UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
COORDENAÇÃO DO CURSO DE TECNOLOGIA EM SISTEMAS PARA INTERNET
CAMPUS TOLEDO

JOÃO PEDRO MONTEIRO FERNANDES

ESTRATÉGIAS PARA COREOGRAFIA DE MICROSSERVIÇOS

TRABALHO DE CONCLUSÃO DE CURSO

TOLEDO
2018

JOÃO PEDRO MONTEIRO FERNANDES

ESTRATÉGIAS PARA COREOGRAFIA DE MICROSSERVIÇOS

Trabalho de Conclusão de Curso de Graduação, apresentado ao Curso Superior de Tecnologia em Sistemas para Internet, da Universidade Tecnológica Federal do Paraná – UTFPR, como requisito parcial para obtenção do título de Tecnólogo.

Orientador: Prof. Dr. Edson Tavares de Camargo

TOLEDO
2018

TERMO DE APROVAÇÃO

JOÃO PEDRO MONTEIRO FERNANDES

ESTRATÉGIAS PARA COREOGRAFIA DE MICROSSERVIÇOS

Este trabalho de conclusão de curso foi apresentado no dia 30 de novembro de 2018, como requisito parcial para obtenção do título de Tecnólogo em Sistemas de Telecomunicações, outorgado pela Universidade Tecnológica Federal do Paraná. O aluno foi arguido pela Banca Examinadora composta pelos professores abaixo assinados. Após deliberação, a Banca Examinadora considerou o trabalho aprovado.

_____

Prof. Dr. Wesley Klewerton Guêz Assunção
Responsável pela Atividade de Trabalho de Conclusão de Curso
Coordenação do Curso de Tecnologia em Sistemas para Internet

BANCA EXAMINADORA

_____

Prof. Dr. Vilson Luiz Dalle Mole
UTFPR

_____

Prof. Ms. Eduardo Pezutti Beletato dos Santos
UTFPR

_____

Prof. Dr. Edson Tavares de Camargo
Orientador
UTFPR

“A Folha de Aprovação assinada encontra-se na Coordenação do Curso”

A todos aqueles que, à sua maneira, usam a tecnologia a fim de impactar positivamente a vida de outras pessoas.

AGRADECIMENTOS

Agradeço primeiramente a Deus, por me permitir ter saúde, força, dedicação e persistência durante todo o curso.

Aos meus pais, Marco e Rosi, e irmão, Jonas, pelo amor, incentivo à minha educação formal e apoio incondicional em todos os momentos. Foi através deles que eu aprendi a valorizar o estudo.

À minha namorada, Marcela, por me apoiar, suportar e dar forças nos momentos mais difíceis. Também pelas incontáveis revisões textuais na reta final dessa trajetória.

A todo o corpo docente do curso, por viabilizar minha formação durante todos esses semestres.

Por último, e mais importante, agradeço ao meu orientador pela disponibilidade e disposição em contribuir com seu intelecto para essa pesquisa. Não fui seu aluno nas disciplinas do curso, mas felizmente nos conhecemos no momento exato para que esse trabalho se concretizasse.

We can't say for sure where we'll end up, but one of the challenges of software development is that you can only make decisions based on the imperfect information that you currently have to hand. (FOWLER; LEWIS, 2014).

Nós não podemos dizer com certeza onde vamos parar, mas um dos desafios do desenvolvimento de *software* é que você só pode tomar decisões com base nas informações imperfeitas que tiver em mãos. (FOWLER; LEWIS, 2014).

RESUMO

FERNANDES, João Pedro Monteiro. **Estratégias para Coreografia de Microsserviços.** 2018. Trabalho de Conclusão de Curso (Curso Superior de Tecnologia em Sistemas para Internet), Universidade Tecnológica Federal do Paraná. Toledo, 2018.

A arquitetura de microsserviços é um estilo arquitetural em que a aplicação é composta por um conjunto de serviços independentes e com responsabilidades bem definidas. Comparados às aplicações tradicionais, chamadas de monolíticas, os microsserviços apresentam baixo acoplamento, garantem a interoperabilidade da aplicação e possibilitam a escalabilidade independente das funcionalidades que de fato requerem maior desempenho. Apesar das suas vantagens, a natureza distribuída da arquitetura de microsserviços requer que os serviços cooperem entre si para compor funcionalidades complexas. Para realizar a composição de microsserviços, há duas estratégias que se destacam: a orquestração e a coreografia. A estratégia de orquestração implica no alto acoplamento da comunicação uma vez que conta com uma entidade central que atua como orquestrador. Por outro lado, a coreografia favorece o baixo acoplamento e a descentralização do *software*. No entanto, por se tratar de uma tecnologia proposta recentemente, o desafio em criar os microsserviços e coreografá-los é grande uma vez que os microsserviços ainda estão sendo conhecidos pela comunidade de desenvolvedores. Muitos dos trabalhos encontrados na literatura ainda se preocupam em apresentar um cenário de aplicação de microsserviços e somente contrapô-lo à arquitetura monolítica. O objetivo deste trabalho é investigar estratégias para coreografia de microsserviços. Como resultado deste trabalho, descobriu-se que a coreografia de microsserviços pode ser aplicada através de duas estratégias. A primeira estratégia é baseada em eventos e a segunda estratégia é chamada de programação coreográfica. As duas estratégias são descritas neste trabalho juntamente com exemplos de estudo de caso. Uma aplicação construída de acordo com a estratégia de coreografia baseada em eventos também foi construída e avaliada.

Palavras chave: Microsserviços. Coreografia de Microsserviços. Coreografia Baseada em Eventos. Programação Coreográfica.

ABSTRACT

FERNANDES, João Pedro Monteiro. **Strategies for Microservices Choreography**. 2018. Trabalho de Conclusão de Curso (Curso Superior de Tecnologia em Sistemas para Internet), Universidade Tecnológica Federal do Paraná. Toledo, 2018.

The microservice architecture is an architectural style in which the application is composed of a set of independent services and with well-defined responsibilities. Compared to traditional applications, named monolithic, the microservices have a low coupling, guarantee the interoperability of the application and allow the independent scalability of the functionalities that require higher performance. Despite its advantages, the distributed nature of the microservice architecture requires that the services cooperate to compose complex features. There are two strategies to perform the composition of microservices: the orchestration and the choreography. The orchestration strategy implies the high coupling of communication since it has a central entity that acts as an orchestrator. On the other hand, choreography promotes low coupling and software decentralization. However, because it is a newly proposed technology, the challenge in creating the microservices and choreographing them is hard since the microservices are still being known by the developer community. Many of the works found in the literature are always concerned with presenting a scenario of application of microservices and only compare it to the monolithic architecture. The objective of this work is to investigate strategies for the choreography of microservices. As a result of this work, it was discovered that microservice choreography could be implemented through two strategies. The first one is an event-based strategy, and the other is called choreographic programming. These two strategies are described in this paper along with case study examples. An application built according to the event-based choreography strategy was also implemented and explained.

Keywords: Microservices. Microservices Choreography. Event Based Choreography. Choreographic Programming.

LISTA DE FIGURAS

SUMÁRIO

1 INTRODUÇÃO

Aplicações *web* são comumente desenvolvidas utilizando linguagens de programação que oferecem meios para decompor suas regras de negócio em unidades menores. Essas unidades geralmente compartilham recursos entre si e dependem do ambiente de execução em que a aplicação é implantada (GUIDI et al., 2017). Grande parte das aplicações é construída para trocar suas mensagens fazendo uso apenas da memória principal. Ou seja, são desenvolvidas considerando uma arquitetura onde todas as suas unidades menores estão em um único ambiente de execução. Ainda que as linguagens forneçam meios para decompor a solução do problema, as unidades criadas não são independentemente executáveis, pois estão inseridas no mesmo processo. Dessa forma, a aplicação *web* se torna uma única unidade executável, que é denominada de arquitetura monolítica.

Apesar dos benefícios em construir uma aplicação monolítica, esse estilo de arquitetura dificulta a escalabilidade e a distribuição do *software* (TRIPOLI; CARVALHO, 2016). Uma alternativa que está se tornando popular e ganhando tração na comunidade de desenvolvimento é a arquitetura de microsserviços. A arquitetura de microsserviços é um estilo arquitetural em que a aplicação é composta por um conjunto de serviços independentes e com responsabilidades bem definidas (FOWLER; LEWIS, 2014). Comparados ao monólito, os microsserviços apresentam baixo acoplamento, garantem a interoperabilidade da aplicação e possibilitam a escalabilidade independente das funcionalidades que de fato requerem maior desempenho (DRAGONI et al., 2017).

A característica de independência dos microsserviços permite implantar, corrigir, escalar individualmente cada serviço e distribuí-los em diferentes servidores localizados tanto em uma rede local quanto na Internet. Geralmente, os microsserviços fazem uso da tecnologia de contêineres. Os contêineres oferecem isolamento entre a aplicação de *software* e o ambiente de execução, garantindo flexibilidade e portabilidade no gerenciamento dos serviços (PEINL; HOLZSCHUHER; PFITZER, 2016). Por outro lado, a característica distribuída da arquitetura de microsserviços requer que os serviços cooperem entre si para compor funcionalidades complexas e mais elaboradas da aplicação (GUIDI et al., 2017). Nesse contexto, há duas estratégias que se destacam: a orquestração e a coreografia. A estratégia de

orquestração para a composição de microsserviços é obtida através da implementação de uma entidade central que atua como orquestrador. No entanto, essa estratégia implica no alto acoplamento da comunicação.

Coreografia é um método de composição de serviços distribuídos em que cada entidade possui a responsabilidade de conhecer o momento e a forma de desempenhar sua função na composição (BRAVETTI; ZAVATTARO, 2007). Essa estratégia define que os serviços devam conhecer minimamente o escopo global e cooperar entre si para determinar quando deverão participar de um determinado fluxo. Essa característica torna a coreografia mais apropriada à arquitetura de microsserviços, pois favorece o baixo acoplamento e a descentralização do *software*. No entanto, por se tratar de uma tecnologia proposta recentemente, o desafio em criar os microsserviços e coreografá-los é grande uma vez que os microsserviços ainda estão sendo conhecidos pela comunidade de desenvolvedores. A coreografia exige a comunicação distribuída entre unidades de *software* possivelmente localizadas em domínios distintos. Tal fato exige protocolos de comunicação, ferramentas para localização e a composição da regra de negócio usando unidades distribuídas. Muitos dos trabalhos encontrados na literatura ainda se preocupam em apresentar um cenário de aplicação de microsserviços e contrapô-lo à arquitetura monolítica. Ou ainda, se preocupam em aplicar a composição de microsserviços através da orquestração.

O objetivo deste trabalho é investigar estratégias para coreografia de microsserviços. Para atingir o objetivo proposto, os seguintes objetivos específicos foram definidos:

- Pesquisar e sumarizar referencial teórico sobre arquitetura monolítica, arquitetura de microsserviços e coreografia;
- Investigar as ferramentas e linguagens disponíveis para coreografia de microsserviços;
- Projetar, implementar e coreografar uma aplicação *web* nomeada Crypsense com arquitetura de microsserviços;
- Desenvolver uma análise crítica considerando aspectos de implementação e coreografia;
- Redigir TCC e artigo científico.

Para atingir os objetivos propostos, uma extensa busca na literatura foi realizada. Como resultado deste trabalho, descobriu-se que a coreografia de microsserviços pode ser aplicada através de duas estratégias. A primeira estratégia é baseada em eventos e a segunda estratégia é chamada de programação coreográfica. As duas estratégias são descritas neste trabalho juntamente com exemplos de estudo de caso. Uma aplicação construída de acordo com a estratégia de coreografia baseada em eventos também foi construída e avaliada.

Este texto segue organizado da seguinte forma. O Capítulo 2 apresenta o referencial teórico, incluindo definições sobre a arquitetura monolítica, a arquitetura de microsserviços, a composição de microsserviços através de orquestração e coreografia e modelagem de coreografias. O Capítulo 3 explora a coreografia e apresenta duas estratégias para realizá-la. É também apresentado um estudo de caso onde uma coreografia é realizada aplicando as estratégias elencadas. O capítulo termina apresentando uma comparação entre as estratégias. O Capítulo 4 desenvolve uma aplicação coreografada usando a abordagem baseada em eventos descrita no Capítulo 3. O Capítulo 5 apresenta a conclusão e as considerações finais, incluindo trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Microsserviços são uma alternativa à arquitetura monolítica, onde o *software* é construído como uma única unidade inseparável. Através dos microsserviços o *software* é dividido em unidades independentes que se comunicam através da rede para executar as funcionalidades do sistema. No entanto, para que os microsserviços se comportem como um sistema distribuído é necessário definir e implementar estratégias de composição.

Este capítulo apresenta os conceitos de: arquitetura monolítica, arquitetura de microsserviços, composição, coreografia e modelagem de microsserviços.

2.1 ARQUITETURA MONOLÍTICA

Linguagens de programação, de forma geral, dispõem de recursos para decompor a complexidade do *software* em abstrações menores (DRAGONI et al., 2017). Por exemplo, as linguagens que suportam o paradigma de orientação a objetos permitem a implementação de classes e a união dessas classes em pacotes ou *namespaces*. Desenvolvedores geralmente constroem uma única unidade de *software* a partir desses pacotes, ainda que a linguagem de programação ofereça recursos para a decomposição.

A abordagem empregada habitualmente durante o desenvolvimento de *software*, em que o programador implementa todas as funcionalidades em uma única aplicação, ou artefato executável, resulta em um monólito. Monólito é um *software* em que suas abstrações menores não são independentemente executáveis e estão contidas no mesmo processo, fazendo assim com que a aplicação assuma uma arquitetura monolítica (STUBBS; MOREIRA; DOOLEY, 2015).

A Figura 1 ilustra uma arquitetura de *software* monolítica tradicional para o fornecimento de uma aplicação *web*. É importante destacar que embora haja a separação habitual em cliente, servidor de aplicação e banco de dados, a implementação de todas as regras de negócio e execução das funcionalidades do

sistema estão concentradas em um único processo no servidor de aplicação. O servidor irá responder ao cliente e inserir e recuperar informações no banco de dados.

Figura 1 – Arquitetura Monolítica
Fonte: Traduzido e adaptado de FOWLER; LEWIS, 2014.

Uma aplicação de *software* cuja arquitetura seja monolítica possui benefícios durante os estágios iniciais de desenvolvimento. Uma vez que nessa fase a aplicação não exige sua distribuição através da rede, as regras de negócio podem ser inseridas num único artefato executável, tornando a sua construção menos complexa (TRIPOLI; CARVALHO, 2016). Além disso, a implantação da aplicação no servidor também é facilitada, pois será somente necessário realizar o *upload* do artefato executável e suas dependências para o ambiente de execução (DRAGONI et al., 2017). Por fim, no momento em que a aplicação não corresponder com o desempenho desejado, é possível escalar horizontalmente implantando outras instâncias da mesma aplicação em diferentes servidores e acessá-las através de um balanceador de carga (STUBBS; MOREIRA; DOOLEY, 2015).

Em contrapartida, há cenários em que esse estilo arquitetural dificulta o desenvolvimento, como por exemplo a construção de soluções distribuídas e de larga escala (GIARETTA; DRAGONI; MAZZARA, 2017). Dado que as regras de negócio são encapsuladas em um único artefato executável, possivelmente haverá um forte acoplamento e a produtividade do programador será afetada negativamente. Outra desvantagem é que devido ao grande volume de regras de negócio centralizadas no monólito, o rastreamento, a identificação e a correção de problemas exigirão muito mais esforço (DRAGONI et al., 2017). Dificuldades também serão encontradas no gerenciamento de bibliotecas e dependências da aplicação. Tal fato ocorre porque bibliotecas e suas versões podem divergir entre o ambiente de desenvolvimento e o

ambiente de produção. Além disso, as bibliotecas podem conflitar com dependências de outras aplicações implantadas no mesmo ambiente de execução (TOMMASO et al., 2015). Por fim, no momento em que for necessário escalar a aplicação de *software*, a estratégia monolítica se torna imprecisa, pois dificilmente todas as unidades menores contidas no monólito requisitarão o mesmo desempenho, o que resultará em um ambiente subutilizado (OLIVEIRA, 2017).

2.2 ARQUITETURA DE MICROSSERVIÇOS

Grande parte dos desafios encontrados durante o desenvolvimento, manutenção e gerenciamento de aplicações monolíticas decorrem da sua característica principal: um único artefato executável. Uma possível solução é desacoplar as pequenas abstrações e torná-las independentes, como ocorre na arquitetura de microsserviços (KOZHIRBAYEV; SINNOTT, 2017).

A arquitetura de microsserviços surgiu nos últimos anos como uma alternativa às abordagens tradicionais de desenvolvimento de *software*. Nesse estilo de arquitetura a aplicação é composta por uma suíte de serviços pequenos, independentemente executáveis, contidos em seus próprios processos e se comunicando através de mecanismos leves para troca de mensagens (FOWLER; LEWIS, 2014). Dentre suas principais características e benefícios, se destacam:

- Os microsserviços devem possuir limites e responsabilidades bem definidas, favorecendo o baixo acoplamento e alta coesão, seguindo o Princípio da Responsabilidade Única (GUIDI et al., 2017);
- Independência e responsabilidades bem definidas elevam a produtividade da equipe, uma vez que resultam em um código fonte menor e de melhor compreensão, desenvolvimento e manutenção, tanto para desenvolvedores experientes quanto para novos membros da equipe (TRIPOLI; CARVALHO, 2016);
- Microsserviços favorecem a interoperabilidade, posto que a comunicação ocorre através de suas interfaces públicas e mecanismos leves. Dessa forma, diferentemente do monólito construído com uma

única tecnologia, diferentes tecnologias podem ser empregadas para a construção de diferentes microsserviços (DRAGONI et al., 2017);

- Como consequência da sua execução independente, microsserviços podem ser implantados e reiniciados sem impactar diretamente uns aos outros. Essa característica direciona os microsserviços ao uso de *containers* (*conteinerização).* Os *containers* conferem alta liberdade na configuração do ambiente e permite à aplicação se adequar ao que melhor se encaixa para a qualidade do serviço ou custo da infraestrutura (KOZHIRBAYEV; SINNOTT, 2017);
- A escalabilidade de uma aplicação orientada a microsserviços se torna mais eficiente, dado que suas partes são independentes e podem ser escaladas conforme a necessidade aumenta (FOWLER; LEWIS, 2014).

A Figura 2 demonstra a arquitetura de microsserviços alternativa ao monólito ilustrado na Figura 1. Nessa abordagem, cada domínio do *software* pode ser encapsulado em um único serviço independentemente executável. Essa arquitetura, em contraste com o monólito, permite a implementação, implantação e atualização independente de cada serviço.

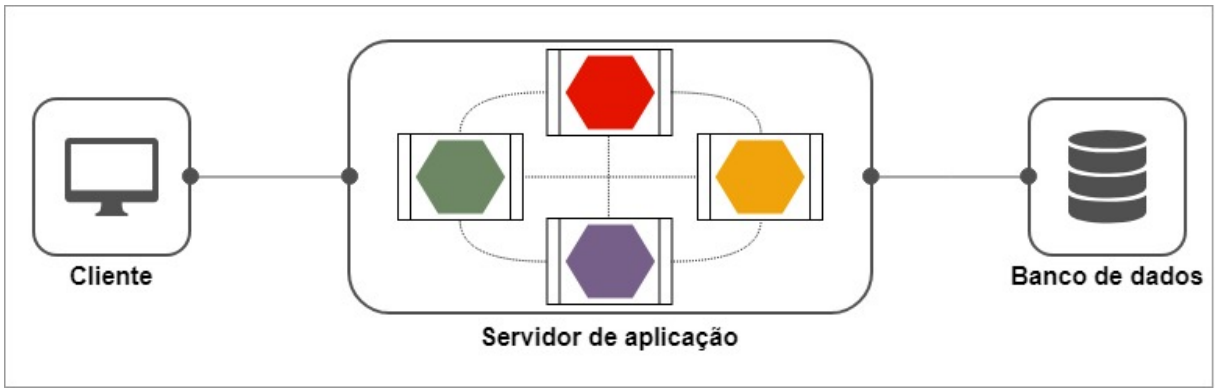

Figura 2 – Arquitetura de Microsserviços
Fonte: Traduzido e adaptado de FOWLER; LEWIS, 2014.

Embora a característica distribuída da arquitetura de microsserviços ofereça subsídios para solucionar muitos desafios impostos pelo monólito, ela apresenta algumas novas dificuldades. A execução em processos diferentes impede que os microsserviços se comuniquem através de invocações de memória, logo, mecanismos para chamadas remotas, sejam elas locais ou através da rede, são necessárias

(DRAGONI et al., 2017). Por exemplo, o Protocolo de Transferência de Hipertexto (*Hypertext Transfer Protocol*, HTTP) é comumente empregado na comunicação síncrona entre os microsserviços, enquanto o Protocolo Avançado de Enfileiramento de Mensagens (*Advanced Message Queuing Protocol*, AMQP) é empregado na comunicação assíncrona (OLIVEIRA, 2017).

Por fim, vale lembrar que microsserviços podem ser considerados como uma evolução da arquitetura orientada a serviços representada pelos serviços web (TRIPOLI; CARVALHO, 2016). Dessa forma, boas práticas e estratégias empregadas na arquitetura orientada a serviços podem ser reaproveitadas no cenário dos microsserviços.

2.3 COMPOSIÇÃO DE MICROSSERVIÇOS

Ao distribuir as regras de negócio através dos microsserviços na rede, a cooperação entre eles se torna fundamental para a execução de funcionalidades mais complexas e elaboradas da aplicação. A composição de microsserviços é obtida através da execução organizada seguindo um fluxo correto de ações a fim de completar uma requisição do cliente, seja ele humano ou outro sistema (GUIDI et al., 2017). Nesse contexto, há duas estratégias que se destacam: a orquestração e a coreografia.

A estratégia de orquestração para a composição de microsserviços é obtida através da implementação de uma entidade central que atua como orquestrador, isto é, um microsserviço que conhece todo o fluxo necessário para alcançar determinada funcionalidade e seus participantes (BRAVETTI; ZAVATTARO, 2007). Uma vez que o fluxo tenha iniciado, o orquestrador envia requisições para cada microsserviço participante e aguarda pelo seu respectivo retorno a fim de compor todas as respostas em uma resposta final para o cliente iniciador. A orquestração, também comum na arquitetura orientada a serviços, é uma estratégia popular e largamente adotada, contudo não está alinhada com os princípios básicos da característica distribuída dos microsserviços. Escolher pela composição por orquestração resulta na concentração de responsabilidades em uma entidade central, que por sua vez aumenta a

dependência de um único elemento e propicia o alto acoplamento da comunicação (DRAGONI et al., 2017).

Por outro lado, a composição por coreografia não requer qualquer implementação central da responsabilidade do controle do fluxo de uma funcionalidade. Essa estratégia define que os serviços devam conhecer minimamente o escopo global e cooperar entre si para determinar quando deverão participar de um determinado fluxo. Essa característica torna a coreografia mais apropriada à arquitetura de microsserviços, pois favorece o baixo acoplamento e a descentralização do *software* (BRAVETTI; ZAVATTARO, 2007).

A Figura 3 ilustra as duas abordagens para composição de microsserviços: orquestração e coreografia. É importante ressaltar a presença do serviço orquestrador na representação à esquerda da Figura 3, que exerce seu papel de através da comunicação com todos os outros serviços. Em contrapartida, a representação à direita ilustra a coreografia através da comunicação entre os próprios serviços.

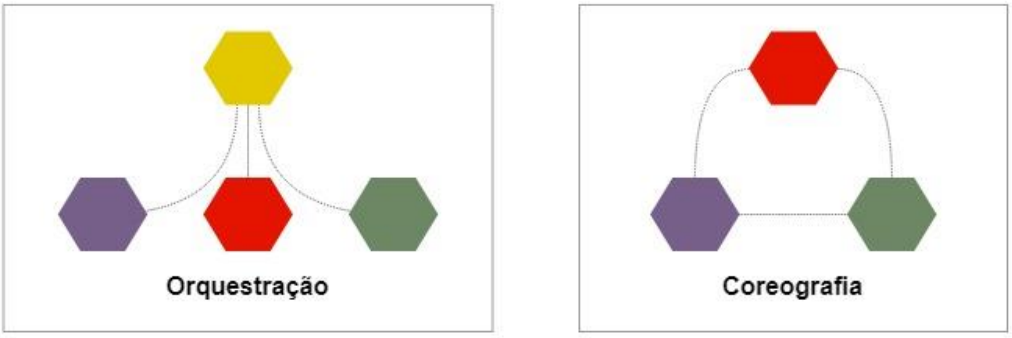

Figura 3 – Diferença entre composição por orquestração e por coreografia
Fonte: Adaptado de GOMES, 2017.

2.4 COREOGRAFIA

A coreografia é um modelo de composição de serviços em que o principal objetivo é proporcionar a colaboração entre membros independentes na execução de processos distribuídos. Nesse modelo, a colaboração se torna factível através da troca de mensagens entre os próprios serviços e, dessa forma, suprime-se a necessidade de um orquestrador para coordenar o fluxo. Portanto, a coreografia é a descrição

global das interações entre os membros integrantes de um sistema distribuído (GOMES, 2017).

Durante o projeto de uma coreografia de microsserviços, três aspectos fundamentais precisam ser delineados: os processos distribuídos, os papéis e os protocolos de comunicação. O processo distribuído é um requisito do *software* que demandará cooperação entre dois ou mais participantes. Cada participante desempenha um papel, isto é, se comporta de tal modo a produzir e, ou, consumir mensagens para colaborar com a composição. Por fim, o protocolo de comunicação estabelece como as mensagens que asseguram a colaboração devem ser trocadas e interpretadas. Uma vez que cada papel foi executado corretamente e houve êxito na composição, afirma-se que houve uma encenação (GOMES, 2017).

Embora a coreografia estabeleça uma descrição global das interações, não é necessário que cada participante tenha ciência de todos os fluxos distribuídos do *software*. Na verdade, é desejado que os participantes conheçam minimamente o escopo global, a fim de responder a estímulos (mensagens) conforme seu domínio e produzir insumos para a continuidade da composição. Dessa forma, diferentes implementações de um mesmo papel podem ser facilmente alternadas conforme necessário ou desejado, seja para buscar um resultado performático ou implementar novas regras de negócios (GOMES, 2017).

Se por um lado a coreografia é benéfica para a colaboração entre serviços ao desacoplar a arquitetura e extrair o serviço orquestrador, por outro o modelo acrescenta obstáculos para a implementação e o gerenciamento do *software*. Como regras de negócios e controles de fluxo para consumo e produção de mensagens estão distribuídos entre os serviços, a complexidade geral do sistema aumenta drasticamente.

Assim como o desenvolvimento de qualquer modelo arquitetural de *software*, é recomendado que a coreografia seja modelada e documentada desde as etapas iniciais do projeto. Logo, sempre que for necessário compreender como determinada encenação é realizada, por exemplo a entrada de novos desenvolvedores na equipe, haverá uma única fonte de informação.

2.4.1 MODELAGEM DE COREOGRAFIA

Coreografias de microsserviços podem ser modeladas através de linguagens específicas. Essas linguagens são divididas em duas categorias (DECKER; KOPP; BARROS, 2008):

- Independentes de implementação: que modelam coreografias na perspectiva de negócios e processos, sem necessariamente especificar detalhes de implementação;
- Específicas de implementação: que modelam coreografias com detalhes técnicos, como definições de protocolos e estrutura das mensagens.

Além da divisão das linguagens em categorias, é possível dividi-las de acordo com a abordagem dos modelos produzidos (DECKER; WESKE, 2007):

- Modelos de interação: as próprias interações entre serviços se tornam os blocos básicos de construção, assim dependências de controle e fluxos de dados são definidos na perspectiva global entre eles. São, no princípio, atômicos e abstratos, para somente depois serem detalhados;
- Modelos de interconexão: são elaborados por participante, ou papel, na coreografia e não ilustram especificamente interações entre serviços, mas sim o recebimento e o envio de mensagens para terceiros.

Há diferenças na forma como as duas abordagens acima representam a modelagem de coreografias e ambas possuem vantagens e desvantagens. O modelo de interação garante melhor visão global das interações entre serviços, no entanto permite a representação de restrições que só podem ser satisfeitas através de comunicações adicionais não existentes no modelo original. Por exemplo, se o serviço C precisa enviar uma mensagem ao serviço D após o serviço A se comunicar com o B, de alguma forma o serviço C precisa saber quando o serviço B recebeu a mensagem. Já o modelo de interconexão garante melhor visão interna do comportamento de cada serviço, embora permita a representação de colaborações incompatíveis. Por exemplo, o modelo permite especificar que o serviço B recebe em dado momento uma mensagem de um serviço A, mas este último serviço nem sequer existe ou envia tal mensagem.

O *Object Management Group* propôs a linguagem gráfica Modelo e Notação de Processos de Negócios (*Business Process Modeling Model*, BPMN), que oferece notação em alto nível para modelar a composição de papeis na coreografia. A BPMN pertence à categoria das linguagens independentes de implementação e garante modelagem de papeis sem requerer detalhes técnicos de implementação. Como resultado, é possível focar totalmente no relacionamento e colaboração entre os serviços durante a etapa de modelagem.

A linguagem BPMN possui 2 versões e a abordagem dos modelos produzidos pode ser diferente entre elas. A versão 1, publicada em maio de 2004, oferece apenas recursos para especificação de coreografias através da interconexão de interfaces, ou seja, o modelo é restrito ao papel e não ilustra realmente como a colaboração acontece. Já a versão 2, publicada em janeiro de 2011, possui novas abstrações para permitir a especificação de coreografias através das próprias interações entre os papeis, o que garante a visão global da colaboração.

Desde os seus primeiros lançamentos, houve grande aceitação dessa linguagem tanto pela academia quanto pela indústria. Atualmente, BPMN é o padrão *de facto* para modelar graficamente processos de negócios. A Figura 4 apresenta os principais elementos gráficos da linguagem BPMN versão 2 para modelagem de coreografias de microsserviços.

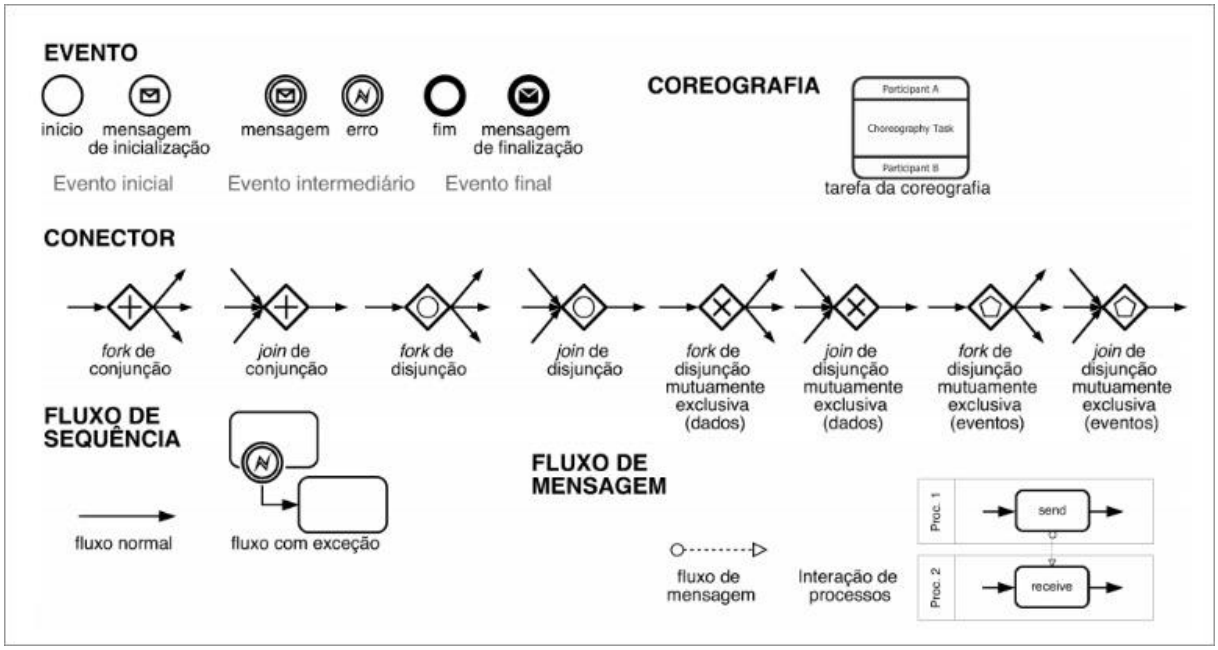

Figura 4 – Principais elementos gráficos da linguagem BPMN versão 2

Fonte: GOMES, 2017.

Como é possível observar na Figura 4, a linguagem BPMN oferece elementos visuais para a modelagem e divide-os em cinco grupos: eventos, coreografias, conectores, fluxos de sequência, fluxos de mensagem. Os eventos permitem especificar o início e o fim da coreografia, além da recepção de mensagens. Os elementos do grupo coreografia permitem modelar uma tarefa coreografada. Essa tarefa é composta por dois ou mais participantes, especificados nas extremidades do elemento, e um nome, especificado no centro. Os conectores possibilitam dividir ou unir duas ou mais tarefas coreografadas, indicando ações paralelas e/ou sequenciais. Por fim, os fluxos de sequência e de mensagem permitem modelar a direção das mensagens.

3 COREOGRAFIA DE MICROSSERVIÇOS

Coreografia de microsserviços é um modelo de composição de serviços que proporciona a colaboração entre serviços independentes na execução de processos distribuídos. Nesse modelo, a colaboração ocorre através da troca de mensagens entre os serviços. As trocas de mensagens entre serviços podem ser simples, mas normalmente tendem a se tornar muito complexas (GOMES, 2017). Nesse contexto, duas estratégias para a coreografia de microsserviços se destacam: a coreografia baseada em eventos e a programação coreográfica.

Este capítulo apresenta as duas estratégias para a coreografia de microsserviços, duas tecnologias para a implementação dessas estratégias e um estudo de caso.

3.1 COREOGRAFIA BASEADA EM EVENTOS

O paradigma de notificação de eventos surge como uma estratégia para promover o planejamento e a implementação de coreografias de microsserviços. Nessa estratégia, membros de uma coreografia atuam como produtores e/ou consumidores de determinados eventos que são compartilhados com outros membros interessados. Sempre que um membro precisa notificar outro, ele simplesmente publica um evento sem conhecer a identidade dos possíveis consumidores (CIANCIA et al., 2010). Dessa forma, a comunicação de vários membros e a execução de diversas pequenas funcionalidades distribuídas podem resultar da notificação de um único evento.

A coreografia baseada em eventos se concentra em como cada membro se comporta na ocorrência de eventos. Naturalmente, a adoção desse paradigma diminui as interdependências entre as implementações dos membros, garantindo um baixo acoplamento. Do mesmo modo, os membros podem ser implantados em diferentes servidores e contextos e se comunicar somente através da publicação e consumo de eventos (CIANCIA et al., 2010). Além disso, o modelo baseado em eventos facilita a

implementação de sistemas distribuídos em que os membros estão sujeitos a mudanças frequentes.

Tradicionalmente as abordagens do paradigma de notificação de eventos baseiam-se em comunicações mediadas. Nessa abordagem, um mediador, ou um serviço terceiro, se torna responsável pelo roteamento da comunicação entre os membros. A mediação é empregada através da recepção de notificações, gerenciamento de eventos e sua distribuição aos respectivos consumidores interessados (CIANCIA et al., 2010). É importante ressaltar que embora a abordagem induza a associação do mediador a um orquestrador, a sua responsabilidade se resume a notificar consumidores sobre eventos publicados por produtores. A Figura 5 ilustra a diferença entre o relacionamento dos membros de uma coreografia com comunicação mediada e o relacionamento par a par de uma coreografia com comunicação não mediada.

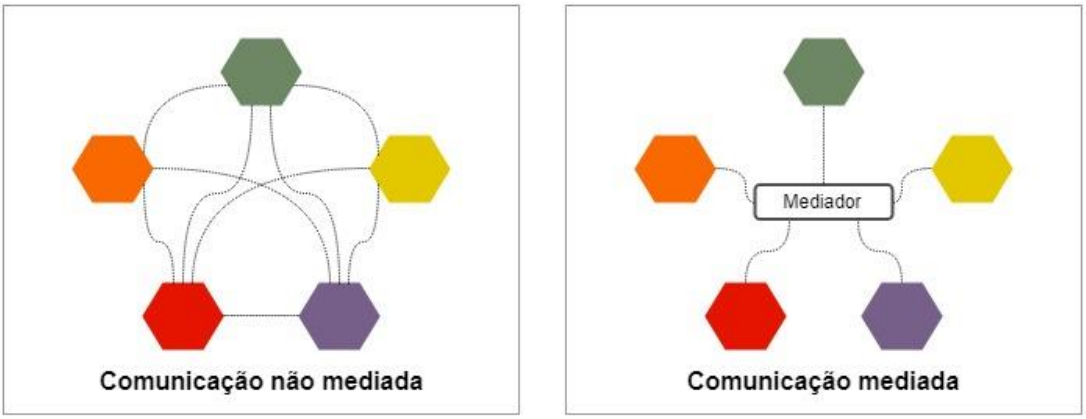

Figura 5 – Diferença entre comunicação mediada e não mediada
Fonte: Traduzido e adaptado de D'AMORE, 2015.

Conforme apresenta a Figura 5, do lado esquerdo a comunicação não mediada conta com a comunicação direta entre os membros da coreografia. A ligação entre os membros se faz necessária sempre que há comunicação entre eles. Em contrapartida, do lado direito há um mediador que conhece quais consumidores estão registrados para receber os eventos publicados pelos produtores.

Alguns mecanismos de comunicação mediada apresentam inteligência significativa. Um bom exemplo é o *Enterprise Service Bus*, que normalmente emprega implementações sofisticadas para roteamento, manipulação e filtro de mensagens e aplicação de regras de negócio. Embora soem como boa escolha, esses mecanismos

ocultam detalhes de domínio e diminuem a coesão do microsserviço. Nesse sentido, a comunidade de coreografia de microsserviços recomenda uma abordagem alternativa: membros inteligentes e mediadores burros. Logo, ao projetar uma arquitetura de comunicação mediada, todas as regras inerentes aos membros são contidas neles mesmos (FOWLER; LEWIS, 2014).

Sistemas de mensagens despontam como uma excelente alternativa para mediar comunicações em coreografias baseadas no paradigma de notificação de eventos. Esses sistemas são serviços executados de forma independente do *software* e permitem a entrega e recepção de mensagens entre serviços distribuídos sem que esses se conheçam diretamente. Sistemas de mensagens são divididos em dois modelos (FUNDAÇÃO DE SOFTWARE APACHE, 2017):

- Filas: os eventos são recebidos pelo sistema e salvos em uma fila, para que um conjunto de serviços consumam alternadamente. Essa característica permite a divisão do processamento de dados entre diversos consumidores sem duplicidade de leitura. Ou seja, a distribuição de um mesmo evento para mais de um consumidor não é possível e o evento desaparece após ser consumido pela primeira vez;
- Publicação-Assinatura (*publish-subscribe*): os eventos são recebidos pelo sistema e retransmitidos para todos os consumidores registrados para receber tais eventos. Essa característica permite que diversos membros executem seu papel na coreografia através de um único evento. Por outro lado, escalar o processamento deixa de ser trivial, pois todos os eventos vão para todos os assinantes.

No contexto da coreografia baseada em eventos, sistemas de mensagens que se enquadrem no modelo publicação-assinatura se sobressaem. Nesse modelo, a encenação de uma coreografia pode envolver mais de dois papéis, e, ainda, o estímulo requisitado para que determinados membros exerçam suas funções pode ser o mesmo. A plataforma Apache Kafka (KREPS; NARKHED; RAO, 2011) é uma solução para sistemas de mensagens que se enquadram no modelo publicação-assinatura e será descrita a seguir.

3.1.1 PLATAFORMA APACHE KAFKA

O LinkedIn[1] é a maior rede social profissional na Internet e conta com mais de 500 milhões de membros que, diariamente, geram bilhões de registros e eventos representando suas atividades. Todos esses dados alimentam seus sistemas internos e possibilitam funcionalidades, como por exemplo “Quem viu seu perfil”, “Pessoas que talvez você conheça” e “Ofertas de empregos recomendadas”. Em uma rede social como a do LinkedIn há a necessidade de uma solução para a distribuição de eventos e integração de um volume intenso de dados online. Nesse contexto, em 2010 foi projetada e desenvolvida pela empresa a plataforma Kafka[2], que desde outubro de 2012 é um projeto *open source* da Fundação de Software Apache (WANG et al., 2015).

O Apache Kafka é uma plataforma escalável de mensagens baseada no padrão de publicação-assinatura tendo como sua arquitetura principal um registro distribuído. A plataforma oferece ainda recursos para a construção de sistemas distribuídos e em tempo real de forma confiável, tolerante a falhas e em alta performance. Suas três principais funcionalidades são (FUNDAÇÃO APACHE KAFKA, 2017):

1. Publicação e inscrição em *stream* de dados, semelhante a filas de mensagens;
2. Armazenamento de *stream* de dados de forma durável e tolerante a falhas;
3. Processamento de *stream* de dados conforme eles ocorrem.

Para oferecer essas funcionalidades e ao mesmo tempo ser uma alternativa relevante frente aos sistemas de mensagens já existentes, o Apache Kafka adota as seguintes características (GARG, 2013):

- Mensagens persistentes: para possibilitar a distribuição de eventos não pode haver perda de dados. Além de garantir o armazenamento das mensagens, o Kafka emprega uma estratégia de persistência que garante tempo constante de leitura;

[1]LinkedIn. Disponível em <https://linkedin.com>. Acesso em nov. 2018.
[2]Apache Kafka. Disponível em <https://kafka.apache.org>. Acesso em nov. 2018.

- Alta taxa de transferência: o Apache Kafka foi projetado para lidar com milhões de mensagens por segundo;
- Distribuído: a plataforma suporta explicitamente o particionamento horizontal em *clusters* distribuindo o consumo entre servidores;
- Suporte a múltiplos clientes: para amplificar a sua adoção e facilitar o uso, há clientes para mais de 15 linguagens de programação, como por exemplo Java, JavaScript, Ruby, PHP e Python;
- Tempo real: as mensagens publicadas se tornam imediatamente visíveis aos inscritos.

Conforme apresenta a Figura 6, o Apache Kafka possui três conceitos chave: tópico, produtor e consumidor. Um tópico é um *stream* de mensagens de um tipo específico em ordem cronológica. Um produtor é uma aplicação, ou processo, que publica uma mensagem em um determinado tópico. Semelhantemente, um consumidor é uma aplicação, ou processo, que se inscreve em um ou mais tópicos e consome as mensagens lá publicadas. A Figura 7 exemplifica como os três conceitos se relacionam dentro da plataforma Kafka. Há dois produtores que publicam mensagens no mesmo tópico e dois consumidores que consomem as mensagens desse tópico.

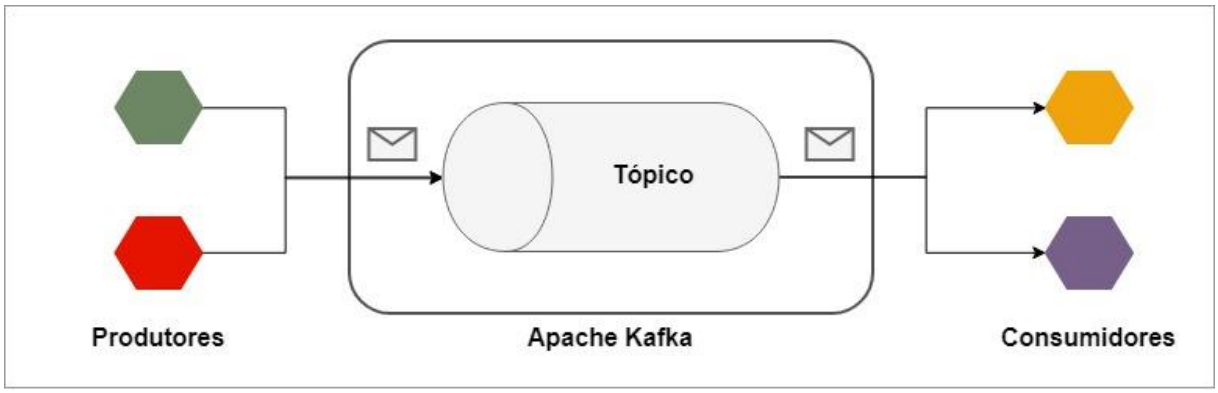

Figura 6 – Arquitetura básica da plataforma Apache Kafka
Fonte: Traduzido e adaptado de GARG, 2013.

Sistemas de mensagens já existem há muito tempo e desempenham um papel crítico para a distribuição de eventos e processamento assíncrono de fluxo de dados. É evidente que o sistema de mensagens oferecido pela plataforma Apache Kafka pode ser comparado com os diversos outros já existentes, nos quais se

destacam o RabbitMQ[3] e o ActiveMQ[4]. No cenário da coreografia de microsserviços baseado em eventos, a distribuição das mensagens deve ocorrer com baixa latência a fim de não retardar a encenação. Logo, ainda que o Apache Kafka tenha funcionalidades comparáveis aos sistemas existentes, a sua performance é extremamente superior, como pode ser analisado na Figura 7.

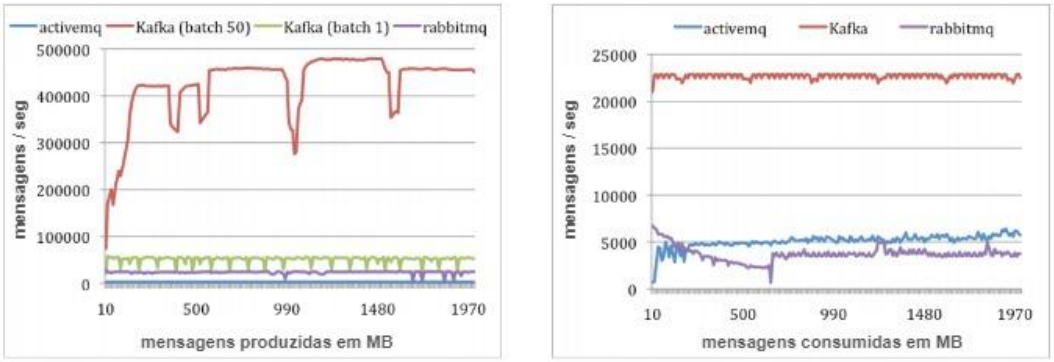

Figura 7 – Comparação de performance entre Apache Kafka e outros sistemas de mensagens
Fonte: Traduzido de KREPS; NARKHED; RAO, 2011.

A Figura 7 ilustra a comparação da performance entre o Apache Kafka, RabbitMQ e ActiveMQ. Nos dois gráficos o eixo x representa a quantidade de dados enviados/consumidos em MB e o eixo y o total de mensagens por segundo. Do lado esquerdo há a comparação da produção de mensagens, o Kafka foi testado publicando lotes de 1 única mensagem e lotes com 50 mensagens. Nesse cenário, o Kafka publicou mensagens a uma taxa de 50.000 e 400.000 mensagens por segundo para lotes de 1 e 50 mensagens, respectivamente. Já do lado direito há a comparação no consumo de mensagens. Nesse cenário, o Kafka consumiu em média 22.000 mensagens por segundo, um desempenho aproximadamente 4 vezes maior do que as outras ferramentas alternativas.

Ao prover funcionalidades cada vez mais requisitadas no dia a dia de grandes aplicações *web* com alta performance, sua adoção tende a ser extensa. Os principais casos de uso são (RAO; KREPS, 2016):

- Airbnb: *stream* de eventos e parte da arquitetura para rastreamento de exceções;

[3]RabbitMQ. Disponível em < https://www.rabbitmq.com>. Acesso em nov. 2018.
[4]ActiveMQ. Disponível em <http://activemq.apache.org>. Acesso em nov. 2018.

- CloudFlare: processamento de *logs*, coleta e distribuição de bilhões de eventos em centenas de servidores;
- Coursera: métricas em tempo real e distribuição de dados para *dashboards;*
- Netflix: Monitoramento em tempo real e processamento de eventos;
- Spotify: Parte do sistema de entrega e armazenamento de *logs*;
- Twitter: Parte da infraestrutura de processamento de *tweets* e dados de usuários;

Outra estratégia para aplicação da coreografia é a Programação Coreográfica, descrita a seguir.

3.2 PROGRAMAÇÃO COREOGRÁFICA

A adoção de uma estratégia de coreografia de microsserviços possui muitos obstáculos. Toda a coreografia pode ser comprometida simplesmente pela implementação incorreta de envio ou recebimento de eventos em um único membro. Entre os principais erros, os mais comuns são: *deadlock* e condição de corrida. O primeiro erro ocorre na espera indefinida por um recurso computacional e interfere no comportamento do sistema, levando-o a não responder. O segundo ocorre em cenários com acessos simultâneos a um recurso compartilhado e leva a computações com valores e retornos incorretos. Nesse contexto, afirma-se que a má comunicação entre os membros é uma falha de confiabilidade (MONTESI, 2013).

Linguagens de programação tradicionais não oferecem recursos para implementar a interação entre os membros de uma coreografia baseada na perspectiva global. Todas as comunicações modeladas são implementadas diretamente na perspectiva individual de cada microsserviço. Essa característica pode levar à construção de *software* não confiável. Por exemplo, um *deadlock* pode ocorrer quando um microsserviço produz determinado evento somente após o recebimento de um outro, no entanto nunca o recebe.

A Programação Coreográfica é um paradigma de programação apresentado em 2013 e surge como estratégia de coreografia de microsserviços. Nesse paradigma, a especificação de coreografia é representada pela formalização das interações esperadas entre os membros a partir da perspectiva global (MONTESI, 2014). Por exemplo, a formalização abaixo representa a interação entre dois papeis: *Alice* e *Bob*. *Alice* envia um livro ao *Bob*, que por sua vez retribui em dinheiro à *Alice*.

$$Alice \rightarrow Bob : livro \,;\, Bob \rightarrow Alice : dinheiro$$

A especificação de coreografia será sempre correta por *design*, pois descreve formalmente o fluxo de comunicação esperado em um *software*. E, por conseguinte, a coreografia pode ser compilada em implementações locais para cada microsserviço, provendo as ações de envio e recebimento que cada um deverá realizar. Essas implementações serão sempre corretas por construção. A Figura 8 ilustra o processo de compilação, nomeado Projeção de *Endpoint*. Conforme apresentado na figura, a especificação de coreografia é utilizada como entrada no processo de projeção de *endpoint*. Esse processo compila a especificação e projeta um serviço executável, apresentado à direita.

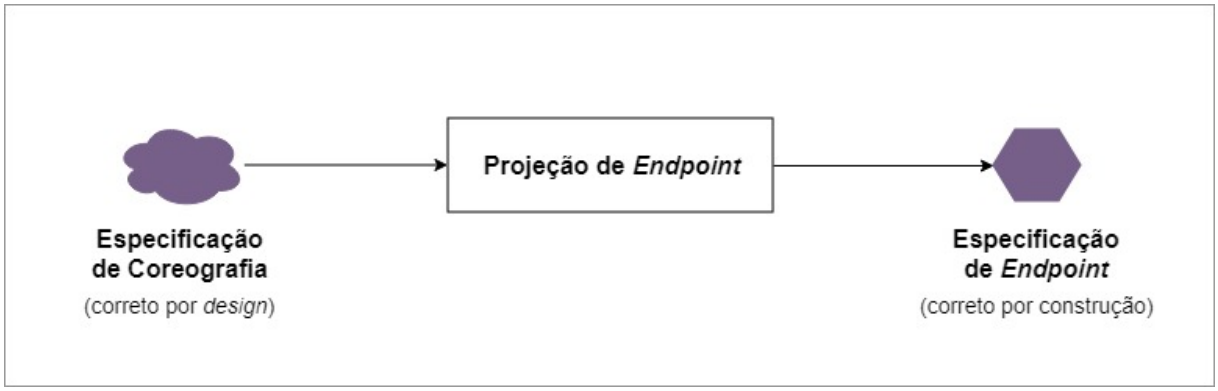

Figura 8 – Metodologia de projeção de *endpoint*
Fonte: Traduzido e adaptado de MONTESI, 2014.

Os microsserviços (*endpoints*) projetados serão sempre corretos por construção, já que são verificados por meio de métodos formais. Como resultado, é obtido um sistema distribuído livre de *deadlocks* e as interações não são mais implementadas separadamente na perspectiva de cada papel da coreografia. Após o

processo de projeção, naturalmente, cada microsserviço pode ser editado e ter suas regras de negócio e de domínio implementadas.

Atualmente há apenas uma linguagem de programação que implementa o novo paradigma, a linguagem Chor (MONTESI, 2013). A linguagem foi proposta em conjunto com o paradigma pelo mesmo autor e ainda é classificada como um protótipo.

3.2.1 LINGUAGEM DE PROGRAMAÇÃO CHOR

Chor[5] é uma linguagem de programação que adota o paradigma de Programação Coreográfica e possibilita a especificação de coreografias para sistemas distribuídos. A linguagem oferece recursos para a especificação da coreografia em um único programa a partir de uma perspectiva global. Não somente, ainda permite a projeção do conjunto de serviços que poderão ser executados independentemente a fim de encenar a coreografia (MONTESI, 2013).

Na linguagem Chor, o recurso de especificação permite que coreografias sejam formalizadas e, principalmente, que as comunicações entre os membros respeitem protocolos previamente especificados. Caso existam inconsistências na especificação formalizada, erros são relatados utilizando realce de sintaxe. Há também o recurso de projeção o qual permite que, ao finalizar a especificação da coreografia, endpoints sejam projetados para a sequência das atividades de implementação. A Figura 9 ilustra o ciclo de vida tradicional de coreografias nesse contexto, desde a sua especificação na linguagem Chor até a execução do microsserviço (MONTESI, 2013).

[5]Chor. Disponível em <http://www.chor-lang.org>. Acesso em nov. 2018.

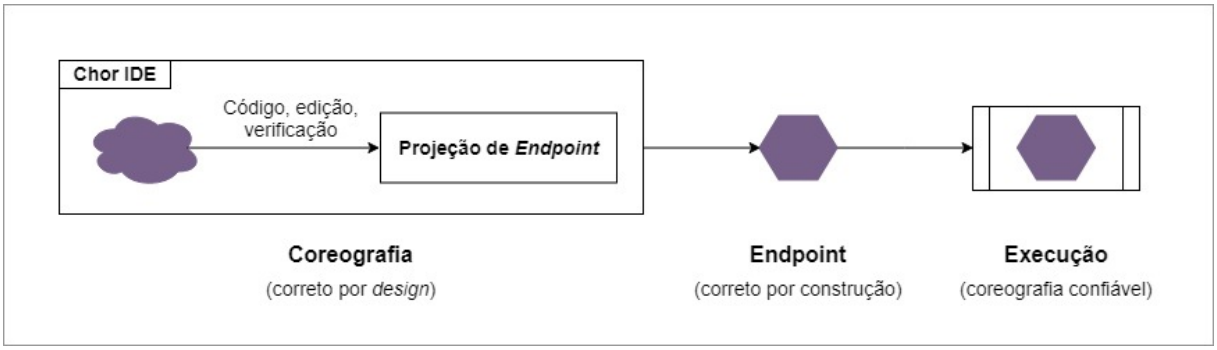

Figura 9 – Metodologia de desenvolvimento na linguagem Chor
Fonte: Traduzido e adaptado de MONTESI, 2013.

Conforme ilustrado na Figura 9, o processo de especificação de coreografia e projeção de *endpoint* (apresentado na Figura 8) ocorre através da extensão Chor. A partir da sua projeção, o *endpoint* se torna disponível para execução independente. É importante ressaltar que a especificação da coreografia é garantidamente correta na especificação, o que resulta em um serviço correto e, principalmente, em uma coreografia confiável.

Programas implementados pela linguagem Chor são compostos pelos seguintes elementos: identificação, preâmbulo e corpo. A identificação permite declarar um nome para a coreografia para aumentar a legibilidade do código. O preâmbulo, por sua vez, é a composição da definição do protocolo de comunicação e a criação do canal de comunicação. Por fim, o corpo é a especificação do comportamento de cada membro da coreografia (MONTESI, 2013).

No Chor, todas as comunicações entre os membros da coreografia ocorrem através de sessões, assim como em serviços *web* tradicionais, e são modeladas como tipos básicos na linguagem. As sessões implementam protocolos, que especificam a ordem e o tipo das interações. A coreografia formalizada a seguir, onde *Alice* envia uma mensagem "*hi*" ao *Bob*, pode ser implementada na linguagem Chor como mostrado no Algoritmo 1.

$$Alice \rightarrow Bob : hi$$

Algoritmo 1

```
1 program MyProgram;
2
```

```
 3 protocol MyProtocol
 4 {
 5     A -> B : hi(string)
 6 }
 7
 8 public myChannel : MyProtocol
 9
10 main
11 {
12     alice[A] start bob[B] : myChannel(s);
13     alice.”Hello” -> bob.message : hi(s)
14 }
```

Na linha 1 o programa *MyProgram* é identificado. Entre as linhas 3 e 6 o protocolo *MyProtocol* é definido, estabelecendo uma interação entre dois papeis. Nesse caso, o papel *A* deve enviar uma *string* ao papel *B* através da operação *hi*. Na linha 8 o canal de comunicação *myChannel* é criado e relacionado ao protocolo definido anteriormente. Entre as linhas 10 e 14 ocorre, de fato, a implementação da coreografia. Na linha 12 o serviço *alice* (implementando o papel *A*) inicializa o serviço *bob* (implementando o papel *B*) e cria uma sessão *s* através do canal de comunicação *myChannel*. Na linha 13 o serviço *alice* opera de acordo com a definição do protocolo. No fim, após a execução dos dois serviços projetados, a coreografia foi encenada.

A adoção da linguagem como estratégia para a coreografia de microsserviços garante quatro principais benefícios (MONTESI, 2014):

- Clareza: pode ser difícil entender como papeis distribuídos interagem entre si, devido a necessidade de combinar todas as suas ações de comunicação, tanto para entrada quanto para saída, manualmente. Chor permite especificar todos os comportamentos explicitamente;
- Produtividade: desenvolver protótipos de sistemas distribuídos complexos pode ser um processo longo, principalmente quando são muitos papeis envolvidos na coreografia. A linguagem permite definir o

comportamento de todos os papeis em um único programa e, também, gerar o código fonte para cada serviço automaticamente;

- *Deadlock*: a implementação de programas concorrentes é propensa a erros de *deadlock* e eles são difíceis de detectar. Coreografias especificadas em Chor são estaticamente verificadas e livres de *deadlocks*;
- Protocolos: protocolos com vários participantes podem ser difíceis de implementar, acabam sendo especificados informalmente. Chor permite especificar o protocolo da coreografia e, ainda, verifica se a implementação está em conformidade.

Apesar de ser uma linguagem, o Chor é disponibilizado atualmente apenas como um complemento para o ambiente de desenvolvimento integrado Eclipse[6]. No entanto, a falta de uma versão *standalone* não impede a entrega de todos os recursos necessários para a programação de coreografias. Segundo os autores do projeto, há trabalhos em andamento para a disponibilização de uma versão própria em 2019 (MONTESI, 2018).

Embora ofereça os recursos esperados em uma linguagem que adote o paradigma de Programação Coreográfica, o projeto ainda é classificado como um protótipo. Dessa forma, o Chor ainda carece de mecanismos avançados para viabilizar a sua adoção na construção de um sistema distribuído de larga escala. Por exemplo, a linguagem ainda é limitada a tipos e estruturas de dados simples, como inteiros e *strings*, e não possui sistema de depuração integrado. Além disso, há somente suporte para a projeção de serviços com código fonte na linguagem de programação Jolie[7] (MONTESI, 2014).

Ainda que não seja uma alternativa completamente viável para a coreografia de grandes sistemas, a linguagem é promissora e está em plena evolução. O projeto possui código fonte aberto e é livre para uso. Além disso, a equipe acolhe bem novos colaboradores e possui um repositório no GitHub[8] para facilitar a contribuição de terceiros.

[6]Eclipse. Disponível em <https://www.eclipse.org/ide>. Acesso em nov. 2018.
[7]Jolie. Disponível em <https://www.jolie-lang.org>. Acesso em nov. 2018.
[8]Repositório no GitHub. Disponível em <https://github.com/chorlang>. Acesso em nov. 2018.

3.3 ESTUDO DE CASO

Para analisar a diferença entre as duas estratégias exploradas nas seções anteriores, especificamente Coreografia Baseada a Eventos e Programação Coreográfica, uma coreografia simples foi proposta. A coreografia foi definida e modelada utilizando a linguagem de modelagem BPMN 2.0, para então, a partir do modelo gerado, ser implementada por cada alternativa técnica de cada estratégia.

Uma das implementações representa a construção necessária para adotar a estratégia de notificação de eventos através da Plataforma Apache Kafka. Por outro lado, a outra representa a implementação da coreografia no paradigma de Programação Coreográfica utilizando a linguagem Chor.

A figura 10 representa o modelo BPMN resultante do processo de modelagem. O modelo apresenta os papeis *Introducer* e *Greeter*. A coreografia é iniciada com a requisição de saudação (*Greet request*) pelo *Introducer*, que envia uma *introduction* ao *Greeter*. O *Greeter*, por sua vez, ao receber uma *introduction*, responde (*Greet send*) com uma *greeting*.

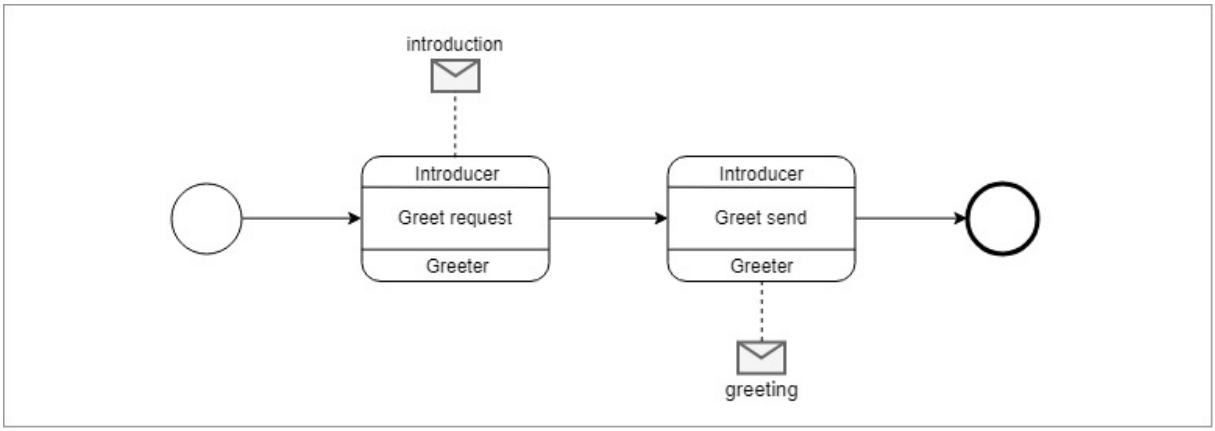

Figura 10 – Coreografia para estudo de caso

Fonte: Autoria própria.

O modelo apresenta dois papeis que desempenham ações de comunicação de entrada e de saída. No momento em que o papel iniciante receber a saudação, a coreografia é considerada encerrada e encenada. A seguir o modelo é implementado tanto na plataforma Apache Kafka, de acordo com o paradigma de notificação de

eventos, quanto na linguagem Chor, de acordo com o paradigma de programação coreográfica.

3.3.1 IMPLEMENTAÇÃO COM A PLATAFORMA APACHE KAFKA

Além de oferecer implementações em diferentes linguagens de programação, a Plataforma Apache Kafka distribui arquivos executáveis para o gerenciamento do serviço de mensagens. A implementação dos membros da coreografia proposta foi utilizada a linguagem de programação JavaScript[9], seu ambiente de execução Node.js[10] e a implementação Kafkajs[11].

A inicialização da Plataforma Apache Kafka foi realizada a partir do arquivo executável distribuído juntamente com a própria plataforma:

```
${kafka_dir}/bin/kafka-server-start.sh config/server.properties
```

Após a execução do arquivo executável acima (onde `${kafka_dir}` representa o caminho absoluto para o diretório no qual a instalação da plataforma se encontra), o serviço estará disponível na sua porta padrão 9092. Uma vez que o serviço esteja em execução, consumidores e produtores podem se conectar e iniciar comunicações.

Para a comunicação entre os membros da coreografia apresentada no estudo de caso, dois tópicos são necessários, um para a requisição de saudação, *`greet-request`*, e outro para o envio da saudação, *`greet-send`*. Na implementação Kafkajs, os tópicos nos quais os consumidores e produtores se conectam são criados sob demanda, logo não foi necessário criá-los antecipadamente.

Uma das desvantagens da arquitetura de microsserviços é a redundância de código fonte e essa característica pode ser observada neste cenário. Dado que os dois papeis presentes na coreografia atuarão como consumidores e produtores, as implementações serão semelhantes para cada um. Para evitar esse problema uma

[9]JavaScript. Disponível em: <https://developer.mozilla.org/en-US/docs/Web/JavaScript>
[10]Node.js. Disponível em <https://nodejs.org/en>
[11]Kafkajs. Disponível em <https://www.npmjs.com/package/kafkajs>

única implementação para consumir e produzir mensagens na plataforma Apache Kafka foi especificada e compartilhada com cada serviço.

A classe *Producer* (Apêndice A) é responsável por fornecer instâncias de produtores para o Apache Kafka. É importante destacar o método construtor, entre as linhas 4 e 11, que define a conexão com o serviço já inicializado e disponível na porta 9092. Além disso, o método *publish*, entre as linhas 13 e 17, recebe como parâmetro uma mensagem e a publica no sistema de mensagens.

A classe *Consumer* (Apêndice B) é responsável por fornecer instâncias de consumidores para o Apache Kafka. O método construtor é o mesmo da classe anterior, dessa forma foi ocultado. A classe implementa o método *subscribe*, entre as linhas 6 e 19, que recebe como parâmetro o tópico de onde as mensagens serão consumidas, uma chave (para caso haja a necessidade de filtrar e responder a uma mensagem específica) e uma função *onMessage* para *callback*, que será invocada sempre que uma mensagem de interesse for recebida.

Com as classes *Producer* e *Consumer* disponíveis, é possível implementar o comportamento de um serviço que desempenhe o papel *Greeter* de acordo com o Algoritmo 2:

Algoritmo 2

```
1  // Hidden Consumer and Producer imports
2
3  const consumer = new Consumer();
4  const producer = new Producer();
5
6  consumer.subscribe({
7    topic: 'greet-request',
8    onMessage({ key, value }) {
9      producer.publish({
10       topic: 'greet-send',
11       message: {
12         key,
13         value: `Hi, ${value}`,
14       },
```

```
15     });
16   },
17 });
```

No código do Algoritmo 2, o comportamento do serviço é implementado entre as linhas 6 e 17. É importante destacar que primeiro ocorre a inscrição no tópico *greet-request* e, como resposta a cada mensagem recebida, há a publicação da saudação no tópico *greet-send*.

Similarmente, com as classes disponíveis, é possível implementar o comportamento de um serviço que desempenhe o papel *Introducer* da seguinte forma:

Algoritmo 3

```
1 // Hidden Consumer and Producer imports
2
3 const consumer = new Consumer();
4 const producer = new Producer();
5
6 producer.publish({
7   topic: 'greet-request',
8   message: {
9     key: 'message-key-0',
10    value: 'Alice',
11  },
12 });
13
14 consumer.subscribe({
15   topic: 'greet-send',
16   filteredKey: 'message-key-0',
17   onMessage(greeting) {
18     console.log(greeting);
19   },
20 });
```

No código fonte do Algoritmo 3, o comportamento do serviço é implementado em duas etapas. Na primeira etapa, entre as linhas 6 e 12, ocorre a publicação da requisição de saudação no tópico *greet-request*. Já na segunda etapa, entre as linhas 14 e 20, ocorre a inscrição no tópico *greet-send* para consumir a saudação recebida. É importante destacar o uso de uma chave, através da palavra-chave *key*, que possibilita filtrar somente essa requisição no tópico.

3.3.2 IMPLEMENTAÇÃO COM A LINGUAGEM CHOR

Como apresentado na Seção 3.2.1, a coreografia de um sistema distribuído composto por dois ou mais microsserviços é especificada na linguagem Chor em um único arquivo. Essa característica garante a perspectiva global das interações durante a implementação do sistema. A construção da coreografia modelada e apresentada na Figura 10 representa muito bem isso, e o resultado é o Algoritmo 4 a seguir.

Algoritmo 4

```
 1 program Greeting;
 2
 3 protocol GreetingProtocol
 4 {
 5     I -> G : greetRequest(string);
 6     G -> I : greetSend(string)
 7 }
 8
 9 public channel : GreetingProtocol
10
11 main
12 {
13     s1[I] start s2[G] : channel(s);
14     s1.”Alice” -> s2.name : greetRequest(s);
15     s2.(“Hi, ” + name) -> s1.greeting : greetSend(s)
```

```
16 }
```

A definição do protocolo, que ocorre entre as linhas 3 e 7, formaliza quantos e quais serão os papeis na coreografia e como eles interagem entre si. A construção da coreografia no corpo no algoritmo, entre as linhas 11 e 16, implementa de forma fixa o envio da introdução "*Alice*" e o retorno, também fixo, "*Hi, Alice*". Esse algoritmo é então utilizado como parâmetro de entrada para o processo de projeção de *endpoint* e os respectivos serviços são gerados.

A partir do momento em que os *endpoints* são gerados, as regras de negócio ou especificidades de domínio podem ser implementadas diretamente no código fonte do microsserviço. Por exemplo, o nome enviado pode ser lido em um arquivo presente no sistema de arquivos do sistema operacional ou consultado em um sistema gerenciador de banco de dados.

3.3.3 COMPARAÇÃO ENTRE AS ESTRATÉGIAS DE COREOGRAFIA

O estudo de caso apresentado neste capítulo mostra que a mesma coreografia pode ser implementada e encenada perfeitamente em qualquer uma das duas estratégias introduzidas. No entanto, cada estratégia possui suas características e a sua seleção depende do cenário no qual ela será empregada. O Quadro 1 apresenta a comparação entre as características das duas estratégias.

Característica	**Baseada em eventos**	**Programação coreográfica**
Mecanismo	Comunicação	Programação
Perspectiva	Individual	Global
Relacionamento dos membros	Comunicação mediada	Comunicação ponto a ponto
Maturidade	Alta	Baixa
Facilidade de manutenção	Alta	Baixa
Documentação	Completa	Apenas a sua tese
Linguagens de programação suportadas	Mais de 15 linguagens	Apenas Jolie

Casos de estudo	Diversos casos triviais e complexos	Quantidade limitada e trivial

Quadro 1 – Comparação entre as estratégias de coreografia

Fonte: Autoria própria.

Analisando a comparação exposta no Quadro 1, considera-se que a estratégia baseada em eventos é a melhor escolha hoje para a coreografia de microsserviços. Essa decisão se deve ao fato da estratégia ser mais madura para realizar implementações, possibilitar a interoperabilidade entre diferentes linguagens de programação, apresentar diversos casos de estudo e documentação detalhada. Ainda assim, é importante destacar que a estratégia baseada em programação coreográfica possui atributos intrinsecamente relacionados com a coreografia de microsserviços e se posiciona como forte alternativa e referência para o futuro.

No Capítulo 4 apresenta-se a implementação de uma arquitetura de microsserviços que adota a estratégia de coreografia baseada em eventos.

4 IMPLEMENTAÇÃO DE COREOGRAFIA BASEADA EM EVENTOS

A estratégia de coreografia de microsserviços baseada em eventos mostrou-se uma alternativa madura e de alto desempenho para uso em casos reais. Este capítulo apresenta a implementação de um protótipo de sistema distribuído utilizando microsserviços e a coreografia baseada em eventos através da plataforma Apache Kafka.

4.1 DESCRIÇÃO

O Crypsense é uma plataforma distribuída em desenvolvimento e pretende ser uma ferramenta auxiliar para compra e venda de ativos digitais. Seu objetivo será oferecer aos usuários recursos avançados para suprir as necessidades não atendidas pelas corretoras do mercado de ativos digitais. Entre os recursos, se destacam:

- Dados históricos: o recurso de dados históricos destacará ao usuário informações importantes relacionadas aos ativos digitais de interesse. Por exemplo, volume de negociações, mínimas e máximas para cada ativo, e indicadores gerais. Essas informações podem auxiliar o usuário a tomar suas decisões;
- Negociações: a plataforma permitirá a compra e venda manual de ativos digitais com suporte a diversas corretoras do mercado de ativos digitais;
- Testes de estratégias: será possível definir uma estratégia para automação de negociação e analisar seu resultado a partir de dados históricos;
- Automações: a partir do momento em que uma estratégia de automação se apresenta como lucrativa nos testes, será possível habilitá-la para sua execução automática na plataforma.

4.2 ARQUITETURA

A arquitetura da plataforma Crypsense será composta por oito microsserviços. Dois deles se comunicarão com serviços externos para envio e recebimento de informações de compra e venda do mercado de ativos digitais. Um outro irá expor um serviço *web* como ponto de acesso aos usuários. A Figura 11 a seguir ilustra essa arquitetura.

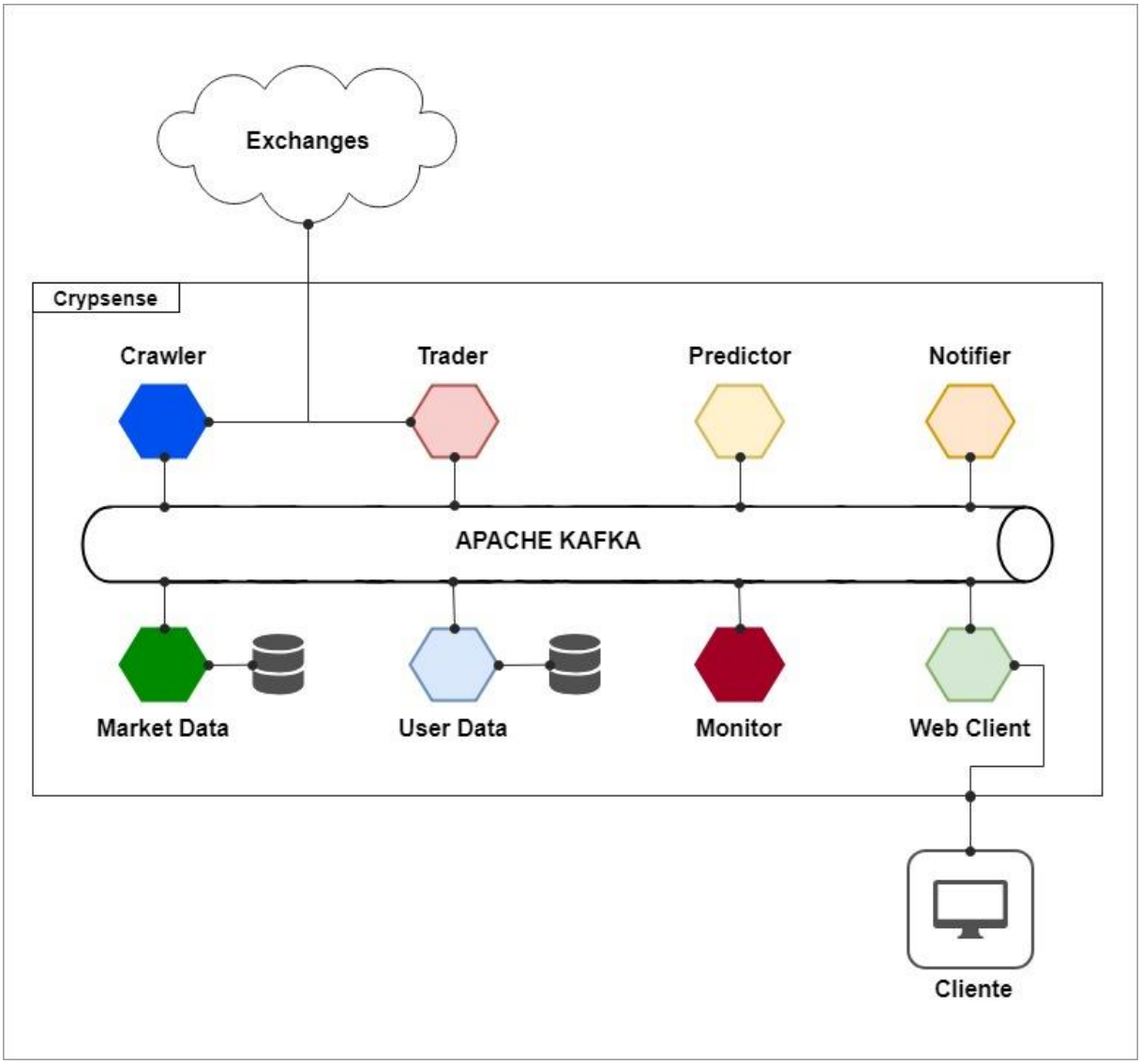

Figura 11 – Arquitetura da Plataforma Crypsense

Fonte: Autoria própria.

Conforme pode ser observado na Figura 11, a plataforma é formada por oito microsserviços que se comunicam através do Apache Kafka. Cada microsserviço possui uma responsabilidade bem definida, conforme descrito a seguir:

- Monitor: armazenar a inscrição e sinalização dos outros membros na plataforma e fornecer dados para monitoramento e controle;
- Crawler: se comunicar com corretoras do mercado de ativos digitais para obter as últimas negociações e publicá-las no Apache Kafka, tornando esses dados disponíveis para os outros microsserviços da plataforma;
- Market Data: consumir e armazenar as negociações capturadas pelo Crawler e publicar dados sumarizados;
- Predictor: consumir dados sumarizados produzidos pelo Market Data, através de métricas calcular possíveis valorizações e/ou desvalorizações e compartilhar suas decisões com os outros membros da plataforma;
- Trader: consome as previsões publicadas no Apache Kafka pelo Predictor e interage com o serviço externo das corretoras para comprar ou vender ativos;
- Webclient: fornecer, através de uma interface gráfica, todos os dados consumidos na plataforma para o usuário, por exemplo últimos valores, últimas decisões de compra ou venda do Predictor e relatório de negociações;
- User Data: armazenar informações e negociações do usuário inseridas através do microsserviço Webclient;
- Notifier: gerar notificações para o usuário.

Embora todos os microsserviços essenciais tenham sido representados na arquitetura, apenas três foram implementados: Market Data, Crawler e Monitor. Esses microsserviços estão destacados na Figura 11 com um preenchimento de cor mais forte e sem borda.

4.3 MODELAGEM DAS COREOGRAFIAS

Os três microsserviços implementados, Monitor, Crawler e Market Data, colaboram entre si para desempenhar alguns recursos oferecidos pela plataforma Crypsense. O microsserviço Monitor interage com todos os outros microsserviços a fim de possibilitar a análise do comportamento interno dos participantes do sistema. Já os microsserviços Crawler e Market Data colaboram entre si para a obtenção e armazenamento das negociações capturadas em tempo real.

A Figura 12 a seguir ilustra o papel do Monitor na plataforma. Essa coreografia é encenada pelo Monitor e todos os outros microsserviços membros da plataforma Crypsense. Por isso, a descrição do serviço que colabora com o Monitor foi modelada para representar um serviço genérico, nomeado Service.

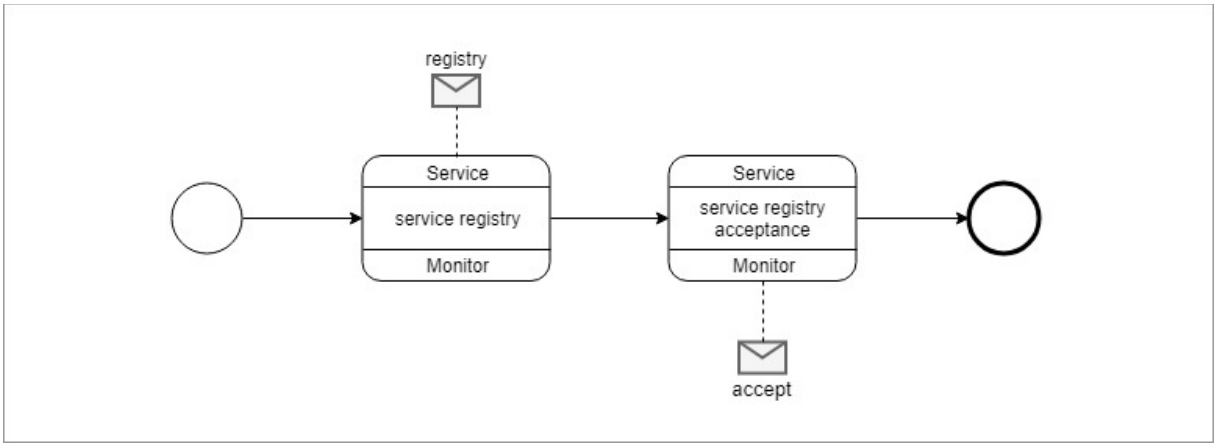

Figura 12 – Coreografia entre os microsserviços e o Monitor
Fonte: Autoria própria.

Na Figura 12, o círculo à esquerda representa o início da colaboração entre os dois microsserviços. O microsserviço participante, representado por Service, envia uma solicitação de registro ao Monitor. Este por sua vez, armazena informações do microsserviço participante e responde com um aceite. Como não há nenhuma outra comunicação entre os dois nessa composição, afirma-se que a coreografia foi encenada e o processo pode ser encerrado, com o círculo à direita.

Os microsserviços Crawler e Market Data, assim como qualquer outro microsserviço da plataforma Crypsense, participarão da coreografia de registro. Após o registro de ambos, eles irão colaborar para a coreografia descrita a seguir.

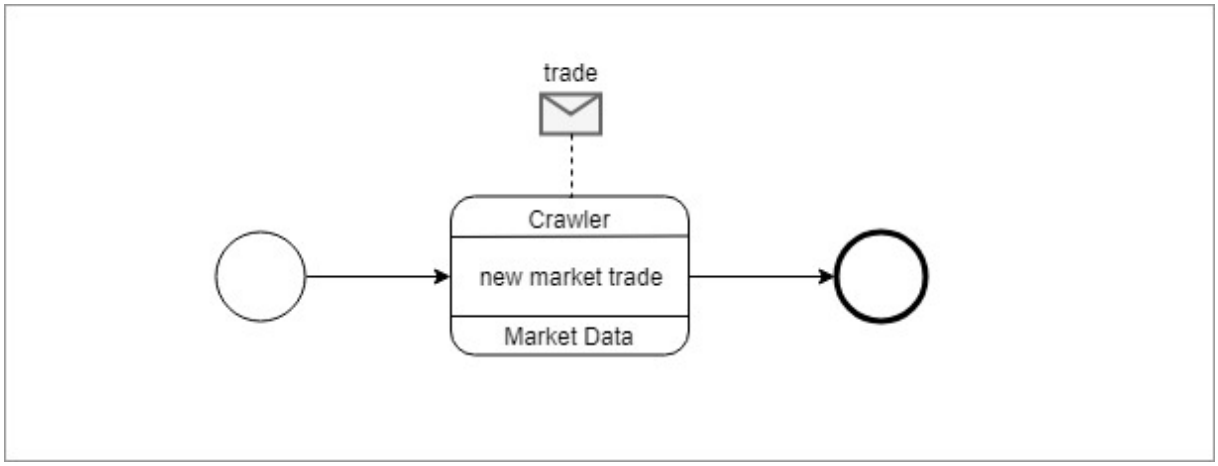

Figura 13 – Coreografia entre os microsserviços Crawler e Market Data
Fonte: Autoria própria.

A Figura 13 ilustra a colaboração entre os microsserviços Crawler e Market Data a fim de desempenhar a função distribuída responsável por obter negociações das corretoras e armazenar na plataforma Crypsense. Ao obter uma nova negociação, o microsserviço Crawler a envia para o Market Data.

Embora os modelos representem uma comunicação direta entre os participantes da coreografia, eles não se comunicam diretamente. Todas as mensagens são publicadas em um tópico no Apache Kafka e, posteriormente, consumidas pelos consumidores registrados em tais tópicos. Por exemplo, o microsserviço Crawler publica a negociação capturada através do serviço externo de uma corretora em um tópico no Apache Kafka. O Market Data, por sua vez, se registra nesse mesmo tópico e consome em tempo real todas as negociações. Além disso, é importante mencionar que embora os modelos representem somente a tarefa de comunicação entre os microsserviços, não significa que eles não possuam outras responsabilidades e não possam participar em outras coreografias. A linguagem BPMN é flexível e permite a modelagem apenas das comunicações entre microsserviços.

4.4 IMPLEMENTAÇÃO

Os microsserviços Crawler, Market Data e Monitor da Plataforma Crypsense foram implementados através das mesmas tecnologias apresentadas na Seção 3.3.1, ou seja, a linguagem de programação JavaScript, seu ambiente de execução Node.js e a implementação Kafkajs. Adicionalmente, também foi utilizado o banco de dados *open source* orientado a documentos MongoDB[12].

Para a comunicação entre os membros da coreografia de registro de serviço apresentada na Figura 12, dois tópicos são necessários. Um tópico é reservado à requisição de registro, nomeado *service-registry-request*. O outro tópico é reservado para o envio do aceite, nomeado *service-registry-acceptance*. Já para a comunicação entre os membros da coreografia da Figura 13, apenas um tópico é necessário, nomeado *new-market-trades.*

Conforme analisado na Seção 3.3.1, a redundância de código fonte é realmente uma desvantagem claramente perceptível nessa estratégia. Dado que praticamente todos papeis presentes na coreografia atuarão como consumidores e produtores, as implementações se repetem em todos eles. Para contornar essa característica, os códigos fonte dos Apêndice A e Apêndice B foram compartilhados com os serviços. No entanto, embora ocorra a redundância de código fonte, uma vez que a implementação da lógica de comunicação é disponibilizada em cada microsserviço basta definir seus comportamentos específicos.

[12]MongoDB. Disponível em <https://www.mongodb.com>. Acesso em nov. 2018.

5 CONCLUSÃO

A arquitetura de microsserviços é de fato uma alternativa à arquitetura monolítica e garante diversos benefícios. Por exemplo, a execução independente de serviços, interoperabilidade e baixo acoplamento. No entanto, esse modelo arquitetural requer que a composição dos serviços distribuídos seja gerenciada de forma adequada. A coreografia de microsserviços é um dos modelos de composição de serviços e se destaca por ter valores semelhantes aos da arquitetura de microsserviços.

A coreografia de microsserviços pode ser aplicada através de duas estratégias. De um lado a estratégia baseada em eventos, que já é adotada em larga escada atualmente e possui diversos casos de uso para comprovar sua utilidade. Além disso, através da plataforma Apache Kafka é possível coreografar serviços em larga escala e manter um altíssimo desempenho. Do outro lado, a estratégia baseada em programação coreográfica é um projeto pouco maduro, simples e sem muito uso. No entanto, possui valores intrinsecamente ligados à coreografia e desponta como uma alternativa para o futuro.

O principal objetivo da pesquisa, identificar e explorar as estratégias para coreografia de microsserviços, foi atingido. No entanto, algumas extensões deste trabalho, que por limitação de tempo e escopo não puderam ser investigadas, podem ocorrer. Por exemplo, a adoção do uso de *containers* como ambiente de execução dos microsserviços, a exploração da linguagem de microsserviços Jolie, e, também, a contribuição para a evolução do paradigma de programação coreográfica e, especificamente, da linguagem Chor.

REFERÊNCIAS

BRAVETTI, Mario; ZAVATTARO, Gianluigi. **Towards a Unifying Theory for Choreography Conformance and Contract Compliance**. Software Composition, p.34-50, 2007. Springer Berlin Heidelberg.

CIANCIA, Vincenzo et al. **Event Based Choreography**. Science of Computer Programming, v. 75, n. 10, p.848-878, out. 2010. Elsevier BV.

D'AMORE, Jean Robert. **Scaling Microservices with an Event Stream**. 2015. Disponível em <https://www.thoughtworks.com/insights/blog/scaling-microservices-event-stream>. Acesso em nov. 2018.

DECKER, Gero; KOPP, Oliver; BARROS, Alistair. **An Introduction to Service Choreographies**. IT - Information Technology, v. 50, n. 2, p.122-127, 2008. Walter de Gruyter GmbH.

DECKER, Gero; WESKE, Mathias. **Local Enforceability in Interaction Petri Nets**. Lecture Notes In Computer Science: 5th International Conference, Brisbane, Australia, p.305-319, 24-27 set. 2007. Springer Berlin Heidelberg.

DRAGONI, Nicola et al. **Microservices: Yesterday, Today, and Tomorrow**. Present And Ulterior Software Engineering, p.195-216, 2017. Springer International Publishing.

FOWLER, Martin; LEWIS, James. **Microservices**. 2014. Disponível em <http://martinfowler.com/articles/microservices.html>. Acesso em nov. 2017.

FUNDAÇÃO DE SOFTWARE APACHE. **Apache Kafka: Introduction**. 2017. Disponível em <https://kafka.apache.org/intro>. Acesso em nov. 2018.

GARG, Nishant. **Apache Kafka**. Packt Publishing, 2013.

GIARETTA, Alberto; DRAGONI, Nicola; MAZZARA, Manuel. **Joining Jolie to Docker**. Advances In Intelligent Systems And Computing, p.167-175, 2018. Springer International Publishing.

GOMES, Raphael de Aquino. **Implantação Eficiente de Múltiplas Coreografias de Serviços em Nuvens Híbridas**. 2017. 256p. Tese de Doutorado. Instituto de Informática, Universidade Federal de Goiás. Goiânia, 2017

GUIDI, Claudio et al. **Microservices: A Language-Based Approach**. Present And Ulterior Software Engineering, p.217-225, 2017. Springer International Publishing.

KOZHIRBAYEV, Zhanibek; SINNOTT, Richard O.. **A performance comparison of container-based technologies for the Cloud**. Future Generation Computer Systems, v. 68, p.175-182, mar. 2017. Elsevier BV.

KREPS, Jay; NARKHED, Neha; RAO, Jun. **Kafka: a Distributed Messaging System for Log Processing**. Networking Meets Databases, Athens, Greece, 12 jun. 2011.

MONTESI, Fabrizio. **Choreographic Programming**. 2013. 259p. Tese de Doutorado – IT University of Copenhagen, Copenhague, 2013.

MONTESI, Fabrizio. **Kickstarting Choreographic Programming**. Lecture Notes In Computer Science, p.3-10, abr. 2016. Springer International Publishing.

OLIVEIRA, Rodrigo Felipe Albuquerque Paiva de. **Uma Arquitetura de Microsserviços de Internet das Coisas para Casas Inteligentes**. Revista de Engenharia e Pesquisa Aplicada, v. 2, n. 2, 27 jul. 2017. Revista de Engenharia e Pesquisa Aplicada.

RAO, Jun; KREPS, Jay. **Apache Kafka: Powered By**. Disponível em <https://cwiki.apache.org/confluence/display/KAFKA/Powered+By>. Acesso em nov. 2018.

STUBBS, Joe; MOREIRA, Walter; DOOLEY, Rion. **Distributed Systems of Microservices Using Docker and Serfnode**. 2015 7th International Workshop On Science Gateways, jun. 2015. IEEE.

TOMMASO, Paolo di et al. **The impact of Docker containers on the performance of genomic pipelines**. Peerj, v. 3, 24 set. 2015. PeerJ.

TRIPOLI, Crislaine da Silva; CARVALHO, Rodrigo Pimenta. **Micros-serviços: características, benefícios e desvantagens em relação à arquitetura monolítica que impactam na decisão do uso desta arquitetura**. II Seminário de Desenvolvimento em SOA com Cloud Computing e Conectividade, ago. 2016.

WANG, Guozhang et al. **Building a replicated logging system with Apache Kafka**. Proceedings Of The Vldb Endowment, Kohala Coast, Hawaii, v. 8, n. 12, p.1654-1655, 1 ago. 2015. VLDB Endowment.

APÊNDICES

APÊNDICE A – CLASSE PRODUCER PARA APACHE KAFKA

```
1 import { Kafka } from 'kafkajs';
2
3 export default class Producer {
4   constructor() {
5     const kafka = new Kafka({
6       clientId: `${Date.now()}`,
7       brokers: ['localhost:9092'],
8     });
9
10    this.client = kafka.producer();
11  }
12
13  async publish({ topic, message }) {
14    await this.client.connect();
15    await this.client.send({ topic, messages: [message] });
16    await this.client.disconnect();
17  }
18 }
```

APÊNDICE B – CLASSE CONSUMER PARA APACHE KAFKA

```
1 import { Kafka } from 'kafkajs';
2
3 export default class Consumer {
4   // hidden constructor method
5
```

```
  async subscribe({ topic, filteredKey, onMessage }) {
    await this.client.connect();
    await this.client.subscribe({ topic, fromBeginning: true });
    await this.client.run({
      async eachMessage({ message }) {
        const key = message.key.toString();
        const value = message.value.toString();

        if (!filteredKey || filteredKey === key) {
          onMessage({ key, value });
        }
      },
    });
  }
}
```