Estratégias para Implementação de Coreografia de Microsserviços

João Pedro Monteiro Fernandes, Edson Tavares de Camargo

[1] Universidade Tecnológica Federal do Paraná - Campus Toledo (UTFPR)
CEP: 85902-490 – Toledo – PR – Brasil

ojoaofernandes@gmail.com, edson@utfpr.edu.br

Abstract. *The microservice architecture encourages the composition of services through choreography. Choreography favors low coupling and software decentralization. The big challenge is to find out suitable approaches to carry it out according to the architectural style proposed by the microservices. This paper describes and compares two strategies for choreographing microservices. The first is called event-based and makes use of a mediator to route the messages. The second is called choreographic programming, and its major advantage is to provide a global overview of choreography. A case study with four microservices was implemented using each strategy. Results include a comparative table and the number of messages per minute supported by each implementation.*

Resumo. *A arquitetura de microsserviços incentiva a composição de serviços através da coreografia. A coreografia favorece o baixo acoplamento e a descentralização. Surge então o desafio de encontrar abordagens adequadas para realizá-la de acordo com estilo arquitetural proposto pelos microsserviços. Este trabalho descreve e compara duas estratégias para coreografar microsserviços. A primeira estratégia é baseada em eventos e faz uso de um mediador para transmitir as mensagens. A segunda é chamada de programação coreográfica e sua grande vantagem é fornecer uma visão global da coreografia. Um estudo de caso com quatro microsserviços foi implementado em cada estratégia. Resultados incluem uma tabela comparativa e o número de mensagens por minuto suportado pelas implementações.*

1. Introdução

A arquitetura de microsserviços é um estilo arquitetural, inspirada na Arquitetura Orientada à Serviços (SOA), em que a aplicação é composta por uma suíte de serviços independentes e com responsabilidades bem definidas [Dragoni et al. 2017]. Comparados ao monólito, termo usado para definir uma aplicação que é executa em um único processo computacional, os microsserviços apresentam baixo acoplamento, garantem a interoperabilidade da aplicação e possibilitam a escalabilidade das funcionalidades que de fato requerem maior desempenho [Fowler and Lewis 2014]. A arquitetura vem chamando a atenção da comunidade de desenvolvedores e muitos são os trabalhos encontrados na literatura que contrapõem os microsserviços à arquitetura monolítica e mesmo à SOA [Cerny et al. 2018, Stubbs et al. 2015, Indrasiri and Siriwardena 2018].

Comparados à SOA, os microsserviços levam a uma maior autonomia dos serviços e promovem o desacoplamento uma vez que os serviços não precisam concordar em contratos em nível global [Cerny et al. 2018, Stubbs et al. 2015]. Dessa forma,

é possível implantar, corrigir, escalar individualmente cada serviço e distribuí-los em diferentes servidores localizados tanto em uma rede local quanto na Internet. 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. Ao contrário do que tradicionalmente é feito em Serviços Web inspirados em SOA, microsserviços não contam com um componente responsável pela orquestração e por isso preferem a coreografia [Cerny et al. 2018].

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 and Zavattaro 2007]. 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 leves para troca de mensagens, ferramentas para localização e a composição da regra de negócio usando unidades distribuídas.

O objetivo deste trabalho é apresentar e comparar duas estratégias para coreografia de microsserviços. A primeira estratégia é baseada em eventos e a segunda estratégia é chamada de programação coreográfica. Na estratégia de notificação de eventos, membros de uma coreografia atuam como produtores e/ou consumidores e a comunicação é mediada por sistemas de mensagens, como o Apache Kafka, ActiveMQ e RabbitMQ. Por outro lado, a programação coreográfica possui a vantagem de obter um sistema distribuído livre de *deadlocks*, comunicação ponto-a-ponto e a especificação de coreografia é representada pela formalização das interações esperadas entre os membros a partir da perspectiva global. Um sistema com quatro microsserviços foi proposto e implementado em cada estratégia. Resultados apresentados incluem uma tabela comparativa entre as estratégias a partir de nove característica elencadas. Também são apresentados o número de mensagens por minuto obtidos após executar cada implementação do estudo de caso proposto.

Este artigo segue organizado da seguinte forma. A Seção 2 apresenta a arquitetura de microsserviços. A Seção 3 explora a coreografia e apresenta duas estratégias para realizá-la. Um estudo de caso onde uma coreografia é implementada aplicando as estratégias descritas é apresentado na Seção 4. A Seção 5 apresenta os resultados obtidos. Por fim, a Seção 6 descreve a conclusão.

2. Arquitetura de Microsserviços

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 et al. 2015]. Por exemplo, considere 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.

Uma aplicação de software cuja arquitetura seja monolítica possui benefícios durante os estágios iniciais de desenvolvimento. 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 et al. 2015]. Em contrapartida, há cenários em que esse estilo arquitetural dificulta o desenvolvimento. Dado que as regras de negócio são encapsuladas em um único artefato executável, possivelmente haverá um forte acoplamento o que acarreta negativamente a produtividade do programador, dificulta correções na aplicação e no gerenciamento de bibliotecas.

Uma possível solução é desacoplar as pequenas abstrações e torná-las independentes, como ocorre na arquitetura de microsserviços [Kozhirbayev and 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 and 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 mais fácil entendimento, desenvolvimento e manutenção, tanto para desenvolvedores experientes quanto para novos membros da equipe [Indrasiri and Siriwardena 2018];
- 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 and 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 and 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 HTTP (*HyperText Transfer Protocol*) é comumente empregado na comunicação síncrona entre os microsserviços, enquanto os Protocolos AMQP (*Advanced Message Queuing Protocol*) e MQTT (*Message Queuing Telemetry Transport*) são empregados na comunicação assíncrona [Indrasiri and Siriwardena 2018]. Por fim, vale lembrar que microsserviços podem ser considerados como uma evolução da arquitetura orientada a serviços representada pelos serviços web [Fowler and Lewis 2014]. Dessa forma, boas práticas e estratégias empregadas na arquitetura orientada a serviços podem ser reaproveitadas no cenário dos 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 and Zavattaro 2007]. 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 ao invés da 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.

Figura 1. Diferença entre composição por orquestração e por coreografia [Gomes 2017].

A Figura 1 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 1, 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. A seguir, descreve-se a coreografia de microsserviços.

3. Coreografia de Microsserviços

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 ocorre através da troca de mensagens entre os próprios serviços. Suprime-se a necessidade de um orquestrador para coordenar o fluxo. Portanto, a coreografia é a descrição global das interações entre os membros de um sistema distribuído [Barker et al. 2009].

Durante o projeto de uma coreografia de serviços, 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.

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. Coreografias de microsserviços podem ser modeladas através de linguagens específicas. Essas linguagens são divididas em duas categorias independentes de implementação e específicas da implementação. A primeira modela coreografias na perspectiva de negócios e processos, sem necessariamente especificar detalhes de implementação. Nesse contexto é possível citar linguagens como Let's Dance e BPMN 2.0 [Gomes 2017]. A segunda linguagem modela coreografias com detalhes técnicos, como definições de protocolos e estrutura das mensagens. Estão nessas categorias linguagens como WS-CDL, WSFL e BPEL4Chor [Gomes 2017].

A linguagem gráfica Modelo e Notação de Processos de Negócios (Business Process Modeling Model, BPMN) oferece notação em alto nível para modelar a composição de papeis na coreografia. A linguagem possui 2 versões e a abordagem dos modelos produzidos pode ser diferente entre elas. A versão 1 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, mais recente, possui novas abstrações para permitir a especificação de coreografias através das próprias interações entre os papeis, o que garante a visão global da colaboração.

3.1. Coreografia Baseada em Eventos

Na estratégia de notificação de eventos, membros de uma coreografia atuam como produtores e/ou consumidores. Um consumidor se inscreve para receber determinados eventos. Sempre que um membro precisa se comunicar com outro, ele simplesmente publica ou produz um evento. Não é necessário conhecer a identidade dos possíveis consumidores [Zhang and Chen 2015]. 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 diminui as interdependências entre os serviços, garantindo um baixo acoplamento. Do mesmo modo, os serviços podem ser implantados em diferentes servidores e contextos e se comunicar somente através da publicação e consumo de eventos. 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 [Ciancia et al. 2010]. 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. 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 2 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.

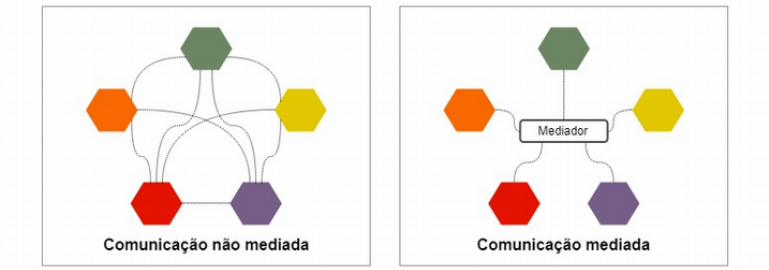

Figura 2. Diferença entre comunicação mediada e não mediada [D'Amore 2015].

Conforme apresenta a Figura 2, 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 há a comunicação entre eles. Por outro lado, 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 and 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 [Garg 2013, Foundation 2018]: Filas e Publicação-Assinatura (*publish-subscribe*). No primeiro os eventos são recebidos pelo sistema e salvos em uma fila, para que então um conjunto de consumidores os consumam alternadamente. Essa característica permite a divisão do processamento de dados entre diversos consumidores sem duplicidade de leitura. No segundo, 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 todas os eventos vão para todos os assinantes. A plataforma Apache Kafka [Kreps et al. 2011] é uma solução para sistemas de mensagens que se enquadram no modelo publicação-assinatura e é empregada neste trabalho para realizar a coreografia baseada em eventos.

3.2. Programação Coreográfica

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 recente 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 2015]. 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: $\text{Alice} \rightarrow \text{Bob} : \text{livro}; \text{Bob} \rightarrow \text{Alice} : \text{dinheiro}$.

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. As implementações de microsserviços (*endpoints*) sempre corretas por construção. 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.

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 que permite que, ao finalizar a especificação da coreografia, *endpoints* sejam projetados para a sequência das atividades de implementação.

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. 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 as interações.

Apesar de ser uma linguagem, o Chor é disponibilizado atualmente apenas como um complemento para o ambiente de desenvolvimento integrado Eclipse. 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 [Giallorenzo et al. 2018]. 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 [Lanese et al. 2015].

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 para facilitar a contribuição de terceiros.

4. Estudo de Caso

O estudo de caso proposto consiste de quatro microsserviços. A coreografia foi definida e modelada utilizando a linguagem de modelagem BPMN 2.0. Então, a partir do modelo gerado, a coreografia foi implementada em cada estratégia. A estratégia de notificação de eventos foi implementada através da Plataforma Apache Kafka. A programação no paradigma de Programação Coreográfica faz uso da linguagem Chor.

A Figura 3 apresenta quatro papeis que simulam a comunicação de acordo com a topologia de um anel. O microsserviço `Initiator` inicia a coreografia ao enviar uma mensagem `initiate` para o `Stream Processor A`, que ao recebê-la a envia para

o microsserviço `Stream Processor B`. Este, por sua vez, envia a mensagem para o `Returner`. O `Returner` então encerra a coreografia ao devolver a mensagem para o microsserviço `Initiator`. As subseções seguintes apresentam a implementação do modelo nas estratégias descritas.

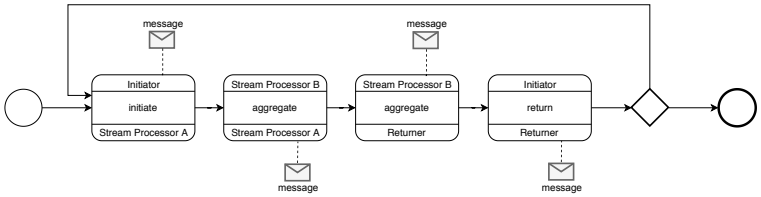

Figura 3. Coreografia para estudo de caso usando quatro microsserviços.

4.1. Implementação na Plataforma Kafka

Para a implementação dos membros da coreografia proposta foi utilizada a linguagem de programação JavaScript, seu ambiente de execução Node.js v10.15.1, o gerenciador de pacotes JavaScript NPM v6.8.0, a implementação kafka-node (pacote NPM) v4.0.2, uuid (NPM package) v3.3.2. Também, o Apache Kafka v2.1.0 gerenciado pelo Apache Zookeeper v3.4.13

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 permite que os microsserviços sejam desenvolvidos em diversas linguagens de programação. Além disso, oferece recursos para a construção de sistemas distribuídos e em tempo real de forma confiável, tolerante a falhas e em alta performance. É possível também gerenciar serviços através de arquivos executáveis que são oferecidos junto com a distribuição padrão [Foundation 2018].

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.

Uma das desvantagens da arquitetura de microsserviços é a redundância de código fonte e essa característica pode ser observada no estudo de caso proposto. Dado que os papeis presentes na coreografia atuarão como consumidores e produtores, as implementações serão semelhantes para cada um. Para evitar esse problema uma única implementação para consumir e produzir mensagens na plataforma Apache Kafka foi especificada e compartilhada com cada serviço. Isso foi realizado através da implementação do `ServiceRunner` e do `BrokerCommons`. O primeiro implementa as funcionalidades para executar os serviços. E o segundo é uma coleção das funções utilizadas por todos os serviços.

Apesar do Algoritmo 1 apresentar apenas o código para o `Initiator`, código semelhante é empregado nos demais serviços apenas modificando as linhas 10, 15 e 16. O código inicia importando as bibliotecas descritas nas linhas 1 a 4. A linha 5 importa a implementação do comportamento do serviço. As linhas 7 e 8 instanciam o serviço

`Initiator` e o `ServiceRunner`. As linhas 9 a 17 contêm as configurações do `Initiator`. Por exemplo, as linhas 15 e 16 especificam o tópico que será publicado e o tópico que será consumido. As linhas 19 a 25 especificam o processo de execução do serviço pelo `ServiceRunner`. A linha 20 define as configurações escritas acima no escopo global do `ServiceRunner`. As linhas 21 a 24 adicionam os passos necessários para que o serviço se conecte ao Apache Kafka, verifique se os tópicos existem e instancia um consumidor e produtor. Por fim, a linha 25 executa os passos na ordem definida e executa o serviço.

```
1  import ServiceRunner from 'packages/service-runner';
2  import {
3   brokerConnection, consumerConnection, producerConnection, topicsSetup,
4  } from 'packages/broker-commons';
5  import Service from './service';
6
7  const service = new Service();
8  const runner = new ServiceRunner();
9  const config = {
10   group: 'initiators',
11   broker: {
12     host: '127.0.0.1',
13     port: '9092',
14   },
15   publishToTopics: ['queue.event-ia'],
16   consumeFromTopics: ['queue.event-ri'],
17 };
18
19 runner
20   .prepare(config)
21   .step(brokerConnection)
22   .step(topicsSetup)
23   .step(consumerConnection)
24   .step(producerConnection)
25   .run(service);
```

Listing 1. Código JavaScript para executar o microsserviço `Initiator`

4.2. Implementação na Linguagem Chor

A Chor versão Alpha e a linguagem Jolie v1.7.1 foram empregadas na coreografia do sistema distribuído descrito no estudo de caso. Importante observar que a linguagem Chor especifica a coreografia em um único arquivo, conforme apresenta o código do Algoritmo 2. Essa característica garante a perspectiva global das interações durante a implementação do sistema.

A definição do protocolo, que ocorre entre as linhas 3 e 8, formaliza quantos e quais serão os papeis na coreografia e como eles interagem entre si. Por exemplo, o `Initiator` se comunica com o `StreamProcessor1` através da função `initiate` ao enviar uma `string`. A construção da coreografia no corpo no algoritmo, entre as linhas 12 e 23, implementa de forma fixa o envio da mensagem a cada microsserviço e o seu retorno, também fixo. Essa construção deve, obrigatoriamente, implementar o protocolo definido no início. Esse código é então utilizado como parâmetro de entrada para o processo de projeção de *endpoint* e os respectivos microsserviços são gerados em Jolie.

```
1  program Serial;
2
3  protocol SerialProtocol {
4      Initiator        -> StreamProcessor1 : initiate  (string) ;
5      StreamProcessor1 -> StreamProcessor2 : aggregate (string) ;
6      StreamProcessor2 -> Returner         : aggregate (string) ;
7      Returner         -> Initiator        : return    (string)
8  }
9
10 public SerialChannel : SerialProtocol
11
12 main {
13     i[Initiator] start
14         sp1[StreamProcessor1] ,
15         sp2[StreamProcessor2] ,
16         r[Returner]
17         : SerialChannel (channel);
18
19     i    . "message" -> sp1 . payload : initiate  (channel) ;
20     sp1  . (payload) -> sp2 . payload : aggregate (channel) ;
21     sp2  . (payload) -> r   . payload : aggregate (channel) ;
22     r    . (payload) -> i   . payload : return    (channel)
23 }
```

Listing 2. Código Chor para especificar a coreografia

Ao gerar os *endpoints* as regras de negócio ou especificidades de domínio podem ser implementadas diretamente no código fonte do microsserviço. Por exemplo, a mensagem enviada pode ser lida em um arquivo presente no sistema de arquivos do sistema operacional ou consultada em um sistema gerenciador de banco de dados.

5. Resultados

Tendo como base o modelo gerado, a coreografia foi implementada por cada alternativa técnica de cada estratégia. A partir do estudo de caso, conclui-se 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. A Tabela 1 apresenta a comparação através de nove características observadas nas duas estratégias a partir das descrições apresentadas nas Seções 3.1, 3.2 e o estudo de caso implementado nas Seções 4.1 e 4.2. Além disso, também são apresentados o número de mensagens por minuto (vazão) obtidos a partir das implementações do estudo de caso. As implementações foram executadas em um máquina com sistema Linux Kernel 4.15.0-46-generic, distribuição Ubuntu 16.04.1 com 64 bits, processador Intel Core i5-3230M CPU @ 2.60GHz de quadro núcleos, memória DRAM Controller 7859MiB.

Analisando a comparação exposta na Tabela 1, considera-se que a estratégia baseada em eventos possui características importantes, como documentação completa, facilidade de manutenção, suporte a várias linguagens de programação e possui diversos estudos de casos. Essas características não se limitam a coreografia usando a ferramenta Apache Kafka. Também são encontradas essas características em outros sistemas de mensagens. As vantagens da programação coreográfica estão na perspectiva global e na construção por *design* que garante um sistema livre de *deadlocks*.

Característica	Baseada em eventos	Programação coreográfica
Mecanismo	Comunicação	Programação
Perspectiva	Individual	Global
Comunicação	Mediada	Ponto a ponto
Maturidade	Alta	Baixa
Facilidade de manutenção	Alta	Baixa
Documentação	Completa	Apenas a sua tese
Linguagens de programação	Mais de 15 implementações	Apenas Jolie
Estudo de casos	Diversos casos triviais e complexos	Quantidade limitada e trivial
Verificação por *design*	Não	Sim

Tabela 1. Comparação entre as estratégias de coreografia.

A principal vantagem na implementação de uma coreografia baseada em eventos é que uma interação não precisa ocorrer diretamente entre dois papéis. Basta que os microsserviços estejam conectados através do mesmo tópico, um produz e o outro consome. Essa característica permite escalar horizontalmente a aplicação apenas com a execução de um novo microsserviço interpretando o mesmo papel. Também, todas as tentativas de interação, eventos publicados, são registrados no mediador (Apache Kafka), assim não há perda de mensagem e é possível que o sistema continue uma ação distribuída mesmo após falha parcial ou integral dos microsserviços.

Em contrapartida, os microsserviços são implementados individualmente na sua própria perspectiva do todo. Logo, por não permitir visualizar globalmente os relacionamentos entre os membros da coreografia e dependendo da complexidade das interações, empregar modelagem, por exemplo através da BPMN 2.0, será essencial. Inevitavelmente os microsserviços serão iguais em diversos aspectos (por exemplo, a conexão com o *broker*, instanciação do produtor, consumidor e interação com os tópicos) e não há uma forma inteligente para projetá-los. Assim, será necessário implementar, gerenciar e distribuir bibliotecas para garantir a reusabilidade de código entre os microsserviços.

A implementação baseada na programação coreográfica destaca-se pela especificação em uma perspectiva global, ou seja, as comunicações são descritas sem quaisquer detalhes específicos de implementação de quem interpreta o papel. A implementação se concentra nas interações entre os membros. Em contrapartida, a especificação é utilizada como entrada para o processo de projeção de *endpoints* apenas uma vez. Ainda não é possível adotar o paradigma e sua linguagem, Chor, como ferramenta de trabalho numa metodologia de desenvolvimento de software incremental, por exemplo. Obrigatoriamente, cada alteração na especificação exige uma nova projeção de *endpoints*. A seguir são apresentados os resultados referentes à vazão para as implementações realizadas.

A Figura 4 exibe a vazão (eixo y) obtida após diversas execuções das implementações do estudo de caso. As execuções consideram diferentes tamanhos de mensagens (eixo x): 2^5 até 2^{12} bytes. A linha roxa exibe a vazão obtida na execução via Chor/Jolie e a linha azul a vazão via Kafka/Node.js. Mesmo ao aumentar o tamanho da mensagem, o estudo de caso desenvolvido via Chor apresenta uma taxa de transferência de 2,5x acima da vazão presente via Kafka/Node.js. Esse é um ponto altamente favorável à abordagem de programação coreográfica. Importante destacar que as duas abordagens utilizam protocolo binário próprio para comunicação entre os serviços.

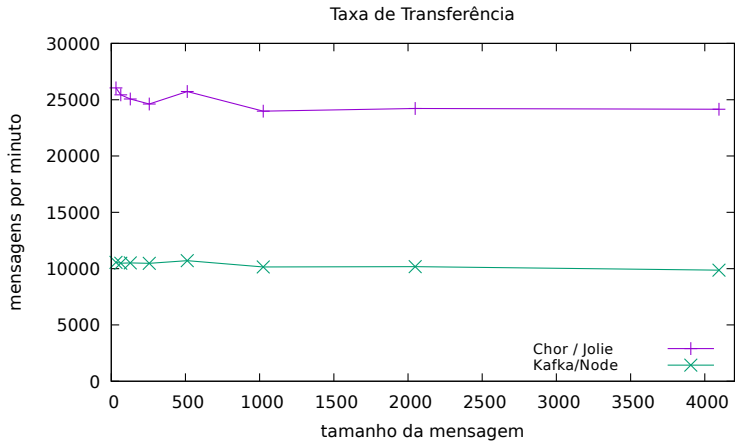

Figura 4. Taxa de transferência.

6. 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 bom desempenho. Do outro lado, a estratégia baseada em programação coreográfica é um projeto pouco maduro. Ainda assim, possui valores intrinsecamente ligados à coreografia e desponta como uma alternativa promissora. Suas características garantem maior vazão quando comparada a estratégia baseada em eventos usando Apache Kafka.

Trabalhos futuros incluem a adoção do uso de *containers* como ambiente de execução dos microsserviços, distribuição em diferentes domínios, escalabilidade horizontal e tolerância a falhas. Outra direção para trabalhos futuros incluem a exploração da linguagem de microsserviços Jolie e, também, a contribuição para a evolução do paradigma de programação coreográfica.

Referências

Barker, A., Walton, C. D., and Robertson, D. (2009). Choreographing web services. *IEEE Transactions on Services Computing*, 2(2):152–166.

Bravetti, M. and Zavattaro, G. (2007). Towards a unifying theory for choreography conformance and contract compliance. In *Proceedings of the 6th International Conference on Software Composition*, SC'07, pages 34–50, Berlin, Heidelberg. Springer-Verlag.

Cerny, T., Donahoo, M. J., and Trnka, M. (2018). Contextual understanding of microservice architecture: Current and future directions. *SIGAPP Appl. Comput. Rev.*, 17(4):29–45.

Ciancia, V., Ferrari, G., Guanciale, R., and Strollo, D. (2010). Event based choreography. *Science of Computer Programming*, 75(10):848–878.

D'Amore, J. R. (2015). Scaling microservices with an event stream. Em 2019-03-20.

Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., and Safina, L. (2017). *Microservices: Yesterday, Today, and Tomorrow*, pages 195–216. Springer International Publishing, Cham.

Foundation, A. (2018). Apache kafka - a distributed streamin platform. Em 2019-03-20.

Fowler, M. and Lewis, J. (2014). Microservices. Em 2019-03-20.

Garg, N. (2013). *Apache Kafka*. Packt Publishing.

Giallorenzo, S., Montesi, F., and Gabbrielli, M. (2018). Applied choreographies. In *FORTE 2018*, pages 21–40. Springer.

Gomes, R. d. A. (2017). *Implantação eficiente de múltiplas coreografias de serviços em nuvens híbridas*. PhD thesis, Universidade Federal de Goiás, http://repositorio.bc.ufg.br/tede/handle/tede/7351.

Guidi, C., Lanese, I., Mazzara, M., and Montesi, F. (2017). Microservices: a language-based approach. *CoRR*, abs/1704.08073.

Indrasiri, K. and Siriwardena, P. (2018). *Microservices for the Enterprise: Designing, Developing, and Deploying*. Apress.

Kozhirbayev, Z. and Sinnott, R. O. (2017). A performance comparison of container-based technologies for the cloud. *Future Generation Computer Systems*, 68:175 – 182.

Kreps, J., Narkhede, N., and Rao, J. (2011). Kafka: A distributed messaging system for log processing. In *Proceedings of 6th International Workshop on Networking Meets Databases (NetDB), Athens, Greece*.

Lanese, I., Montesi, F., and Zavattaro, G. (2015). The evolution of jolie - from orchestrations to adaptable choreographies. In Nicola, R. D. and Hennicker, R., editors, *Software, Services, and Systems*, volume 8950 of *Lecture Notes in Computer Science*, pages 506–521. Springer.

Montesi, F. (2013). *Choreographic Programming*. Ph.D. thesis, IT University of Copenhagen. urlhttp://www.fabriziomontesi.com/files/choreographic_programming.pdf.

Montesi, F. (2015). Kickstarting choreographic programming. In Hildebrandt, T. T., Ravara, A., van der Werf, J. M., and Weidlich, M., editors, *Web Services, Formal Methods, and Behavioral Types - 11th International Workshop, WS-FM 2014, Eindhoven, The Netherlands, September 11-12, 2014, and 12th International Workshop, WS-FM/BEAT 2015, Madrid, Spain, September 4-5, 2015, Revised Selected Papers*, volume 9421 of *Lecture Notes in Computer Science*, pages 3–10. Springer.

Stubbs, J., Moreira, W., and Dooley, R. (2015). Distributed systems of microservices using docker and serfnode. In *2015 7th International Workshop on Science Gateways*, pages 34–39.

Zhang, Y. and Chen, J.-l. (2015). Constructing scalable internet of things services based on their event-driven models. *Concurrency and Computation: Practice and Experience*, 27(17):4819–4851.