de um banco de dados.

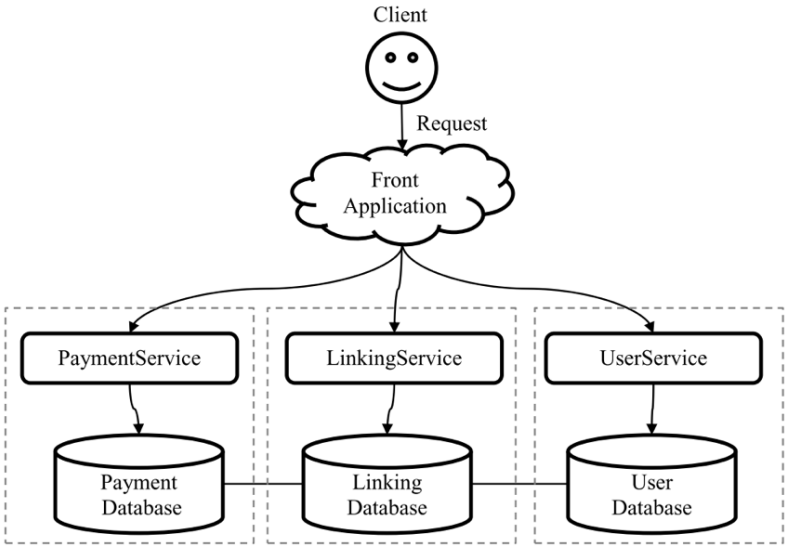
3.2.2. Banco de dados por serviço

No padrão de banco de dados por serviço, cada banco de dados é especificamente parte da implementação de um microsserviço isolado, como ilustrado na Figura 13. Ele não pode ser acessado diretamente por outros serviços. Com este conceito, é possível utilizar diferentes tecnologias de armazenamento de dados em diferentes serviços, uma abordagem chamada de por Fowler (2014) como Persistência Poliglota.

De acordo com Richardson (2018), ao utilizar um padrão de banco de dados isolado por serviço, os mesmos se mantem fracamente acoplados, sendo esta uma das caraterísticas chaves da arquitetura de microsserviços. Durante o desenvolvimento, o *schema* fica passível de modificação sem ter que ajustar os demais serviços que compõem a aplicação. Em *runtime*, um serviço vai estar isolado dos demais, garantindo que um serviço nunca vai ser bloqueado por outro que pode estar prendendo uma das tabelas através um *lock*.

Outro motivo pela utilização deste padrão, condiz com a capacidade de escalar o banco de dados na mesma proporção a qual os microsserviços escalam. Conforme Brown e Woolf (2016), para usar armazenamento escalável, a decisão é frequentemente tomada pela utilização de um banco NoSQL, ao invés de bancos relacionais. Isto se deve ao fato de que a representação nativa de dados que um microsserviço apresenta para o mundo externo é através de interface REST (JSON ou XML) e é mais facilmente armazenada neste tipo de banco.

Figura 13 - Banco de dados por serviço



Fonte: Risthein, 2015

4. ESCALABILIDADE

Em seu artigo, Agrawal et al (2011) explicam que com a experiência ganha nas últimas décadas com os líderes de tecnologias que fornecem serviços pela internet (Google, Amazon, etc.), percebe-se que a infraestrutura de aplicações no contexto da nuvem deve ser altamente confiável, disponível e escalável. Confiabilidade é um requisito chave para garantir acesso continuo a um serviço e é definido como a probabilidade de que dado sistema vai estar funcionando quando necessário como uma medida sobre um dado período de tempo. Parecido com isso, a disponibilidade é o percentual de vezes em que um sistema vai estar funcionando como requerido. Já o requisito escalabilidade é necessário devido à constante flutuação de carga que são comuns em serviços de contexto *web*. Esta variação pode ocorrer relacionada com um período de tempo, como diário, semanal ou mensal. Outra fonte de variação de carga está diretamente relacionada com o crescente número de usuários no serviço. Portanto, a escalabilidade surgiu ambos, como um requerimento crítico tanto como um desafio fundamental no contexto de computação em nuvem.

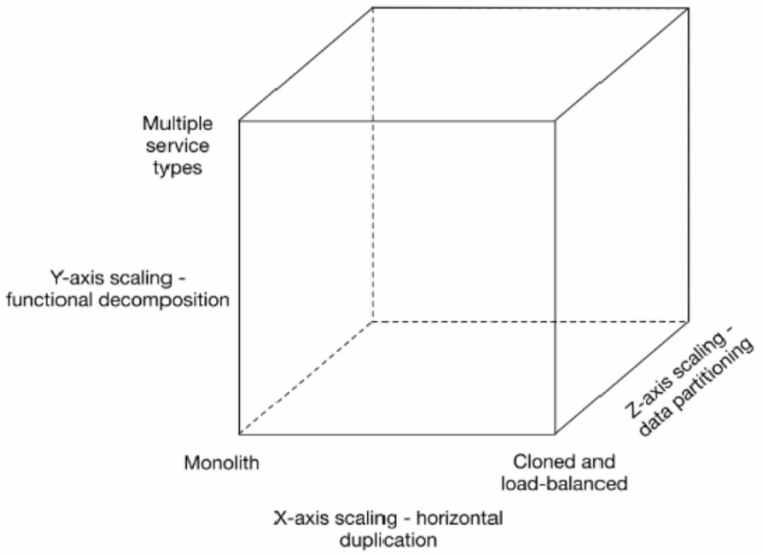
Em uma perspectiva de alto nível, a escalabilidade ajuda a melhorar a disponibilidade e a performance quando a demanda de acesso ao sistema está crescendo. Se os dados não estão disponíveis, a aplicação não pode ser executada. O acesso aos dados de forma contínua e performática é imprescindível para os sistemas atuais.

4.1. Scale Cube para Aplicações

Scale Cube é um modelo que foi introduzido por Abbott e Fisher (2009) e tem o objetivo de ajudar na tomada de decisão em como dividir e escalar serviços, dados e transações. Em suma, é uma maneira de fácil visualização dos três principais métodos de escalar uma aplicação. Seus princípios são explicados no decorrer do capítulo, utilizando o livro *The Art of Scalability* como referência.

O *Scale Cube* (Figura 14) consiste de três eixos: X, Y e Z. Cada eixo endereça uma abordagem diferente de escalar um serviço. O canto inferior esquerdo, representado pelas coordenadas X=0, Y=0 e Z=0, indica o pior caso, sendo um serviço monolítico onde todas funções existem dentro de uma única base de código em um único servidor, utilizando os recursos finitos como memória, velocidade de processamento, rede, armazenamento, etc.

Figura 14 - Scale Cube



Fonte: Namiot e Sneps-Sneppe (2014)

O eixo X do cubo representa a distribuição de carga entre múltiplas instâncias de uma mesma aplicação e um conjunto de dados. A abordagem é simples, consiste em clonar o sistema e serviço permitindo que os mesmos sejam executados em N servidores e cada um deve lidar com 1/N do total de requisições. Idealmente o método de distribuição é através de um *loadbalancer*.

O eixo Y do cubo representa a decomposição e segmentação por função, serviço ou recurso. A quebra de um serviço pode ser mais facilmente entendida através da utilização de ações por verbo, como por exemplo “Logar”, já a quebra de recursos pode ser identificada por substantivos, como “Informação de Conta”. Esse método não só ajuda na distribuição das requisições como no eixo X, mas também resulta em uma melhor distribuição de memória entre os serviços que compõem a aplicação.

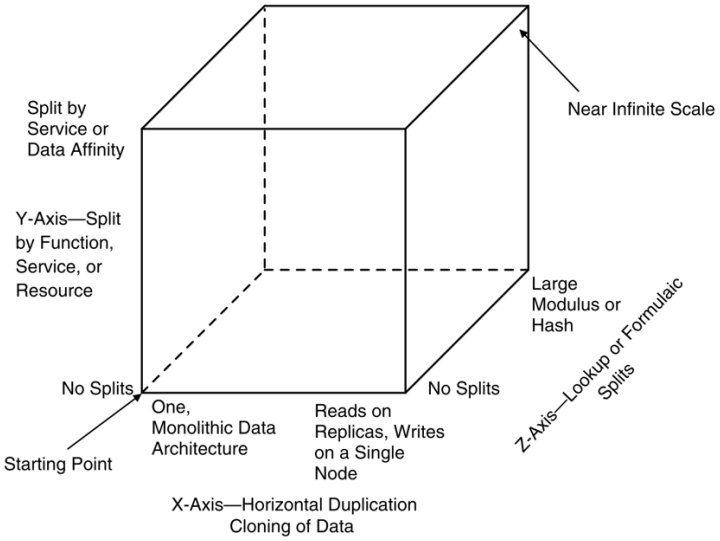
O eixo Z do cubo representa a distribuição e segmentação de trabalho por cliente, localização, necessidade do cliente ou alguma outra função de discriminação. Neste modelo em cada servidor será executado uma cópia da aplicação, similar ao eixo X. A grande diferença é que cada servidor será responsável por um único *subset* dos dados. Um componente ficará responsável pelo roteamento de cada requisição ao servidor adequado.

4.2. Scale Cube para Bancos de Dados

De acordo com Abbott e Fisher (2009), o conceito do *Scale Cube* não muda quando aplicado para bancos e outros sistemas de persistência de dados. Portanto, este capítulo tem o

objetivo de focar na nomenclatura e tema dos eixos tornando-os mais fáceis de usar como uma ferramenta para escalar o armazenamento dos dados.

Figura 15 - Scale Cube para Banco de Dados



Fonte: Abbott e Fisher (2009)

4.2.1. Eixo X do Scale Cube para Bancos de Dados

O eixo X do Scale Cube para banco de dados representa unicamente a replicação dos dados. Se tivermos uma camada de dados escalada usando somente o eixo X e consistindo de N sistemas, cada um dos N bancos de dados terá exatamente as mesmas informações, com pequenas diferenças resultantes do *delay* da replicação. Uma requisição pode ser atendida por qualquer um dos N bancos. Tipicamente escritas são feitas em um único nodo das cópias replicadas de dados, desta maneira reduzindo os conflitos de leitura e escrita entre os bancos e garantindo que um único nodo seja responsável pelas propriedades ACID do banco.

O escalamento através do eixo X é de fácil visualização e implementação. A grande maioria dos bancos possui tecnologia nativa para criação de nodos de replicação de dados, utilizando o conceito de *master* e *slave.*

A divisão pelo eixo X torna fácil o escalamento do banco com o aumento do número de transações ou requisições. Com o crescente número de requisições, basta adicionar novos nodos de leitura. Existe um limite para o número de sistemas que podem ser empregados, e esse limite normalmente nos levará a outros métodos de escala, pois o crescimento das transações continua a aumentar. É possível que esse limite não seja atingido devido ao aumento no atraso de tempo na replicação dos dados causado pelo grande número de nós adicionais de somente

leitura. Desta maneira, não podemos contar apenas com o escalamento baseado no eixo X, até mesmo em sistemas com crescimento relativamente baixo de dados se o crescimento de transações irá aumentar com o tempo.

Uma das principais desvantagens deste sistema é que os problemas oriundos do crescimento de dados não são resolvidos com a crescente quantidade de dados. Quando os dados aumentam, até mesmo em tabelas indexadas, o tempo de resposta tende a aumentar também.

4.2.2. Eixo Y do Scale Cube para Bancos de Dados

O eixo Y do *Scale Cube* para banco de dados representa uma separação de dados através do seu contexto ou significado dentro do *schema* ou do sistema de armazenamento. Quando separando os dados em conjuntos diferentes, isso normalmente representa o alinhamento de uma divisão em eixo Y já pré-determinada de uma aplicação com o contexto dos dados que são utilizados pelos serviços oriundos desta divisão. Em suma, a quebra pelo eixo Y consiste na separação dos dados em *schemas* que tem importância para os serviços específicos que realizarão operações nestes dados.

Uma abordagem que pode ser utilizada para a divisão dos dados é através de "substantivos". Esta abordagem pode gerar alguns benefícios em um primeiro momento, como a simplicidade de mapeamento através de afinidade dos dados e relações dos dados similares ao relacionamento presente em bancos de dados relacionais. A principal desvantagem neste cenário é que teremos que mudar a forma na qual a divisão dos serviços foi feita para a divisão por recursos, ou podemos ignorar e termos uma aplicação sem tolerância a falhas. Já ao realizar a divisão dos dados por recursos e dividir a aplicação por contexto de serviço, teremos serviços se comunicando com diversos bancos diferentes, isso significa que a disponibilidade do sistema será afetada, e isso é algo não desejado para as aplicações. Por estes e outros motivos, é fortemente recomendado a divisão para ambos, tanto para o armazenamento quanto para a aplicação, através da orientação a recursos ou orientada a serviços.

Este tipo de divisão geralmente é implementado para endereçar problemas associados com bancos que cresceram demais em complexidade e tamanho, e que possuem a tendência de continuar a aumentar. Este modelo ajuda a escalar a aplicação em termos de transações, pois a divisão ocorre em sistemas lógicos ou físicos diferentes, possibilitando que o número de transações aumente.

4.2.3. Eixo Z do Scale Cube para Bancos de Dados

O eixo Z do *Scale Cube* para banco de dados representa a separação dos dados através de atributos pesquisados ou determinados no momento da transação, com o objetivo de diminuir a carga de trabalho sobre determinado conjunto. Frequentemente são implementados como uma divisão por requisitante ou cliente, também podem ser divididos por catálogos de produto, ou determinado por um id de produto, ou qualquer outro tipo de divisão determinado em tempo de execução da requisição.

A divisão pelo eixo Z tem como objetivo, usufruir dos benefícios da quebra por eixo Y sem um bias para a ação (serviço) ou recurso em si. Ao fazer isso, este modelo tende a oferecer um balanceamento mais uniforme entre toda a informação do que a divisão sozinha através do eixo Y. Um exemplo deste modelo é a utilização de um algoritmo determinístico para armazenar e localizar a informação, sendo assim possível ter uma distribuição igualitária de demanda entre todos os sistemas de armazenamento de aplicação.

Esta divisão se assemelha, em termos de escalabilidade de transações, a divisão por eixo X, pois estamos dividindo os dados e como resultado as transações entre múltiplos sistemas. Porém não temos as restrições do eixo X, pois não estamos replicando nenhum dado. A utilização deste modelo de forma isolada possui diversos problemas, um deles é que estamos dividindo nossos dados em diversas unidades lógicas ou físicas, de modo que se tivermos uma falha de rede, software, ou hardware, uma parte da aplicação ficará completamente indisponível.

A divisão por eixo Z auxilia no crescimento do número de transações, na diminuição do tempo de processamento através da limitação dos dados necessários para executar qualquer transação, e por fim, no planejamento de capacidade através da distribuição da demanda mais uniforme entre os sistemas. Este tipo de divisão é mais efetivo em crescimento de usuários, clientes, requisições, ou outros elementos de dados que podem ser distribuídos uniformemente.

4.2.4. Juntando os eixos do Scale Cube

Por fim, é perceptível que a utilização de qualquer divisão por eixo de forma isolada possui diversas desvantagens. A recomendação é o planejamento de no mínimo dois eixos ou até mesmo três, mesmo que seja implementado somente um deles. Em um primeiro momento, uma boa abordagem é que se planeje a utilização da divisão por eixo Y ou Z em adição a implementação por eixo X.

Ao se deparar em uma situação de crescimento acelerado de dados, a solução pode ser implementada de forma rápida e simples, por exemplo ao dividir a aplicação por tipo de cliente (eixo Z), um divisão secundária através de funcionalidade (eixo Y), e para a redundância e crescimento das transações pode-se adotar a replicação dos dados através do eixo X. Com este modelo é possível dividir os clientes por raias através do eixo Z, encaixar cada raia deste eixo em funções específicas do eixo Y. O eixo X existe para a redundância e escalamento das transações. Ao utilizar a proposta descrita, é gearantido para o sistema alta disponibilidade e alta escalabilidade.

5. GERENCIAMENTO DE DADOS DISTRIBUÍDOS

Na arquitetura de microsserviços, cada um deles possui seu próprio armazenamento de dados privado. Diferentes microsserviços podem usar diferentes bancos de dados SQL e NoSQL. Enquanto esta arquitetura possui vantagens significativas, ela cria alguns desafios no gerenciamento de dados distribuídos. O primeiro desafio é como implementar transações de negócio que mantêm a consistência entre múltiplos serviços. O segundo desafio é como implementar buscas que acessam dados de múltiplos serviços (RICHARDSON, 2016).

Trabalhar com transações é muito mais fácil em um sistema monolítico, pois estes utilizam um único banco de dados. Microsserviços e múltiplas transações de dados trazem desafios complexos. Consistência eventual talvez tenha que ser utilizada ao invés de transações. Outras soluções são utilizar transações distribuídas ou compensação de transações (KALSKE, 2017).

No decorrer deste capítulo são abordados os principais conceitos envolvidos quando implementando sistemas que necessitem modificar e acessar dados em um serviço diferente do requisitante.

5.1. Paradigma BASE

Nos últimos anos, a quantidade de dados cresceu drasticamente, gerando a necessidade de implementar novas soluções que forneçam maior escalabilidade do que as soluções tradicionais baseadas em bancos de dados com propriedades ACID. Surgiu então novos princípios, resumidos sobre o paradigma BASE (*Basically Available, Soft-state, Eventual consistency*) (SIMON, 2012).

De acordo com Brewer (2012), ACID e BASE são duas filosofias de *design* em pontos opostos de um espectro de consistência e disponibilidade. Sendo que as propriedades ACID focam em consistência e são a abordagem tradicional para implementar bancos de dados. Já o termo BASE foi criado por ele e seus colegas no final dos anos 90 para suprir as demandas de design emergentes para alta disponibilidade e deixar explícito a escolha e o espectro.

O modelo BASE recentemente se popularizou devido à utilização de diversos sistemas com banco de dados NoSQL. Diferente de ACID, BASE oferece mais como um conjunto de diretrizes de programação do que um conjunto de propriedades especificadas rigorosamente e sua implementação leva em conta uma variedade de formas de aplicação. Dentre elas, está uma

abordagem para evitar transações distribuídas a fim de eliminar custos de performance e disponibilidade que estão associados com protocolos de *commit* distribuído (XIE et al., 2014).

O Quadro 1 demonstra uma comparação entre as principais características de cada um dos dois paradigmas, ACID e BASE.

Quadro 1 - Características ACID x BASE

ACID	BASE
Forte Consistência	Fraca consistência – *aceitável dados ausentes*
Isolamento	Disponibilidade em primeiro lugar
Prioridade para "commit"	Melhor desempenho
Transações aninhadas	Respostas aproximadas aceitáveis
Disponibilidade?	Agressivo
Conservador	Mais simples
Evolução complicada (*schemas*)	Mais rápido
	Evolução simplificada

Fonte: (BREWER, 2000)

5.2. Teorema CAP

O Teorema CAP é um conceito o qual resultou de uma análise sobre as consequências e as implicações da mudança de paradigma imposta pelo modelo BASE. Ela foi apresentada por Eric Brewer durante uma palestra no Simpósio sobre Princípios da Computação Distribuída no ano 2000. No decorrer dos anos o teorema CAP foi amadurecendo constantemente com desenvolvimentos e pequenos ajustes, na sua grande maioria por Brewer. Este teorema é um dos mais importantes achados para sistemas de armazenamento distribuído.

O teorema CAP foi introduzido como uma relação de compromisso entre *consistência*, *disponibilidade* e *partição tolerante a falhas*, os termos que compõem o teorema são definidos a seguir:

Consistência*:* O C do CAP é definido por Fox e Brewer ([s.d.]) como uma cópia única de dados, ou seja, todos os nodos devem ver a mesma informação no mesmo instante.

Disponibilidade*:* Dados são considerados altamente disponíveis se um dado consumidor das informações sempre tem acesso a uma replica. Disponibilidade de dados pode ser atingida através de redundância, como replicação de dados.

***Partição tolerante a falhas**:* Este termo significa que um sistema como todo pode continuar operante diante da partição entre réplicas de dados.

Segundo Simon (2012), o teorema CAP diz que só é possível ter no máximo duas dessas propriedades para qualquer sistema de dados compartilhados. Fox e Brewer ([s.d.]) descrevem as três combinações possíveis a seguir:

1 ***Perda da Partição tolerante a falhas (CA sem P)**:* Bancos de dados distribuídos só podem fornecer uma semântica distribuída na ausência de uma partição de rede separando os pares de servidores.

2 ***Perda da Disponibilidade (CP sem A)**:* Dados podem ser usados somente se a sua consistência é garantida. Isso implica em um *lock* pessimista, pois é necessário bloquear qualquer objeto que está sendo atualizado até que sua modificação seja propagada para todos os nodos. No caso de uma partição de rede, pode levar um certo tempo até que todos os bancos estejam em um estado consistente novamente, dessa maneira, não é possível fornecer uma alta disponibilidade.

3 ***Perda de Consistência (AP sem C)**:* No caso de uma partição, os dados ainda podem ser usados, mas como os nodos não podem se comunicar, não há garantia que os dados estejam consistentes. Isso implica na utilização de protocolos voltados para resolver inconsistências.

A perda de tolerância a partição não é factível em ambientes reais, pois sempre haverá partição de rede. Portanto a decisão deve ocorrer entre consistência e disponibilidade, as quais podem ser representadas por ACID (Consistência) e BASE (Disponibilidade) (SIMON, 2012).

5.3. Consistência Eventual

Consistência eventual é um modelo na computação distribuída que garante quando ocorrido um *update* em um registro na base de dados, eventualmente, em algum ponto no futuro, todo o acesso a essa informação em qualquer nodo, retornará o mesmo valor, ou seja, o banco de dados será consistente no momento em que a atualização for replicada para todos os nodos. Cada nodo pode atualizar sua própria cópia dos dados, se dois ou mais nodos atualizarem o mesmo registro, teremos um conflito de dados. Algoritmos de resolução de conflitos são necessários para alcançar a convergência – estado no qual todos os nodos do sistema tenham alcançado a consistência eventual -. Um exemplo deste tipo de algoritmo é a *ultima escrita*

vence, este modelo inclui um *timestamp* junto das atualizações a fim de identificar a última escrita e resolver os conflitos (YANAGA, 2017).

De acordo com Karwatka et al ([s.d.]) , consistência eventual não é uma técnica de programação, mas sim algo que deve ser pensado durante a fase de design do sistema. Este modelo de consistência está diretamente ligado com o Teorema CAP e é classificado como BASE, em contraste as tradicionais garantias ACID.

5.4. Arquitetura Orientada a Eventos

Nesta arquitetura, um microsserviço publica um evento quando algo acontece, como por exemplo o *update* em uma entidade de negócio. Outros microsserviços subscrevem para receber estes eventos. Quando um microsserviço recebe um evento, ele pode atualizar sua própria entidade, as quais podem levar a mais eventos sendo gerados (RICHARDSON, 2016).

Eventos podem ser utilizados para implementar transações de negócio que chama múltiplos serviços. Uma transação consiste de uma série de passos. Cada passo consiste de um microsserviço atualizando uma entidade e publicando um evento que dispara o próximo passo. É importante ressaltar que estas transações não são ACID, e por consequência oferecem uma garantia menor de consistência, como a consistência eventual. Este modelo de transações foi referenciado como o BASE, já abordado neste trabalho (RICHARDSON, 2016).

Este modelo pode servir para manter uma *view* de dados atualizada composta por informações distribuídas entre diversos serviços, como por exemplo, um serviço pode subscrever para receber eventos relacionados com pedido e cliente, desta forma ele pode manter uma view de dados com as informações atualizadas (RICHARDSON, 2016).

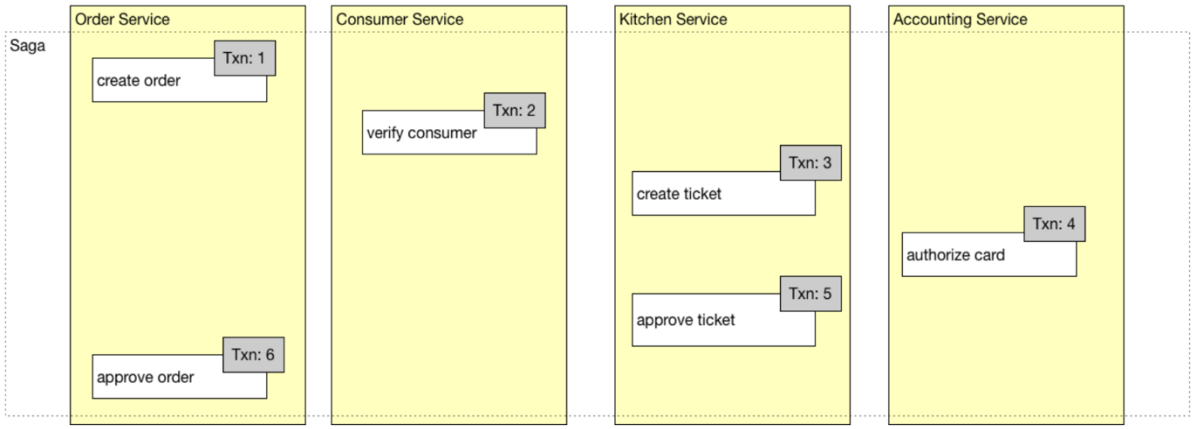
5.5. Padrão SAGA

O padrão SAGA foi proposto por Garcia-Molina e Salem (1987) como uma abordagem para solucionar problemas de transações muito longas em bancos de dados, as quais são denominadas pelo autor como *Long Lived Transactions (LLTs)*. Este tipo de transação é problemático pelo fato de segurar recursos do banco de dados por longos períodos de tempo, atrasando a conclusão de transações mais simples e frequentes. Uma LLT passa a ser uma saga se ela pode ser escrita como uma sequência de transações que podem ser intercaladas com outras transações. Um sistema de gerenciamento fica responsável por garantir que todas as transações em uma saga sejam terminadas com sucesso, ou transações de compensação são executadas para reverter uma execução parcial.

Segundo Richardson (2018), sagas são um mecanismo para manter os dados consistentes na arquitetura de microsserviços sem ter de usar transações distribuídas. Para a implementação, define-se uma saga para cada comando do sistema que é composto pela chamada de múltiplos serviços.

O exemplo ilustrado na Figura 16 mostra o Serviço de Pedido que implementa a operação criarPedido() utilizando a saga Criar Pedido. A primeira transação local é iniciada por uma requisição externa para criar um pedido. As outras cinco transações locais são disparadas pelo término da anterior.

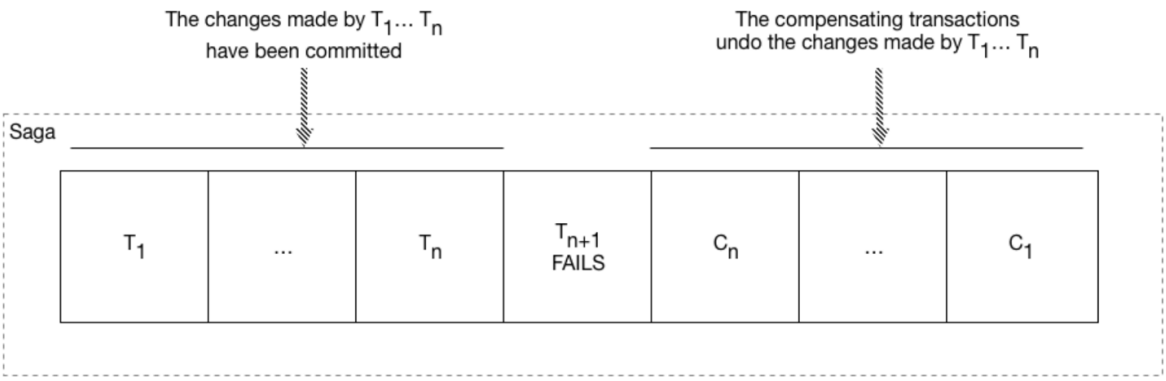
Figura 16 - Criando um pedido utilizando uma Saga



Fonte: Richardson, 2018

Uma característica importante de transações ACID tradicionais, é que uma lógica de negócio pode ser facilmente desfeita através do comando *ROLLBACK* caso seja detectado uma violação na regra, dessa maneira são desfeitas todas as mudanças que foram realizadas no banco de dados até o momento. Já em uma saga, se a transação (n + 1) falha, o efeito das outras n transações devem ser desfeitas. Conceitualmente, cada transação T terá uma transação de compensação C, a qual reverte os efeitos de T. Para reverter todos os n passos, a saga deve executar cada C na ordem reversa. Este exemplo é demonstrado na Figura 17, quando a transação falha, todas as transações de compensação devem ser executadas (RICHARDSON, 2018).

Figura 17 - Transações de compensação



Fonte: Richardson, 2018

Tomando o exemplo da Figura 16, esta saga pode falhar por uma variedade de motivos, como informações inválidas, ou o consumidor em questão não possui permissões para criar ordens. Se uma transação local falha, o mecanismo de coordenação da saga deve executar as transações de compensação para rejeitar o pedido e possivelmente o Ticket. O Quadro 2 mostra as transações de compensação para cada passo da saga Criar Pedido. Importante notar que nem todos os passos requerem transações de compensação, como operações de leitura por exemplo (RICHARDSON, 2018).

Quadro 2 - Transações de compensação para a Saga Criar Pedido

Passo	**Serviço**	**Transação**	**Transação de Compensação**
1	Serviços de Pedido	criarPedido()	rejeitarPedido()
2	Serviço Consumidor	verificarDetalhesConsumidor()	-
3	Serviço Cozinha	criarTicmet()	rejeitarTicket()
4	Serviço Contabilidade	autorizarCartãoDeCredito()	-
5	Serviço Cozinha	aprovarTicket()	-
6	Serviço Pedido	aprovarPedidoi()	-

Fonte: Richardson, 2018

De acordo com Richardson (2018), a implementação de uma saga consiste de uma lógica que coordena os passos da saga. A saga é iniciada por um comando do sistema, quando o primeiro passo termina sua execução, a lógica de coordenação deve chamar o próximo participante, o processo se repete até que todos os participantes sejam chamados, ou até haver falha em algum destes passos, neste caso a lógica de coordenação deve chamar as transações

de compensação, assim revertendo todas as mudanças realizadas pelos passos anteriores. Para estruturar uma saga em tal maneira, existem duas maneiras, Coreografia e Orquestração.

5.5.1. Coreografia e Orquestração em Sagas

O conceito de Coreografia e Orquestração foi explicado anteriormente no item 3.1.1 e se aplica de maneira similar, de forma que na coreografia, quando um serviço realiza um *update* em seu banco de dados, um evento é publicado para o próximo participante, neste evento deve haver toda a informação necessária para que o recipiente seja capaz de fazer o mapeamento das informações recebidas para a sua versão dos dados, como por exemplo, se um serviço de pedido recebe um evento de Cartão de Crédito Autorizado, o mesmo deve ser possível de mapear qual é o pedido relacionado com tal evento. Entre as principais vantagens de utilizar a coreografia, estão: simplicidade de implementação e o fraco acoplamento entre serviços. Já entre as desvantagens, a principal está relacionada com a dificuldade de compreensão, pois a saga acaba sendo uma composição de eventos distribuídas entre diversos serviços (RICHARDSON, 2018).

Já no modelo de Orquestração, deve ser implementado uma classe a qual sua única responsabilidade é dizer para os participantes da saga o que fazer. Esta comunicação se dá de maneira assíncrona, o orquestrador envia uma mensagem para um participante pedindo a execução, quando o participante termina sua execução, é enviada uma resposta para o orquestrador. O serviço deve utilizar mensagens transacionais afim de realizar atomicamente atualização do banco de dados e publicação de mensagens. Uma das principais vantagens deste modelo está no fato de a lógica de negócio ficar centralizada no orquestrador (RICHARDSON, 2018).

5.5.2. Deficiência em Isolamento

O I em ACID representa Isolamento. A propriedade isolamento em transações ACID garante que o resultado da execução de múltiplas transações em paralelo é o mesmo que a execução em série. O banco de dados cria a ilusão de que as transações ACID possuem acesso exclusivo aos dados. Isolamento torna mais fácil a escrita de lógicas de negócio que são executadas em paralelo (RICHARDSON, 2018).

Um dos maiores desafios da implementação de sagas é a falta de isolamento, portanto, sagas podem ser consideradas ACD. O motivo para esta falta de isolamento é que atualizações realizadas por transações locais das sagas, são visíveis a outras sagas no momento em que o *commit* local é realizado, isso pode trazer problemas de inconsistência pelo fato de uma saga modificar um dado já modificado por outra saga, mesmo sem o término da primeira. Estas

anomalias são percebidas principalmente pelo diferente resultado quando duas sagas ou mais são executadas em paralelo e em série. Para tais problemas, são propostos estratégias de design para reduzir ou até mesmo eliminar estas anomalias (RICHARDSON, 2018).

Segundo Richardson (2018), a falta de isolamento pode causar três tipos de anomalias:

***Updates perdidos* –** Uma saga sobrescreve um registro sem realizar a leitura de mudanças realizadas por outra saga.

***Leituras errôneas* –** Uma saga ou transação faz a leitura de mudanças feitas por outra saga que ainda não completou todos os updates.

***Leituras diferentes* –** Duas passos diferentes de uma mesma saga leem a mesma informação, mas com resultados diferentes porque outra saga modificou os dados entre as leituras.

É de responsabilidade do desenvolvedor escrever sagas de forma que o impacto destas anomalias seja minimizado ou até mesmo prevenidos. Lars e Zahle (1998) propõem em seu trabalho uma série de contramedidas para estas anomalias causadas pela falta de isolamento em múltiplos bancos de dados que não utilizam transações distribuídas, estas contramedidas são:

Lock semântico – Um *lock* a nível de aplicação

Update comutativo – Desenhar operações de *update* para serem executadas em qualquer ordem.

Perspectiva Pessimista – Reordenar os passos da saga para minimizar o risco de negócio.

Releitura – Prevenir escritas errôneas através da releitura prévia do dado para verificar se está no mesmo estado antes da escrita.

Versionamento de Arquivo – Gravar os *updates* em um registro para que que eles possam ser reordenados.

Por Valor – Utilizar o risco de negócio da requisição para selecionar o mecanismo de paralelismo.

5.6. Transações distribuídas

De acordo com Kalske (2017), transações distribuídas utilizam um gerenciador de transações para lidar com as transações. Transações distribuídas tentam executar transações como em aplicações monolíticas, mas com múltiplos bancos de dados através da rede. Tipicamente, para garantir a segurança das transações, é optado pela utilização do protocolo *Two-Phase-Commit*. Esta abordagem traz a desvantagem da necessidade de gerar *locks*, os

quais podem denegrir a escalabilidade do sistema. A escalabilidade é uma das principais vantagens da arquitetura de microsserviços, portanto a utilização de *Two-Phase-Commit* pode diminuir o ganho deste modelo arquitetural.

5.6.1. Two-Phase-Commit

Two-Phase-Commit é um algoritmo distribuído utilizado para garantir o término de uma transação em um ambiente distribuído. Assim, via 2PC, uma decisão unânime é alcançada e reforçada entre vários servidores participantes, seja para confirmar ou abortar uma determinada transação, garantindo assim a atomicidade. O protocolo procede em duas fases, ou seja, a fase de preparação (ou votação) e de confirmação (ou decisão), que explica o nome do protocolo (LECHTENB, 2009).

O protocolo é executado por um processo coordenador, enquanto os servidores participantes são chamados de participantes. Quando o iniciador da transação emite uma solicitação para confirmar a transação, o coordenador inicia a primeira fase do protocolo 2PC consultando - por meio de mensagens preparadas - todos os participantes, seja para anular ou confirmar a transação. Se todos os participantes votarem para se comprometerem, então, na segunda fase, o coordenador informará todos os participantes para que comprometam sua parcela da transação enviando uma mensagem de confirmação. Caso contrário, o coordenador instrui todos os participantes a abortar sua parte da transação, enviando uma mensagem de aborto. Entradas apropriadas de log são escritas pelo coordenador, bem como pelos participantes, para permitir a reinicialização de *procedures* em caso de falhas (LECHTENB, 2009).

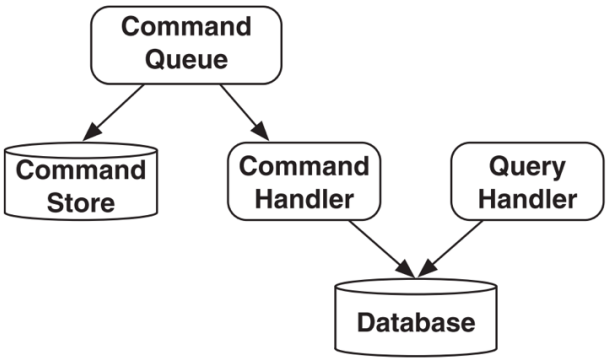
5.7. CQRS

Segundo Wolff (2016), as operações realizadas em bancos de dados podem modificar ou ler os dados. Estes dois tipos de operações podem ser separados, operações que modificam dados e, portanto, possuem efeitos (comandos) podem ser distinguidas de operações que somente leem dados (*queries)*. Também é possível estipular que uma operação não deve alterar simultaneamente o estado e retornar dados. Alguns benefícios são trazidos por esta abordagem, como por exemplo, buscas podem ser fornecidas por uma memória cache. Esta seperação entre comando e busca é chamada de CQS (*Commando Query Separation*).

CQRS (*Commando Query Responsibility Segregation*) é mais extremo que CQS e separa completamente o processamento de buscas e comandos. Neste modelo cada comando é

armazenado em um repositório de comandos. Em adição, pode-se utilizar gerenciadores de comandos. O gerenciador de comandos utiliza os comandos para armazenar o estado atual dos dados no banco de dados. Um gerenciador de buscas utiliza o banco para processar as buscas. O banco de dados pode ser ajustado para atender as necessidades do gerenciador de buscas, como por exemplo, pode-se utilizar um banco em memória, e se caso haver uma falha, os dados podem ser reconstruídos a partir das informações que foram persistidas no repositório de comandos. A Figura 18 exemplifica a definição de CQRS (WOLFF, 2016).

Figura 18 - CQRS



Fonte: Wolff (2016)

No modelo de CQRS, o conceito visa a separação das responsabilidades, por exemplo, uma página web é construída baseada nas informações retornadas por um modelo query, quando o usuário realizar alguma modificação, esta modificação é encaminhada para o modelo de comandos. Quando a modificação termina, as mudanças são comunicadas para o modelo de query para a atualização dos dados de visualização (FOWLER, 2011).

De acordo com Wolff (2016), um dos principais desafios do *design pattern* CQRS é que as transações que possuem operações de leitura e escrita acabam sendo difíceis de implementar, ou seja operações de escrita e leitura talvez tenham que ser implementadas em serviços diferentes. Outro problema é o que tange a consistência de dados através de diferentes sistemas, pois o processamento de eventos é assíncrono, o que implica no término das operações em pontos diferentes no tempo. Este *design pattern* pode ser considerado consistente eventualmente, pois o modelo de busca pode estar fora de sincronia com o modelo de escrita por determinado tempo.

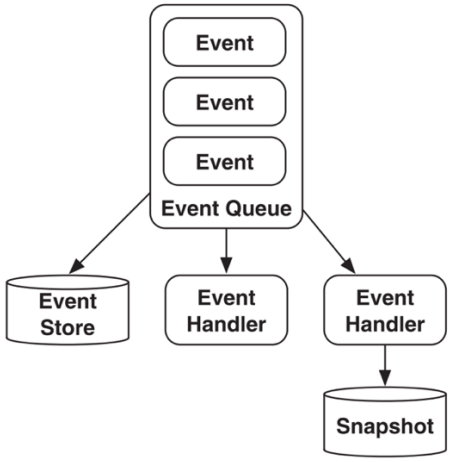
5.8. Event Sourcing

Dividir as informações pode fazer o gerenciamento de dados ainda mais complicado, os bancos individuais podem ficar facilmente dessincronizados ou até mesmo inconsistentes. É necessário adicionar ferramentas para gerenciamento de informações mestres. Enquanto é executada em background, ela deve eventualmente encontrar e corrigir inconsistências. Um dos padrões para tal sincronização é o Event Sourcing. Este padrão tem por finalidade ajudar em tais situações através do fornecimento de um histórico de log confiável de todas as mudanças que podem ser revertidas ou realizadas (KARWATKA et al., [s.d.]).

De acordo com Wolff (2016), ao invés de armazenar o estado dos dados, *Event Sourcing* armazena os eventos que levaram ao estado atual. Enquanto o estado em si não é salvo, ele pode ser reconstruído a partir dos eventos. Eventos não podem ser modificados, ou seja, eventos errôneos devem ser corrigidos por novos eventos. A Figura 19 demonstra de forma global o *design pattern Event Sourcing*, seus componentes são descritos a seguir.

- A fila de evento envia todos os eventos para diferentes recipientes. E pode, por exemplo, ser implementado com um middleware de mensagens.
- O banco de eventos armazena todos os eventos, tornando possível a reconstrução da cadeia de eventos e os eventos em si.
- O event handler reage aos eventos. Ele pode conter a lógica de negócio.
- Não é fácil de acompanhar o estado do sistema, para isso, pode-se optar pela utilização de um snapshot. Em cada evento, ou após determinado período de tempo, os dados do snapshot serão atualizados com os novos eventos gerados.

Figura 19 - Visão geral sobre Event Sourcing



Fonte: Wolff (2016)

6. VALIDAÇÕES PROPOSTAS

De acordo com a revisão bibliográfica realizada, é perceptível a não adoção por transações distribuídas, como por exemplo *Two-Phase-Commit*, este tipo de abordagem afeta diretamente dois pontos importantes do modelo arquitetural de microsserviços, principalmente pelo fato da necessidade de aplicar *lock* nas tabelas. O primeiro ponto afetado é a escalabilidade, já o segundo ponto, de acordo com a análise sobre o Teorema CAP, é a disponibilidade, pois o *lock* impede que dados sejam acessados por outros serviços durante uma transação corrente. Outro problema decorrente de transações distribuídas é que muitos bancos de dados NoSQL, como MongoDB e Cassandra, não suportam tais protocolos. Para resolver problemas mais complexos e manter a base de dados consistente em uma arquitetura de microsserviços, a aplicação pode utilizar diversos mecanismos diferentes sobre o conceito de fraco acoplamento e serviços assíncronos. Para atender tais demandas, aplicações distribuídas podem utilizar diferentes padrões, dentre eles está o padrão SAGA e o *Event Sourcing*. Ambos possibilitam a execução de transações em sistemas distribuídos, com possibilidade de *rollback* em caso de falha, porém o mesmo deve ser desenvolvido semanticamente.

Este capítulo detalha a arquitetura e a implementação da aplicação utilizada como foco de testes no presente trabalho. Desta maneira é possível comparar o desempenho das execuções de cada uma utilizando o modelo SAGA Orquestrada e o *Event Sourcing* com CQRS.

O modelo de aplicação proposto para a implementação e validação é baseado em uma aplicação exemplo, utilizada pelo framework Eventuate de Chris Richardson. O cenário proposto é de que o usuário possa criar um pedido, através de uma chamada REST para um *endpoint* dedicado do microsserviço de Pedido. Na requisição, devem ser fornecidas informações em formato JSON sobre o *id* do usuário, preço total do pedido, *id* do produto e quantidade, esta requisição está ilustrada na Figura 27. Toda chamada para criar um pedido é assíncrona nas duas implementações, ou seja, toda requisição retorna uma resposta imediata, e todas as interações necessárias para a finalização da transação se dão pelo canal de mensageria entre os serviços, que é implementada de forma diferente nas duas aplicações.

6.1. ARQUITETURA E FRAMEWORKS

Os dois primeiros subcapítulos desta seção apresentam os *frameworks* utilizados neste estudo para o desenvolvimento da aplicação Java. Os *frameworks* explorados são Eventuate Tram e Eventuate ES. Os demais capítulos explicam a arquitetura utilizada com estes

frameworks dentro dos modelos propostos, Sagas por orquestração e *Event Sourcing* com CQRS

6.1.1. Eventuate ES

Eventuate é uma plataforma para desenvolvimento de microsserviços assíncronos (Richardson, 2018b). Ela foca no gerenciamento de dados distribuídos, possibilitando que os desenvolvedores foquem na implementação das regras de negócio. A plataforma consiste de dois produtos, Eventuate ES e Eventuate Tram. Esta seção apresenta detalhes de Eventuate ES, mais detalhes sobre Eventuate Tram estão descritos na seção 6.1.2.

Eventuate Event Sourcing (ES) possibilita o desenvolvimento de aplicações com o modelo de programação baseado em *Event Sourcing* – um mecanismo que rastreia todas as modificações em um modelo de dados como uma sequência de eventos armazenados em log de eventos. Toda modificação no modelo de dados é adicionada ao *log*, e pode ser reproduzida a qualquer momento para restaurar o estado da aplicação. Entre outras vantagens, está o fato de que ES serve como um *log* de auditoria, pois é possível identificar todos os eventos que levaram um estado x final. Outro ponto importante é o a possibilidade de os eventos serem consumidos mais quando um serviço falho reconecta.

Este *framework* está disponível em dois modelos, como serviço hospedado na *Amazon Web Services* (AWS) e como um projeto *open source* que pode ser executado em demais ambientes, este denominado Eventuate Local. Para fins deste estudo foi optado pela utilização da versão *open source*.

6.1.2. Eventuate Tram

Eventuate Tram possibilita que uma aplicação Java/Spring envie mensagens assíncronas como parte uma transação de um banco de dados (EVENTUATE, [s.d.]). Ela utiliza a conectividade tradicional do Java com o banco de dados (JDBC) e o Java Persistence API (JPA) para fornecer mensagens transacionais. Isso possibilita que um microsserviço atualize seu estado e publique informações de forma atômica, como uma mensagem ou um evento para outros serviços. O principal objetivo é manter a consistência de dados dentro de uma arquitetura de microsserviços.

Este *framework* fornece diversas maneiras de abstração de comunicação, como Mensagens – enviar e receber mensagens através de canais previamente nomeados -, Eventos

– publicar eventos de um domínio e subscrever a eventos de outros domínios -, Comandos – enviar um comando assíncrono para outro serviço e receber uma resposta.

Eventuate Tram Sagas é um framework que possibilita o processamento de Sagas através de um orquestrador. Ao invés de depender que serviços respondam a eventos de domínio, com este framework é possível definir um orquestrador para a saga, o qual diz aos participantes quais operações devem executar.

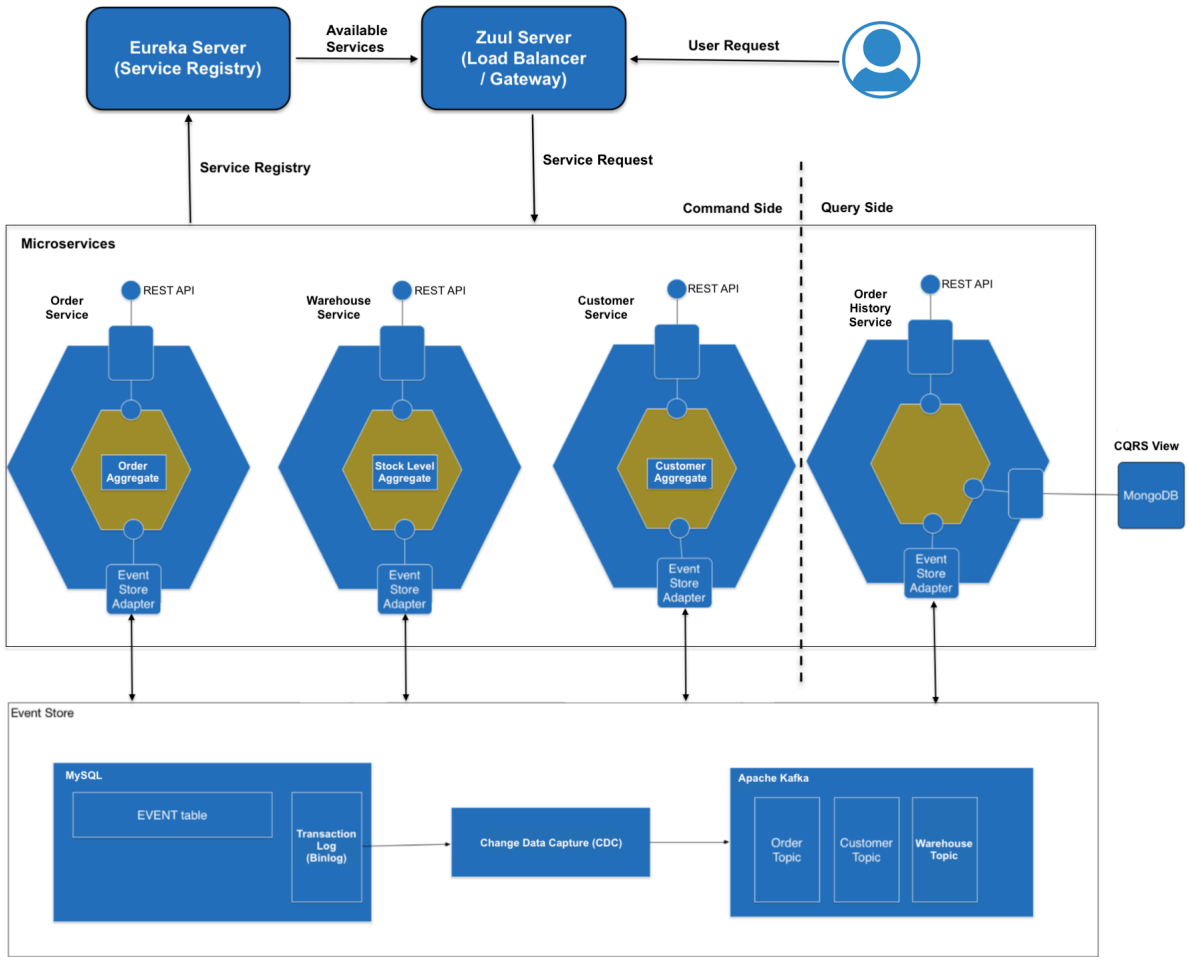
6.1.3. Implementação 1 - Event Sourcing e CQRS (Eventuate ES)

Esta implementação é baseada em *Event Sourcing* e no padrão CQRS, o qual é descrito na seção 5.7. Neste modelo os comandos de negócio são executados diretamente nas entidades as quais correspondem ao respectivo microsserviço. A comunicação é um resultado da troca de eventos, os quais são gerados por outros serviços durante o processamento de um comando ou o processamento de outro evento. Esta seção tem por objetivo explicar os componentes envolvidos na implementação da transação de criação de ordens utilizando o *Event Sourcing*, como meio de gerenciar e reagir aos eventos de cada microsserviço.

Event Sourcing é uma ótima maneira de atualizar o estado e publicar um evento atomicamente, este modelo normalmente é implementado em conjunto com a segregação entre comandos e leituras (CQRS). Além disso, um mecanismo de leitura desacoplado pode subscrever a eventos de diversos domínios diferentes, a fim de fornecer um ponto de leitura central e estruturado. Esta implementação foi desenvolvida utilizando o framework *Eventuate ES*, com a versão Local, isso implica na utilização do banco de dados MySQL para a persistência dos eventos, e o Apache Kafka como plataforma de distribuição de eventos. Outros serviços são necessários e fornecidos pela plataforma, sendo estes o Apache Zookeeper e o *Change Data Capture* (CDC). A arquitetura está ilustrada na Figura 20 e mostra todos os componentes utilizados. Seus respectivos comportamentos estão descritos em mais detalhes nos itens abaixo.

Nesta arquitetura foram implementados os módulos de *Service Registry* e *Zuul Server* para atender as requisições do usuário. Customizações na geração de *ID* da plataforma *Eventuate* foram necessárias a fim de possibilitar a execução dos testes planejados no ambiente local.

Figura 20 - Arquitetura e-commerce utilizando *Event Sourcing* e CQRS



Fonte: do Autor, 2019

- **Microsserviços:** Uma descrição completa do comportamento dos microsserviços que fazem parte desta implementação pode ser encontrada na seção 6.1.4.
- **Banco de dados MySQL**: Este é o banco de dados utilizado por esta aplicação para a persistência de eventos, atualmente é o único banco de dados compatível com o framework Eventuate Local. A instância deste banco é responsável pelo gerenciamento de três tabelas, estas são *EVENTS*, *ENTITIES* e *SNAPSHOTS*. O MySQL ainda possui o papel de fornecer um log de transações o qual é consumido pelo CDC, como uma base o *Event Sourcing*.
- **Change Data Capture (CDC)**: Este serviço é responsável por monitorar o arquivo de log (*Binlog*) gerado pelo MySQL para identificar a inserção de novos eventos na tabela EVENTS, quando identificado novos eventos, o CDC é responsável por publicar estes

eventos no Kafka com tópico que corresponde o aggregate ao qual o evento foi direcionado.

- **Apache Zookeeper**: Apache Zookeeper é um projeto open-source que possibilita a coordenação distribuída confiável (FOUNDATION, 2019a). Ele é responsável por manter um serviço centralizado para gerenciar funcionalidades como configurações, nomeações e fornecer sincronização distribuída. Apesar de não representado na arquitetura, este componente é uma dependência para o Apache Kafka.
- **Apache Kafka**: Apache Kafka é um serviço de *streaming* o qual é responsável pela administração do mecanismo de subscrição e publicação de eventos para os microsserviços. Kafka é baseado no Streams API, o que possibilita aplicações reagirem a eventos em tempo real (FOUNDATION, 2019b).
- **Order History Service:** Este serviço é desenvolvido utilizado o mesmo framework dos demais, a diferença é que ele serve somente como leitura, reagindo aos eventos dos demais microsserviços, porém somente para gerar uma coleção de dados os quais serão utilizados para visualização dos dados, este serviço representa o *Query Side* do modelo CQRS.
- **MongoDB**: MongoDB é um banco de dados NoSQL voltado para armazenamento de documentos, desenhado para facilitar o desenvolvimento e a escalabilidade (INC, 2019). Ele é utilizado nesta aplicação como banco de dados do serviço Order History Service, para armazenar os dados materializados gerados pelos demais serviços.
- **Eureka Server**: Eureka é um serviço baseado em REST utilizado para registrar serviços a fim de localização, balanceamento de carga e controle de falha (NETFLIX, 2019a). Nesta aplicação ele atua principalmente como Service Registry, ou seja, todo serviço de negócio, ao ser executado, se registra no Eureka Server utilizando o Eureka Client, e posteriormente outros serviços, como o Zuul, podem usufruir dessas informaçãoes para fazer o roteamento de requisições dos clientes e também o balanceamento de carga. Uma descrição mais completa sobre o funcionamento de Service Registry está disponível na seção 3.1.3.
- **Zuul Server**: Zuul é o ponto de entrada para todas as requisições, ele foi construído para possibilitar um roteamento dinâmico, monitoramento, resiliência e segurança (NETFLIX, 2019b). Zuul é utilizado nesta implementação como roteamento e balanceamento de carga, ou seja, ele consome a lista de serviços disponíveis no Eureka

e utiliza *Round Robbin* para balancear as requisições, a fim de não sobre carregar sempre a mesma instância.

Em suma, nesta arquitetura o ponto de entrada é uma requisição do usuário ao Zuul Server, o qual age como um roteador que encaminha a solicitação para o microsserviço de destino. A requisição é então processada como um comando, como por exemplo a criação de um pedido, este comando gera como resultado um evento que é persistido na tabela de eventos e também gera um retorno para o usuário, terminando a requisição. Em um segundo momento o módulo CDC consome os últimos eventos adicionados na tabela *EVENTS,* através da leitura do arquivo de log de transação do MySQL (Binlog), estes eventos consumidos são publicados no Apache Kafka, tornando possível que os demais microsserviços que se subscrevem a determinado tópico possam reagir e continuar executando a regra de negócio, o processo é repetido até que um evento final seja produzido, ou seja, um evento o qual nenhum outro microsserviço reage.

6.1.4. Microsserviços Participantes

Esta subseção tem por objetivo descrever o comportamento de cada um dos microsserviços que compõem esta implementação. A mesma é composta por 3 serviços de negócio implementadas utilizando Java Spring Boot, sendo elas, Order Service, Warehouse Service e Customer Service. Um quarto serviço é necessário para visualizar as informações geradas pelos demais, mais detalhes nos itens a seguir.

- **Order Service**: Este serviço implementa um *endpoint* REST para gerenciar pedidos. Salva um pedido como um evento no banco de dados MySQL. Utilizando a plataforma Eventuate ES, ela é responsável pela inserção de eventos que serão consumidos posteriormente pelo Warehouse Service e Customer Service.
- **Warehouse Service**: Este serviço implementa um *endpoint* REST para gerenciar o nível de estoque dos produtos. É responsável por atualizar os níveis de estoques e reservar produtos em forma de eventos, estes serão consumidos pelo serviço de pedido.
- **Customer Service**: Este serviço implementa um *endpoint* REST para gerenciar usuários. Atualiza informações dos usuários no banco de dados em forma de evento, os quais quando publicados pelo CDC serão consumidos pelo serviço de pedido.

- **Order History Service**: Este serviço implementa um *endpoint* REST para visualização das ordens, não emite nenhum evento, e somente subscreve aos eventos que são relevantes para ele. O seu objetivo e fornecer uma *view* através da agregação de eventos gerados pelos de mais serviços. O resultado do processamento dos eventos é armazenado como estado final no banco de dados NoSQL MongoDB. Este serviço representa o *Query Side* do padrão CQRS.

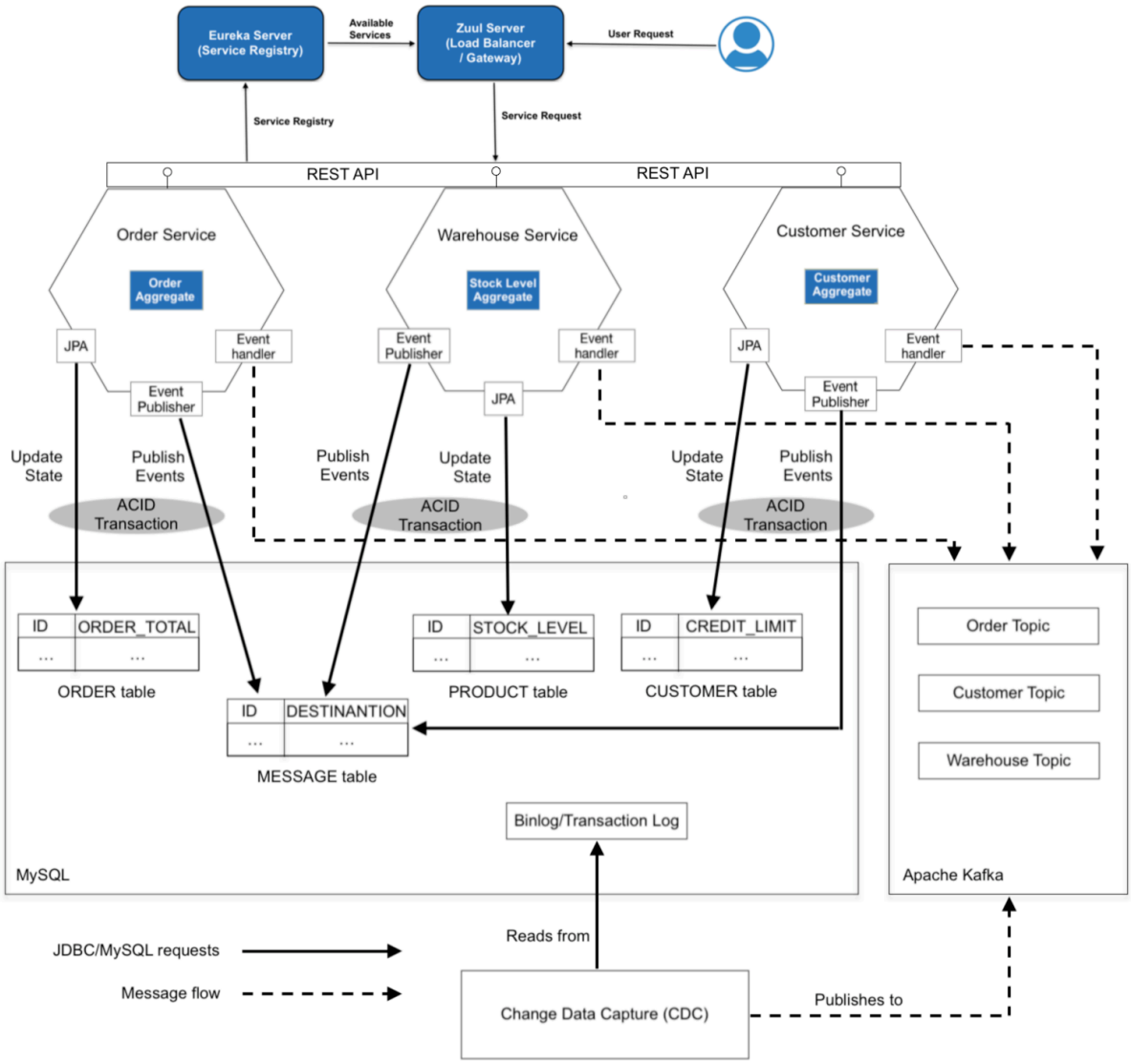
6.1.5. Implementação 2 - Saga Orquestrada (Eventuate Tram)

Como descrito na seção 5.5.1, o desenvolvimento de uma regra de negócio utilizando sagas por orquestração, torna mais fácil a leitura do código, devido ao fato de que toda a lógica está em um único ponto de referência centralizado, este padrão foi adotado como o segundo caso de estudo. Esta seção apresenta em detalhes todos os componentes da arquitetura desenvolvida para implementar a regra de postagem de pedido utilizando o padrão Saga. Esta implementação é voltada para a utilização do framework Eventuate Tram descrito na seção 6.1.2.

Parecido com a implementação baseada em *Event Sourcing*, o desenvolvimento da mesma aplicação utilizando um orquestrador de saga com o framework Eventuate Tram possui uma distribuição similar ao Eventuate ES, sendo necessário alguns dos mesmos módulos, como o Apache Zookeeper, Apache Kafka, banco de dados MySQL e o componente CDC. A descrição de cada um destes módulos está presente na seção anterior, os demais módulos podem ser identificados na Figura 21 e são explicados a seguir.

Igualmente na arquitetura descrita na seção anterior, foram implementados os módulos de *Load Balancer* e *Service Registry* utilizando o Zuul Server e Eureka Server respectivamente. Nesta implementação utilizando o padrão Saga, foi adicionado o serviço *Warehouse Service*, o qual é responsável pelo gerenciamento de produtos e estoque.

Figura 21 - Arquitetura e-commerce utilizando Saga Orquestrada



Fonte: do Autor, 2019

- **Microsserviços**: As funções dos serviços que fazem parte desta implementação estão descritas na seção 6.1.6.
- **Change Data Capture (CDC):** O comportamento deste módulo é exatamente o mesmo descrito da seção anterior, porém nesta implementação ele consome a mensagens da tabela *MESSAGES*, a qual possui uma estrutura ligeiramente diferente, pois é voltada para a comunicação de sagas, ou seja, envio de comandos e respostas.
- **Banco de Dados MySQL:** Nesta implementação cada microsserviço possui sua tabela no banco de dados e somente o estado final de cada entidade é persistido, assim como nas aplicações tradicionais.

Na mesma maneira que a implementação anterior, o ponto de entrada nesta é a partir de uma requisição ao Zuul Server, sendo posteriormente encaminhada ao microsserviço de destino. O microsserviço processa a requisição que pode resultar no início de uma saga, esta saga é criada e persistida em uma tabela a parte no banco de dados e é responsável então por enviar comandos para os participantes e receber a resposta, estas mensagens são trocadas através da tabela *MESSAGES* que é posteriormente consumida pelo componente CDC com o mesmo comportamento da implementação descrita na seção anterior.

Quanto a definição do coordenador da saga, ela pode ser um serviço a parte ou então ser parte de um outro microsserviço, como nesta implementação, onde o coordenador está inserido dentro do contexto do Order Service, definida pela classe CreateOrderSaga. Esta definição utiliza uma abordagem declarativa com uma API fluída para declarar a saga em passos de execução. Cada passo declara um método responsável pela manipulação que vai ser chamado quando o passo for iniciado, também é possível fornecer um passo de compensação caso a transação executada por um participante resulte em uma falha. A saga utilizada por esta implementação com os respectivos passos de execução e compensações está ilustrada na Figura 22, deve-se sempre definir um passo de compensação antes da definição de um passo de execução.

Figura 22 - Definição de saga utilizando Eventuate Tram

```
private SagaDefinition<CreateOrderSagaData> sagaDefinition =
        step()
                .withCompensation(this::rejectOrderForNoStock)
                .step()
                .invokeParticipant(this::reserveStockLevel)
                .step()
                .withCompensation(this::reject)
                .step()
                .invokeParticipant(this::reserveCredit)
                .step()
                .invokeParticipant(this::approve)
                .build();
```

Fonte: do Autor, 2019

6.1.6. Microsserviços Participantes

Diferente da implementação anterior, este modelo não requer nenhum banco de dados adicional como o banco NoSQL MongoDB, pois a leitura é realizada diretamente no banco de dados da plataforma, ou seja, no MySQL. Os três serviços, Order Service, Warehouse Service e Customer Serviçe foram desenvolvidos utilizando o Java Spring Boot e as responsabilidades de cada uma são exatamente as mesmas definidas na implementação anterior, exceto pelo fato

de que o Order Service possui uma responsabilidade adicional que é o gerenciamento da saga de postagem de pedido, este fluxo está descrito na seção 6.2.1, as funções de cada um estão nos itens a seguir.

- **Order Service**: Este serviço implementa um *endpoint* REST responsável por gerenciar as ordens. O serviço persiste o pedido e a representação do CreateOrderSaga no banco de dados MySQL. Utilizando o framework *Eventuate Tram Saga*, ele envia mensagens de comando e processa as respostas.
- **Warehouser Service**: Este serviço implementa um *endpoint* REST para gerenciar o nível de estoque dos produtos, ele é responsável por atualizar estas informações no banco de dados MySQL. Utilizando o framework Eventuate Tram Saga, ele processa comandos, atualiza o nível de estoque para determinado produto e envia de volta uma mensagem de resposta.
- **Customer Service**: Este serviço implementa um *endpoint* REST para gerenciar usuários. Ele é responsável por persistir informações relacionadas ao usuário. Utilizando o framework Eventuate Tram Sagas, ele processa comandos, atualiza o crédito disponível do usuário e envia de volta uma mensagem de resposta.

6.1.7. Removação do módulo CDC

O princípio de funcionamento da plataforma *Eventuate* exige a utilização do módulo *Change Data Caputre* (CDC) para garantir a atomicidade no processamento de um evento. Todo processamento de evento inicia uma nova transação no *MySQL*, e a mesma só será completada no momento em que o evento resultante for inserido com sucesso no banco de dados onde a transação foi iniciada. Isso evita que um evento seja processado parcialmente e uma falha posterior impeça que a mensagem resultante seja enviada, caso isso aconteça, o processamento do evento é revertido e outro consumidor pode processar o evento novamente. Para este estudo, foi necessário desabilitar o CDC e publicar as mensagens diretamente no *Apache Kafka*. O motivo para isso é resultante da necessidade de que o serviço que cria determinado pedido faça a sua aprovação, a fim de ter a mesma base de tempo e calcular o período de execução da regra de negócio.

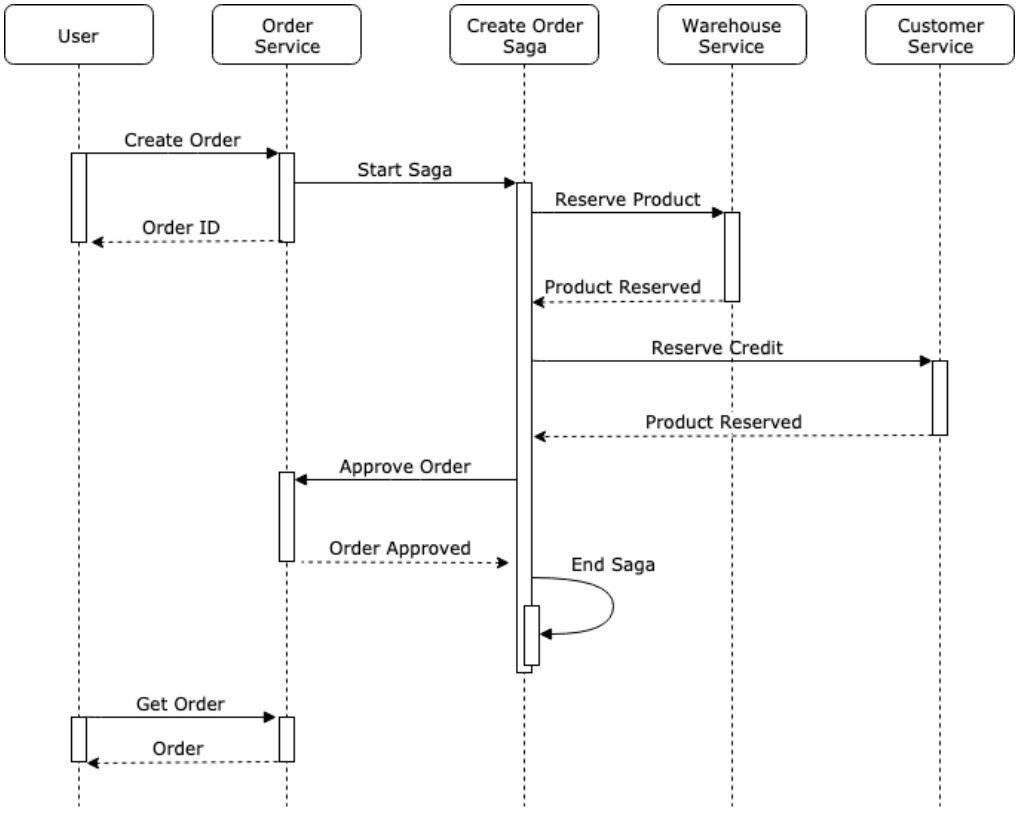
6.2. TRANSAÇÃO PROPOSTA

O objetivo do experimento é analisar os fatores-chave envolvidos durante a execução da transação, peculiaridades de cada modelo, desempenho sobre diferentes níveis de carga, e a diferença de desempenho conforme a variação da quantidade de instâncias. Para isso, a aplicação desenvolvida contempla uma única transação de postagem de pedido, na qual os três serviços, Order Service, Customer Service e Warehouse Service possuem um papel isolado. A transação de postagem de pedido possui comportamento diferente na implementação dentro de cada um dos modelos, as mesmas são descritas com mais detalhes específicos de cada arquitetura nas seções a seguir.

6.2.1. Criar Pedido – Saga Orquestrada

A transação de postagem de pedido dentro do modelo SAGA está representada na Figura 23 como um diagrama de sequência com a função de cada microsserviço. Apesar de o componente *Create Order Saga* fazer parte do *Order Service*, ele foi ilustrado como um componente isolado para fins de melhor entendimento do fluxo da transação. A descrição da transação é feita nos itens abaixo.

Figura 23 - Transação de criação de pedido utilizando SAGA por Orquestração



Fonte: do Autor, 2019

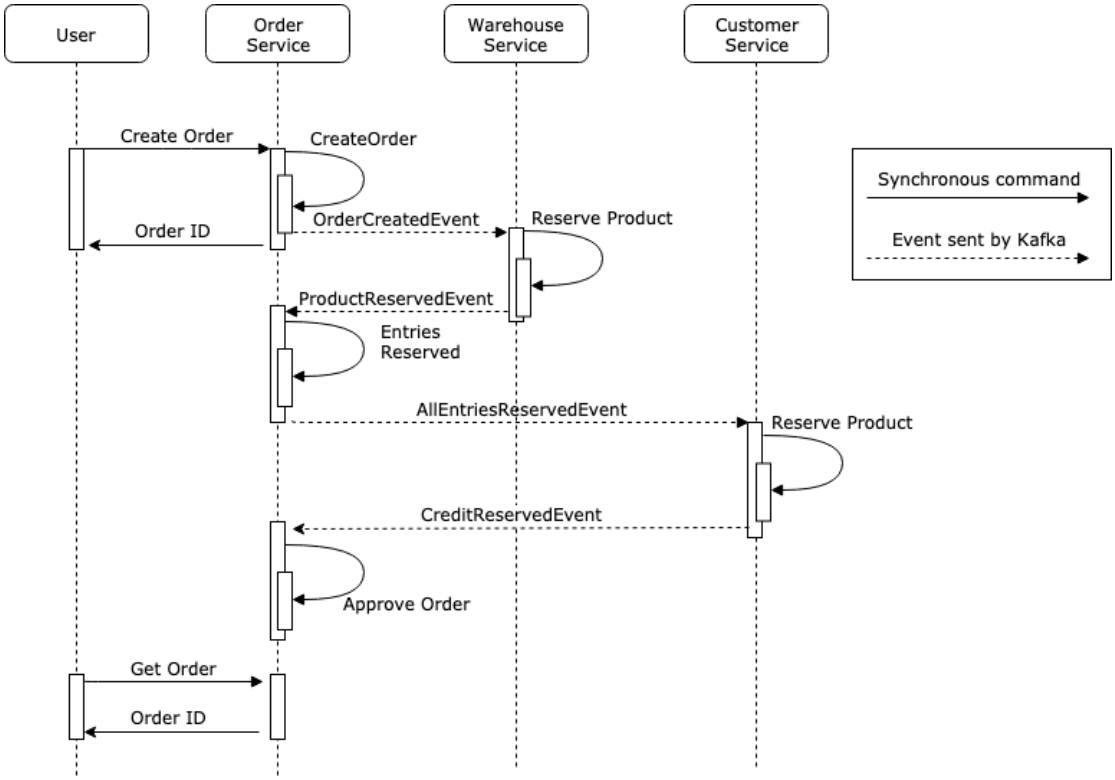
O diagrama de sequência pode ser traduzido nos seguintes passos:

1. O usuário solicita a postagem de um novo pedido ao *Order Service.*
2. O *Order Service* cria um pedido no estado *CREATED* e também uma *CreateOrderSaga* para coordenar o fluxo de aprovação do pedido.
3. O *CreateOrdeSaga* envia um comando *ReserveProduct* para o *Warehouse Service*.
4. O *Warehouse Service* recebe o comando e tenta reservar o produto para aquele pedido. Ele responde com uma mensagem indicando o resultado da operação.
5. O *CreateOrderSaga* recebe a resposta e envia o comando *ReserveCredit* para o *Customer Service.*
6. O *Customer Service* recebe o comando e tenta reservar crédito para aquele pedido, respondendo com uma mensagem o resultado.
7. O *CreateOrderSaga* recebe a resposta.
8. O *CreateOrderSaga* envia um comando *ApproveOrder* para o *Order Service*
9. O *Order Service* recebe o comando e troca o estado do pedido para *APPROVED*, respondendo com o resultado da operação.
10. O *CreateOrderSaga* atualiza o estado da Saga para finalizado.

6.2.2. Criar Pedido – Event Sourcing e CQRS

A Figura 24 representa a transação de criação de pedido utilizando Event Sourcing, ressaltando que a comunicação entre os serviços se dá através do Apache Kafka, porém no diagrama o evento é representado diretamente entre os serviços, a fim de identificar de maneira clara o fluxo da transação. Os passos executados são explicados abaixo.

Figura 24 - Transação de criação de pedido utilizando Event Sourcing



Fonte: do Autor, 2019

O diagrama de sequência pode ser traduzido nos seguintes passos:

1. O usuário solicita a postagem de um novo pedido ao *Order Service.*
2. O *Order Service* cria um pedido no estado *CREATED* e gera um evento *OrderCreatedEvent*.
3. O *OrderCreatedEvent* é recebido pelo *Warehouse Service* que processa o evento reservando o estoque, gerando como resultado um *ProductReservedEvent*.
4. O *ProductReservedEvent* é recebido pelo *Order Service* que atribui o estado do pedido para ALL_ENTRIES_RESERVED, posteriormente gerando um novo evento *AllEntriesReservedEvent*.
5. O *AllEntriesReservedEvent* é recebido pelo *Customer Service* que reserva crédito para o usuário relacionado com o pedido, gerando como resultado o evento *CreditReservedEvent.*

6. O *CreditReservedEvent* é recebido pelo *Order Service* que processa o evento alterando o estado do pedido para APPROVED e gerando um evento *OrderApprovedEvent.*
7. Como nenhum serviço utiliza o *OrderApprovedEvent*, a transação é dada como encerrada.

6.3. TESTES REALIZADOS

Esta seção tem por objetivo descrever as ferramentas utilizadas para realizar os testes, assim como os cenários propostos. Os experimentos descritos na segunda seção são aplicados nas duas implementações.

6.3.1. Ferramenta de teste Apache JMeter

Apache JMeter é uma aplicação *open source* desenhada para realizar testes de cargas funcionais e realizar leituras de performance (FOUNDATION, [s.d.]). Ela pode ser utilizada para simular pesadas cargas no servidor, em grupos de servidores, redes ou objetos para analisar a performance geral sobre diferentes tipos de cargas.

Para realizar os testes nas implementações propostas, um *template* de *request* é aplicado, a requisição utilizada (Figura 25) é um POST que é direcionado para um único *endpoint* do Zuul Server, *http://dominio:8762/order-service/orders*. Esta requisição foi gerada baseado em um *dataset* de dois mil usuários que foram previamente inseridos no banco de dados das duas implementações, a requisição (Figura 25) é única e consome as entradas *customerId*, *orderTotoal*, *productId* e *qty* do arquivo CSV (Figura 26) que é utilizado como input.

Figura 25 - Request de criação de pedido no JMeter

```
{
  "customerId": "${customerId}",
  "orderTotal": {"amount" : ${orderTotal}},
  "entries": [
    {
      "sequence": 0,
      "productId": "${productId}",
      "quantity": "${qty}"
    }
  ]
}
```

Fonte: do Autor, 2019

O principal objetivo da utilização do JMeter nos testes foi simular determinada quantidade de usuários solicitando a postagem de pedido no sistema em um curto período de tempo. O JMeter cria uma nova *thread* para cada usuário e consome uma linha do *dataset* para cada uma das threads geradas, por exemplo, ao criar um teste com 4 usuários, a primeira requisição utilizará os dados da primeira linha na Figura 26 e o quarto utilizará as informações da quarta linha. Caso o número de usuários exceda a quantidade de linhas no *dataset*, o arquivo passa a ser consumido do início novamente.

Figura 26 - *Dataset* consumido pelo JMeter

customerId	orderTotal	productId	qty
0000016a73d283bd-acde4800dee10003	1.9	0000016a73d2f0cc-acde480094d00000	1
0000016a73d283bd-acde4800dee10005	1.9	0000016a73d2f0cc-acde480094d00000	1
0000016a73d283bd-acde4800dee10007	1.9	0000016a73d2f0cc-acde480094d00000	1
0000016a73d283bd-acde4800dee10009	1.9	0000016a73d2f0cc-acde480094d00000	1
...	...	...	...

Fonte: do Autor, 2019

A Figura 27 mostra uma requisição realizada utilizando o *template* da Figura 25 e o *dataset* da Figura 26.

Figura 27 - Requisição realizada com o template

```
POST http://localhost:8762/cqrs-order-service/orders

POST data:
{
  "customerId": "0000016a73d283bd-acde4800dee10205",
  "orderTotal": {"amount" : 1.9},
  "entries": [
    {
      "sequence": 0,
      "productId": "0000016a73d2f0cc-acde480094d00000",
      "quantity": "1"
    }
  ]
}
```

Fonte: do Autor, 2019

6.3.2. Experimentos

Para a execução dos testes, considera-se um cenário hipotético de alta demanda de acesso a um sistema de *e-commerce*, no qual a carga de criação de ordens varia de 100 a 1600 ordens em um curto intervalo de tempo. O principal objetivo com este teste é medir o tempo de resposta médio para a finalização da transação de aprovação das ordens e como esse tempo varia de acordo com a inserção de novas instâncias para atender a alta demanda.

A fim de calcular o tempo em que a transação leva para ser completa, foi adicionado *timestamps* no pedido, T1, T2 e T3. T1 representa o instante em que o *Order Service* criou o pedido, T2 representa o instante em que o *Order Service* aprovou o pedido, e o T3 representa a diferença entre T1 e T2 em milissegundos, assim sendo, representa o tempo total de execução da transação. Estas informações são persistidas no banco de dados para extração posterior a fim de realizar a análise dos resultados.

Foram propostos 3 experimentos variando a carga de usuários simultâneos. Cada experimento é reproduzido 5 vezes alternando a carga de usuários na quantidade de 100, 200, 400, 800 e 1600. Cada experimento tem um número específico de instâncias por microsserviço, o primeiro consiste na utilização de uma instância, o segundo de duas e o terceiro de 4 instâncias. A Tabela 1 ilustra todos os testes propostos. Na seção de resultados os testes serão referenciados utilizando como padrão a seguinte denotação para *Event Sourcing* ***"Experimento 1 – Carga 1 – ES"*** e ***"Experimento 1 – Carga 1 – Saga"*** para sagas.

Tabela 1 - Experimentos e cargas propostas

	Requisições (Usuários)	Instâncias Experimento 1	Instâncias Experimento 2	Instâncias Experimento 3
Carga 1	100	1	2	4
Carga 2	200	1	2	4
Carga 3	400	1	2	4
Carga 4	800	1	2	4
Carga 5	1600	1	2	4

Fonte: do Autor, 2019

Os experimentos 1, 2 e 3 se repetem para as duas implementações. Os ambientes utilizados para os testes estão caracterizados nas duas seções a seguir.

6.3.3. Ambiente de teste local

Os testes foram executados em dois ambientes distintos, o primeiro foi utilizado com o intuito de eliminar a latência de rede através da execução no mesmo computador, este ambiente é descrito na Tabela 3. Os resultados estão disponíveis na seção 6.4.

O framework Eventuate Tram e Eventuate ES utiliza um mecanismo para gerar ids para as mensagens, o qual consiste na concatenação do endereço MAC onde o microsserviço está sendo executado e o *timestamp* no qual a mensagem foi gerada. E devido ao fato de utilizar o mesmo ambiente para fazer os testes, o sistema acaba por gerar os mesmos ids quando gerando dois eventos no mesmo instante, impedindo a continuidade da execução o microsserviço. Para solucionar o problema foi necessário customizar a plataforma a fim de adicionar um número randômico ao endereço MAC, assim evitando colisões de Ids.

Tabela 3 - Ambiente de teste local

Sistema Operacional	macOS Mojave Version 10.14.2
Processador	2.6 GHz Intel Core i7 quad core com 8 MB de cache L3 compartilhado
Memória	16 GB 2133 MHz
Armazenamento	256GB SSD

Fonte: do Autor, 2019

6.3.4. Ambiente de teste distribuído – Amazon EC2

Com o intuito de avaliar as implementações em um âmbito mais condizente com a natureza da aplicação, foi proposto novamente a execução dos testes em um segundo ambiente. O mesmo consiste da utilização do serviço *Amazon Elastic Compute Cloud (EC2),* fornecido pela *Amazon Web Services (AWS),* para a criar instâncias de servidor e realizar o deploy de cada instância de microsserviço em um servidor isolado.

Para montar o ambiente, foram criadas 16 instâncias de servidores com o sistema operacional *Amazon Linux AMI 2018.03.0 (HVM), SSD Volume Type* com o *hardware* t2.micro (1 vCPUs de 2.5 GHz, Intel Xeon Family, 1 GiB de memória RAM). Devido ao fato de grande parte das requisições exigirem interação com o Apache Kafka e MySQL, optamos pela utilização de servidores com performance superior, foram atribuídas a estes microsserviços servidores com o mesmo sistema operacional, porém na categoria t2.xlarge, o que representa

um hardware com 4 vCPUs de 2.3 GHz, Intel Broadwell E5-2686v4 e 16 GiB de memória RAM. A lista completa de instâncias utilizadas e seus respectivos tipos estão descritas na **Tabela 4**.

Tabela 4 - Lista de instância de servidores na Amazon EC2

Name	Instance Type
customer-service1	t2.micro
customer-service2	t2.micro
customer-service3	t2.micro
customer-service4	t2.micro
eureka/zuul	t2.micro
kafka	t2.xlarge
mysql	t2.xlarge
order-service1	t2.micro
order-service2	t2.micro
order-service3	t2.micro
order-service4	t2.micro
order-view-service + MongoDB	t2.micro
warehouse-service1	t2.micro
warehouse-service2	t2.micro
warehouse-service3	t2.micro
warehouse-service4	t2.micro

Fonte: do Autor, 2019

6.3.5. Extração dos resultados

Para a extração dos resultados relacionados aos tempos da transação foram utilizadas as funções disponíveis no banco de dados utilizado por cada implementação. Para cada execução de teste de carga, os dados foram extraídos conforme a **Tabela 5**, utilizando funções de mínimo, máximo, média e desvio padrão dos bancos de dados MySQL e MongoDB, todos os valores se referem a um período de tempo em milissegundos.

Tabela 5 - Coleta de resultados de teste

min	max	média	stddev	primeir ordem criada	última ordem criada	última ordem aprovada	tempo criação	Tempo total
2788	7112	4756	1269	1558499337706	1558499337824	1558499344911	118	7205
2897	7161	4725	1279	1558499429404	1558499429524	1558499436641	120	7237
2720	8953	5329	2177	1558499506511	1558499506666	1558499515608	155	9097
2801.67	7742.00	4936.67	1575.00				131.00	7846.33

Fonte: do Autor, 2019

A **Tabela 6** representa o resultado fornecido pela ferramenta de teste JMeter, os tempos representam o tempo médio de resposta de uma requisição em milissegundos, seguida tempo mínimo, máximo, desvio padrão e *thorughput* calculado. A linha em vermelho representa a média resultante da repetição de três vezes do mesmo teste.

Tabela 6 - Resultado de teste JMeter

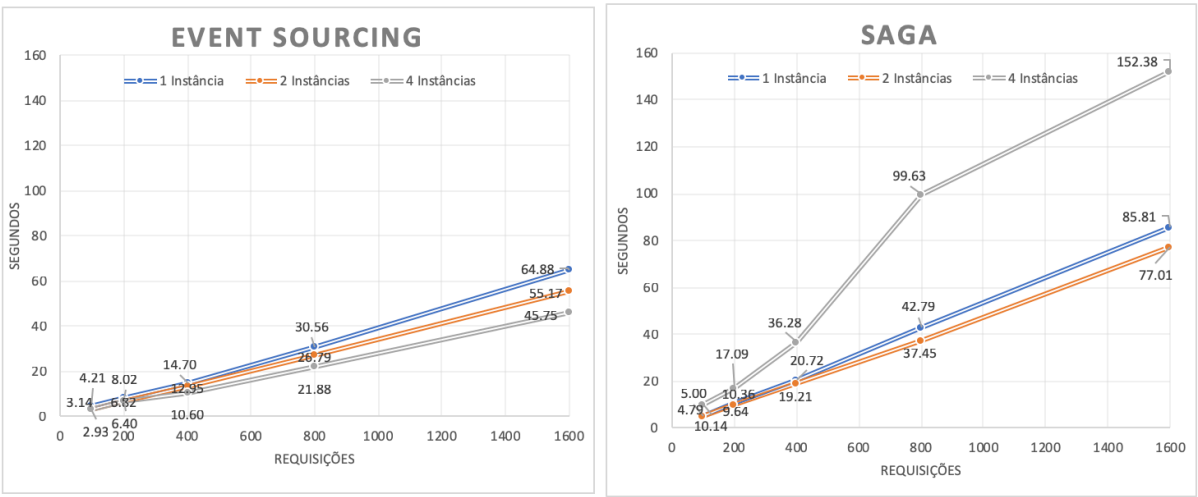
Média	Min	Max	Std. Dev.	Throughput
1741.00	354.00	3112.00	844.00	32.00
1773.00	364.00	3397.00	911.00	29.40
1614.00	318.00	3021.00	791.00	33.00
1709.33	345.33	3176.67	848.67	31.47

Fonte: do Autor, 2019

6.4. RESULTADOS DOS TESTES LOCAIS

Os valores utilizados aqui são frutos de uma média aritmética resultante da execução de três testes no ambiente local com cada uma das cargas propostas na Tabela . O gráfico representado na Figura 28 mostra o tempo levado (em segundos) para processar todas as requisições vs o número de requisições simultâneas, cada reta representa 1, 2 e 4 instâncias de cada microsserviço em ambas as implementações.

Figura 28 - Relação entre tempo total de processamento e número de requisições simultâneas no ambiente local

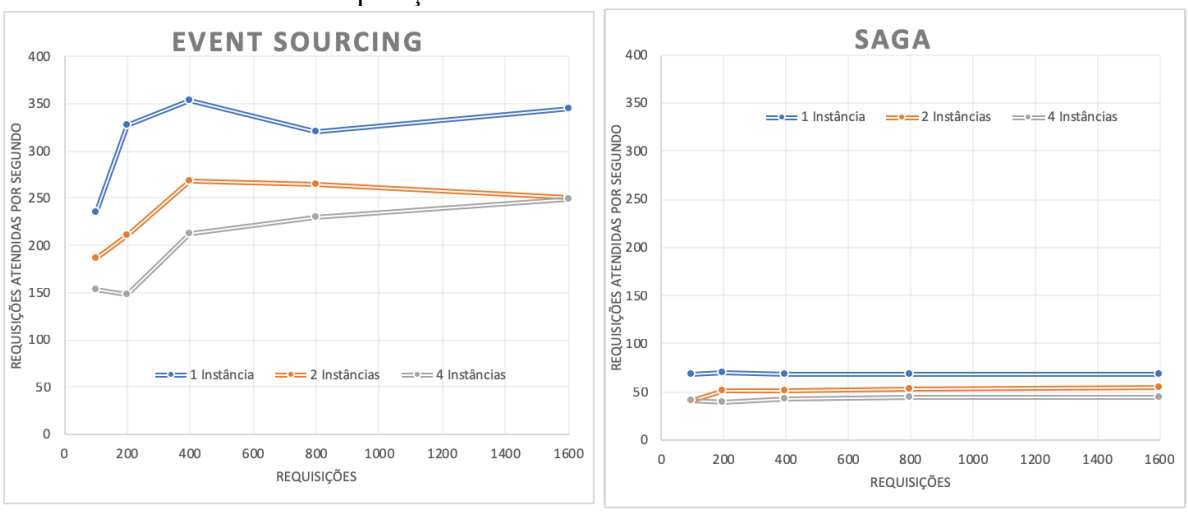


Fonte: do Autor, 2019

O período considerado como tempo total de processamento representado nas curvas da Figura 28 é a diferença entre o momento em que a primeiro pedido do teste foi criado e o momento em que o última pedido foi aprovado. Pelo fato de que o modelo Saga apresenta um

throughput menor, como ilustrado na Figura 29, o tempo total de processamento de todas as ordens do teste acaba sendo relativamente maior. O fato de que a implementação em *Saga* efetua um número muito maior de operações para atender uma única requisição do usuário, acaba diminuindo a vazão das requisições, e como consequência, o tempo necessário para a aplicação terminar de inserir as 1600 ordens - carga de teste 5 - com o estado pendente leva na média de 25 segundos na implementação *Saga*, enquanto a mesma carga na implementação com *Event Sourcing* leva em torno de 5 segundos.

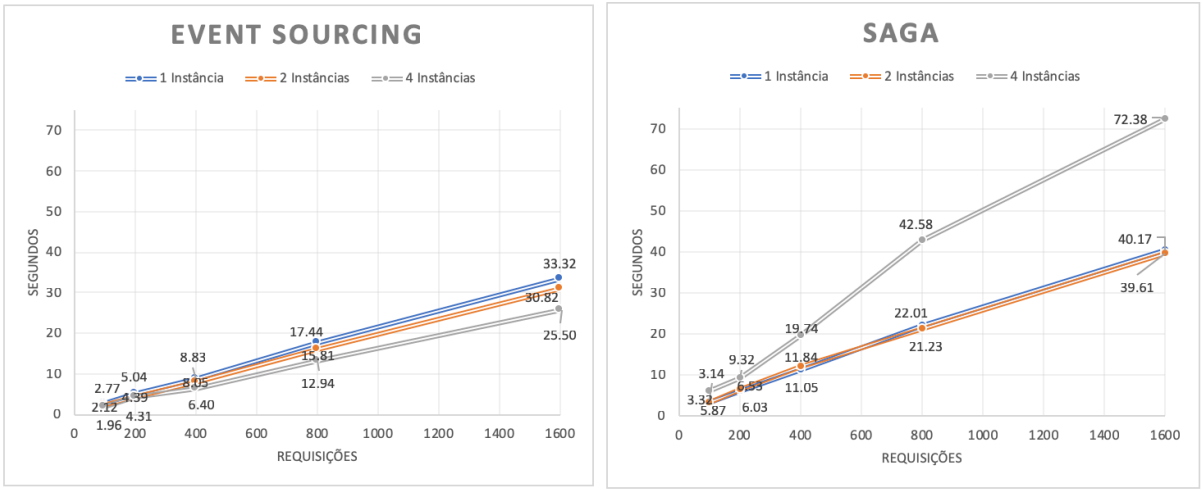
Figura 29 - Relação entre o número de requisições atendidas por segundo e o número de requisições simultâneas no ambiente local



Fonte: do Autor, 2019

Os gráficos da Figura 29 representam quantas requisições por segundo foi possível atender em cada uma das cargas de testes propostas. Percebe-se que o *throughput* da implementação saga é relativamente menor, isso se dá ao fato de que para atender a uma requisição do usuário, o processamento inicial em *Saga* é muito maior comparado com *Event Sourcing*. Em sagas, é necessário criar o pedido, criar uma saga, bloquear a instância da saga no banco de dados e iniciar o envio dos comandos, enquanto que no modelo com *Event Sourcing* é necessário somente criar o evento de pedido criado.

Figura 30 - Tempo médio para completar uma transação de acordo com o número de requisições no ambiente local

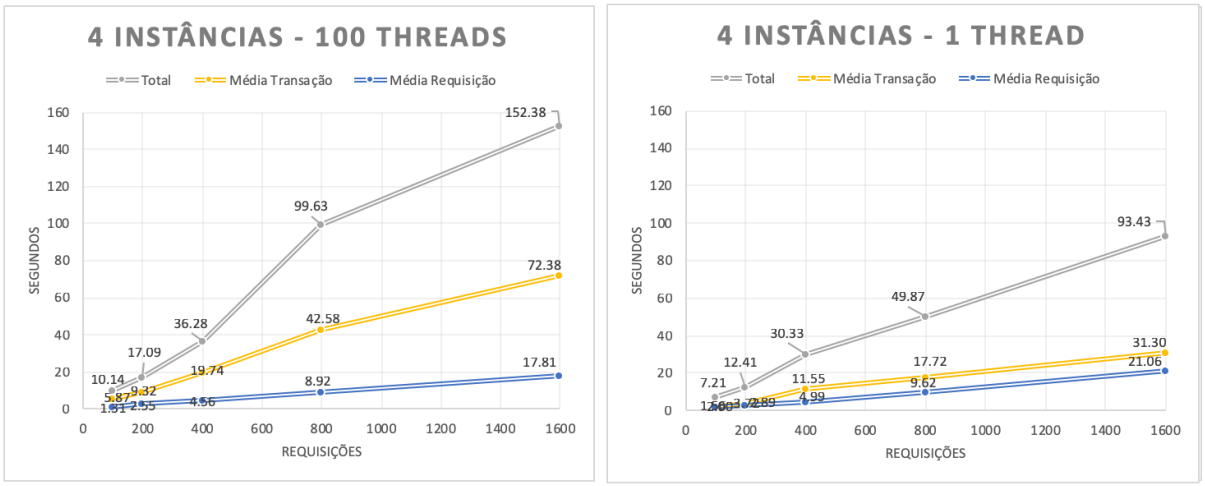


Fonte: do Autor, 2019

Já os gráficos da Figura 30 mostram o tempo médio que cada pedido levou para ser processado, isso representa a diferença entre o momento em que ela foi criada, até o momento em que foi aprovada. De maneira geral, *Event Sourcing* pode ser considerada mais performática em todos os cenários de testes propostos. O primeiro incremento de instâncias resultou em uma diminuição de 7.5% no tempo médio de processamento, e ao aumentar a quantidade de instâncias para 4 houve uma nova redução de 13%. Já na implementação com Saga, o primeiro aumento para duas instâncias não representou um ganho significativo, mantendo os tempos muito semelhantes, e quando aumentado para 4 instâncias, isso representou uma sobrecarga no processamento, resultando em um acréscimo de 83% no tempo médio de processamento da transação.

A diferença nos tempos de processamento em relação a quantidade de instâncias nas duas implementações pode estar relacionada com o *overhead* do processador para criar *n* Threads, onde *n* representa a quantidade de requisições simultâneas. O resultado tende a ser diferente em um ambiente distribuído. A grande quantidade de requisições simultâneas acaba gerando overhead de chaveamento entre threads no processador. Para melhorar o desempenho, foi executado um segundo teste diminuindo de 100 para 1 o número de threads que processam requisições paralelas. Como consequência foi necessário aumentar o número de requisições que podem ser enfileiradas para 1600. Neste cenário, uma única *Thread* processa todas as requisições, e não há concorrência pelo acesso ao pool de conexões do banco de dados, resultando em acessos rápidos. Foi realizado um teste com 4 instâncias utilizando todas as cargas, os tempos são ilustrados na Figura 31.

Figura 31 - Variação no número de *threads* de requisições (*Saga*)



Fonte: do Autor, 2019

Na Figura 31, o gráfico da esquerda representa a configuração padrão do servidor Tomcat, o qual determina a quantidade de 100 threads para atender requisições simultâneas do usuário, já o gráfico da direita representa a adição de uma configuração manual para limitar esse número em 1 *thread*. No caso de 1 *thread*, todas as demais requisições serão adicionadas a uma fila e atendidas no modelo *First In First Out (FIFO)*. Utilizando 1 *thread*, o tempo médio de processamento da requisição do usuário diminuiu de 4700ms para 360ms, dando mais vazão as transações, por isso percebe-se que o tempo de transação diminui na média de 72 segundos para 31 segundos no caso de 1600 requisições simultâneas, em contrapartida, o tempo médio de espera do usuário pelo término da requisição aumentou de 17 para 21 segundos.

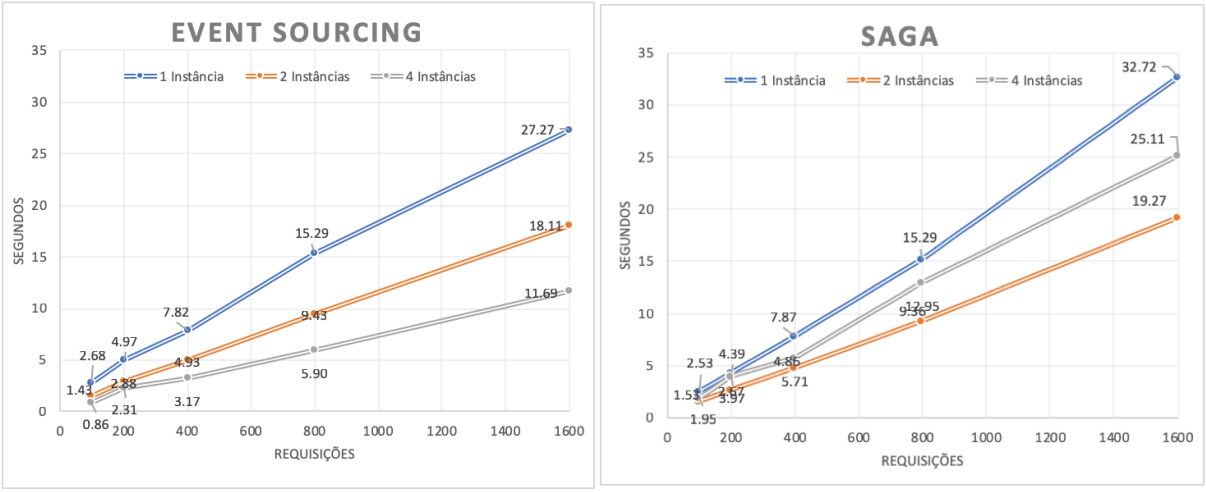
O mesmo teste foi realizado utilizando Event Sourcing, porém o resultado foi ligeiramente diferente. O tempo médio de processamento caiu pela metade, em contrapartida, o tempo em que o usuário espera pela requisição aumentou em 4 vezes, no caso de 1600 requisições, este valor subiu de 3 segundos para 12 segundos. Mesmo com este cenário, o *Event Sourcing* continua sendo mais performático.

Importante ressaltar que este cenário de teste tem fins somente voltados para a análise, pois não é viável a limitação de 1 thread para atender requisições do usuário, caso esta *thread* venha a tomar tempo demais para executar uma requisição, o microsserviço pode ficar indisponível para os demais usuários do sistema.

6.5. RESULTADOS TESTES COM O AMBIENTE DISTRIBUÍDO (AMAZON EC2)

Os resultados dos testes aplicados no ambiente da *Amazon EC2* se baseiam na mesma regra da seção anterior, ou seja, cada carga de teste é executada três vezes e a média desta execução é adotada como valor final. No decorrer desta seção são apresentados os gráficos comparativos das implementações *Saga* e *Event Sourcing* utilizando instâncias em servidores da *Amazon EC2.* Com o intuito de diminuir o máximo possível da latência, as requisições originárias do JMeter foram iniciadas dentro da rede onde os servidores estão hospedados.

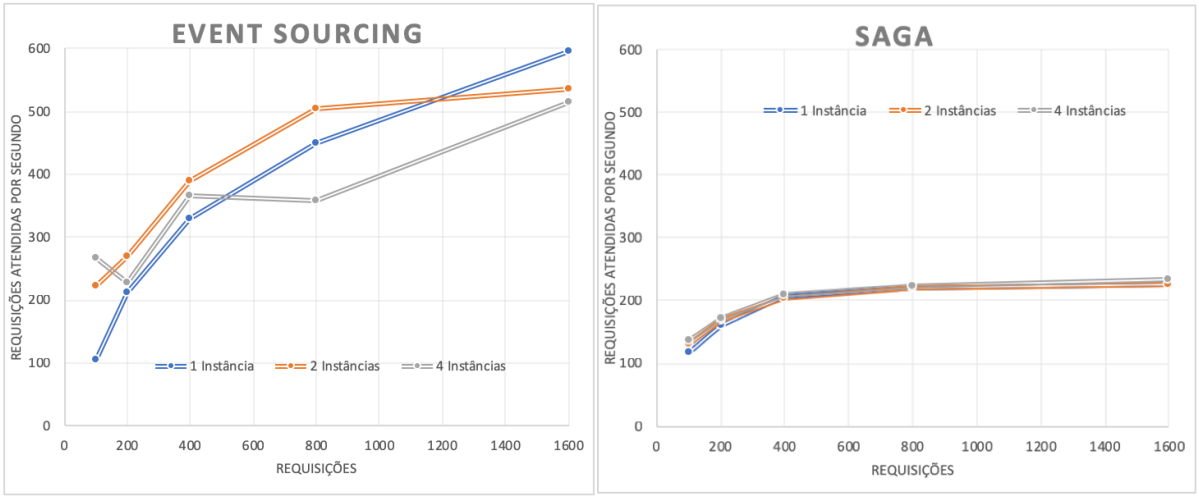
Figura 32 - Relação entre tempo total de processamento e número de requisições simultâneas no ambiente Amazon EC2



Fonte: do Autor, 2019

A Figura 32 ilustra dois gráficos para fazer a comparação entre o tempo total requerido para receber as requisições, processá-las e terminar o fluxo da transação de aprovação de pedido com todas as cargas de testes propostas na Tabela . O gráfico da esquerda mostra os tempos médios na implementação com *Event Sourcing* e o da direita para a implementação com *Saga.* Cada gráfico apresenta três linhas com as cores azul, laranja e preto, as quais representam a execução dos testes respectivamente com 1 instância, 2 instâncias e 4 instâncias. O ganho de performance – teste com carga de 1600 usuários simultâneos – durante a adição de novas instâncias no modelo com *Event Sourcing* é dado como 1:1.5:2.3, enquanto na implementação com *Saga* esse ganho é representado como 1:1.7:1.3. Este comparativo leva em conta vazão de requisições que cada implementação consegue entregar, no caso, *Event Sourcing* possui uma vazão muito maior, e no caso dos testes de 1 instâncias, a diferença de tempo total entre as duas aplicações é dada pela baixa vazão de atendimento de requisições (Figura 33) na implementação *Saga.*

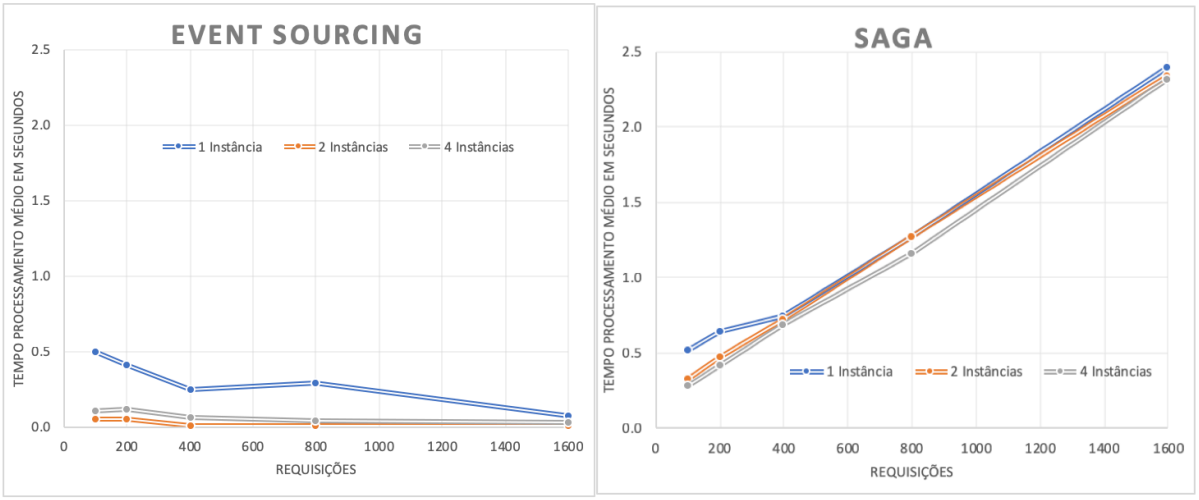
Figura 33 - Relação entre o número de requisições atendidas por segundo e número de requisições simultâneas no ambiente Amazon EC2



Fonte: do Autor, 2019

A Figura 33 demonstra a quantidade de requisições que conseguem ser atendidas, por segundo, em cada uma das implementações. A implementação com *Event Sourcing* converge para uma média de 550 requisições por segundo, enquanto a *Saga* fica em 230 por segundo. Essa diferença de *throughput* é dada pela quantidade de operações que a Saga realiza durante o processamento de cada requisições do usuário, descrita na seção anterior.

Figura 34 - Tempo médio para atender cada requisição no ambiente Amazon EC2

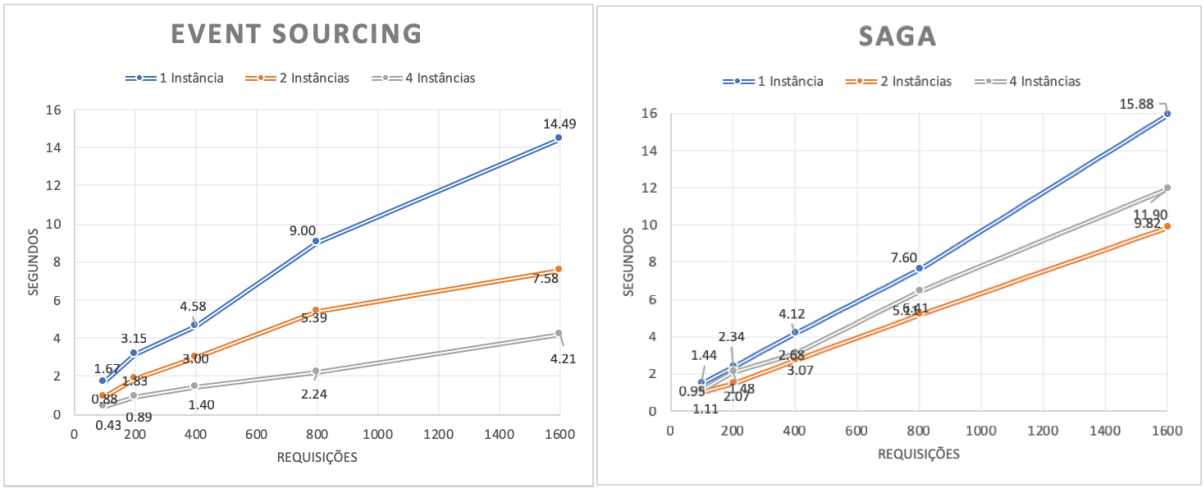


Fonte: do Autor, 2019

A Figura 34 mostra o tempo médio de processamento para cada requisição do usuário – tempo levado para retornar o id do pedido com estado PENDENTE ao usuário. Pelos gráficos é perceptível que o aumento de instâncias em *Event Sourcing* traz ganhos mantendo o tempo médio de atendimento das requisições em uma constante próxima a zero (70ms), enquanto na

implementação Saga, mesmo com a adição de novas instâncias, a reta permanece linear, aumentando o tempo proporcionalmente a quantidade de requisições simultâneas.

Figura 35 - Tempo médio para completar uma transação de acordo com o número de requisições no ambiente Amazon EC2



Fonte: do Autor, 2019

A Figura 35 ilustra dois gráficos, os quais mostram o tempo médio necessário para finalizar uma transação em cada uma das implementações, com um número de instâncias diferente e variando a carga, seguindo o mesmo plano de testes descrito anteriormente. O tempo de processamento representa a diferença entre o instante em que o pedido foi criado e o tempo de aprovação da mesma. Com os resultados obtidos, se percebe que o *Event Sourcing* reage melhor ao aumento de instâncias, representando um ganho de 1:1.9:3.44, já a implementação *Saga* tem um ganho inferior definido por 1:1.6:1.3.

7. CONCLUSÃO

Neste trabalho pode-se conferir uma introdução sobre os principais conceitos envolvidos em uma arquitetura de microsserviços. Foi abordado algumas das motivações que levam a este tipo de arquitetura e quais são os principais componentes básicos, assim como algumas das possibilidades para implementação de regras de negócio que exigem dados distribuídos em diversos microsserviços.

A fundamentação teórica sobre microsserviços, escalabilidade e gerenciamento de dados distribuídos apresentada nos primeiros capítulos, foi fundamental para o desenvolvimento da aplicação e testes descritos no capítulo 6. Os conceitos abordados foram necessários para ilustrar os objetivos do trabalho e justificar as implementações realizadas.

Dentre os dois modelos estudados, pelos resultados dos testes realizados, independente do ambiente, pode-se concluir que a utilização do modelo arquitetural baseado em *Event Sourcing* tende a ser mais rápido, porém a sua implementação requer mais cuidados e coordenação entre os times de desenvolvimento, pois a lógica de negócio fica distribuída em cada microsserviço. Já a implementação utilizando Saga, apesar de ser mais lenta, facilita o desenvolvimento de regras de negócios complexas, e que requeiram muitos participantes, principalmente pelo fato de a lógica ficar centralizada em um único ponto de referência.

Para a escolha do padrão a ser implementado, deve-se avaliar qual métrica é desejada atender com melhor desempenho. Caso a arquitetura necessite um nível alto nível *throughput*, deve-se optar pela utilização de uma arquitetura baseada em eventos. Caso o sistema não possua uma alta demanda de acessos, utilização do Saga com até 800 requisições simultâneas no modelo possui o mesmo tempo de processamento do *Event Sourcing*, e neste caso a sua utilização facilita o desenvolvimento.

Outro ponto positivo na implementação com *Event Sourcing* é a disponibilidade, o tempo consumido para iniciar a transação é relativamente menor, ele se baseia em simplesmente inserir um único evento no banco de dados, e logo em seguida retorna o *id* ao usuário, deixando os demais componentes darem continuidade a transação, assim liberando a *thread* para a próxima requisição ser consumida. Já a implementação em saga consiste em inserir o pedido com seu estado inicial no banco de dados, em seguida persistir a representação da saga, gerar um *lock* neste registro e iniciar a invocação dos participantes, que se resume em persistir as mensagens que serão consumidas posteriormente pelo serviço de CDC. Em suma, o tempo

consumido para atender as requisições vindas do usuário é consideravelmente maior na implementação da *Saga*, o que justifica o *throughput* médio de 230 requisções por segundo, enquanto a implementação com *Event Sourcing* entrega um *throughput* médio de 550 requisições por segundo no ambiente *Amazon EC2*.

Em casos onde seja necessário tratar uma coleção grande de objetos, o modelo de *Event Sourcing* pode se tornar mais desvantajoso em relação ao *Saga*, pois no *Event Sourcing*, eventos são direcionados a uma entidade, como por exemplo, se tivermos uma lista de 1000 produtos diferentes no pedido, 1000 eventos serão disparados solicitando a reserva de cada um destes produtos. Na modelo *Saga*, comandos são disparados para microsserviços, independente de instância de entidade, permitindo que a lista inteira seja processada em um único comando.

Em relação ao consumo de mensagens, *Event Sourcing* subscreve a todas as mensagens de uma entidade, mesmo quando há a necessidade de consumir somente um tipo de evento, isso implica em receber todos os eventos oriundos de tal microsserviço, e então internamente é tomada a decisão se é necessário processar o evento ou comitar a mensagem no Kafka sem realizar nenhum processamento. Já em *Saga*, as mensagens são diretas para os canais de comando e de resposta, diminuindo a necessidade de troca de mensagens que serão descartadas.

Em relação a escalabilidade, o aumento do número de instâncias para atender as requisições trouxeram grande ganho para a implementação em *Event Sourcing*, diminuindo drasticamente o tempo de processamento das transações, este tempo ficou constante e próximo a zero, independente do número de requisições simultâneas. Já a implementação SAGA orquestrada apresentou queda na performance quando aumentando de 2 para 4 instâncias, e não houve melhora no tempo médio de atendimento das requisições do usuário.

Como sugestão de trabalho futuro, seria interessante fazer a comparação entre os *frameworks* de mensageria Apache Kafka e RabbitMQ, para que assim possa ser concluído quais são as principais diferenças e cenários de uso de cada uma.

REFERÊNCIAS BIBLIOGRÁFICAS

ABBOTT, M. L.; FISHER, M. T. **The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise**, 2009.

AGRAWAL, D. et al. Database scalability, elasticity, and autonomy in the cloud (Extended abstract). **Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)**, v. 6587 LNCS, n. PART 1, p. 2–15, 2011.

AMARAL, M. et al. Performance evaluation of microservices architectures using containers. **Proceedings - 2015 IEEE 14th International Symposium on Network Computing and Applications, NCA 2015**, p. 27–34, 2016.

BREWER, E. A. Towards robust distributed systems (abstract). **Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing - PODC '00**, n. January 2000, p. 7, 2000.

BREWER, E. A. CAP Twelve Years Later: How the "Rules" Have Changed. **ICDE 2012 28th IEEE International Conference on Data Engineering University of California, Berkeley CAP**, 2012.

BROWN, K.; WOOLF, B. Implementation patterns for microservices architectures. **Pattern Language of Patterns**, p. 7, 2016.

CARNELL, J. **Spring Microservices in Action**. [s.l.] Manning, 2017.

CHEN, L. Microservices: Architecting for Continuous Delivery and DevOps. **Proceedings - 2018 IEEE 15th International Conference on Software Architecture, ICSA 2018**, n. March, p. 39–46, 2018.

DAYA, S. et al. Microservices from Theory to Practice Creating Applications in IBM Bluemix Using the Microservices Approach. **Ibm**, p. 170, 2015.

DRAGONI, N. et al. Microservices: Yesterday, Today, and Tomorrow. p. 195–212, 2017.

EVENTUATE.IO. **Eventuate Platform**. Disponível em: <http://eventuate.io/>, 2019.

EVENTUATE, I. **Eventuate Tram**. Disponível em: <http://eventuate.io/abouteventuatetram.html>, 2014.

FOUNDATION, A. S. **Apache JMeter**. Disponível em: <https://jmeter.apache.org/>, 2019.

FOUNDATION, A. S. **Apache Zookeeper**. Disponível em: <https://zookeeper.apache.org/>, 2019.

FOUNDATION, A. S. **Apache Kafka**. Disponível em: <https://kafka.apache.org/>., 2019.

FOWLER, M. **CQRS**. Disponível em: <https://martinfowler.com/bliki/CQRS.html>, 2014.

FOX, A.; BREWER, E. A. Harvest and yield . 2001.

GARCIA-MOLINA, H.; SALEM, K. Sagas. **ACM SIGMOD Record**, v. 16, n. 3, p. 249–259, 1987.

INC, M. **MongoDB**. Disponível em: <https://www.mongodb.com/>.

KALSKE, M. Transforming monolithic architecture towards microservice architecture. 2017.

KARWATKA, P. et al. Microservices Architecture for eCommerce. **OEX Divante Business Services**, [s.d.].

LARS, F.; ZAHLE, T. U. Semantic ACID Properties in Multidatabases Using Remote Procedure Calls and Update Propagations. **Software Practice and Experience 28(1):77-98**, 1998.

LECHTENB, J. 2-Phase Commit Protocol. p. 1–6, 2009.

LEWIS, J.; FOWLER, M. **Microservices**. Disponível em: <https://martinfowler.com/articles/microservices.html>. Acesso em: 8 set. 2018.

LIMITEE, D. U. C. Towards an Understanding of Microservices. v. 173, n. August, p. 149–173, 2017.

MESSINA, A. et al. The Database-is-the-Service Pattern for Microservice Architectures. v. 8060, n. January 2018, 2013.

MOZAFFARI, B. Spring Boot Microservices on Red Hat OpenShift Container Platform 3. p. 143–152, 2015.

NAMIOT, D.; SNEPS-SNEPPE, M. On Micro-services Architecture. **International Journal of Open Information Technologies**, v. 2, n. 9, p. 24–27, 2014.

NETFLIX, E. **Eureka**.

NETFLIX, Z. **Zuul**. Disponível em: <https://github.com/Netflix/zuul/wiki>.

NEWMAN, S. **Building Microservices**. [s.l.] O'Reilly Media, Inc., 2015.

PAHL, C.; JAMSHIDI, P. Microservices: A Systematic Mapping Study. **Proceedings of the 6th International Conference on Cloud Computing and Services Science**, n. May, p. 137–146, 2016.

RICHARDSON, C. **Introduction to Microservices**. Disponível em: <https://www.nginx.com/blog/introduction-to-microservices/>, 2015.

RICHARDSON, C. **Event-Driven Data Management for Microservices**. Disponível em: <https://dzone.com/articles/event-driven-data-management-for-microservices-par>. Acesso em: 8 nov. 2018.

RICHARDSON, C. **Microservices Pattern. Manning.** 2018.

RISTHEIN, E. Faculty of Information Technology Migrating the Monolith to a Microservices Architecture : the Case of TransferWise. 2015.

SANTIS, S. DE et al. Evolve the Monolith to Microservices with Java and Node. **IBM Redbooks**, 2016.

SIMON, S. Brewer's CAP Theorem Report to Brewer's original presentation of his CAP Theorem at the Symposium on Principles of Distributed Computing (PODC) 2000. **CS341 Distributed Information Systems, University of Basel, HS2012**, p. 1–6, 2012.

TAIBI, D.; LENARDUZZI, V.; PAHL, C. Architectural Patterns for Microservices: A Systematic Mapping Study. **Proceedings of the 8th International Conference on Cloud Computing and Services Science**, n. March, p. 221–232, 2018.

WOLFF, E. **Microservices: Flexible Software Architecture**. Crawfordsville, Indiana - United States. First: Pearson Education, Inc., 2016.

XIE, C. et al. Salt : Combining ACID and BASE in a Distributed Database. **Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation. Broomfield, CO**, p. 495–509, 2014.

YANAGA, E. **Migrating to Microservice Databases From Relational Monolith to Distributed Data**. 1. ed. [s.l.] O'Reilly Media, Inc., 2017.

YÖYEN, E. Building Microservices with Spring Boot. **Leanpub**, 2017.