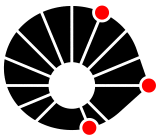

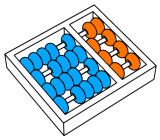

Análise de dependências via comunicação assíncrona em arquiteturas baseadas em microsserviços

J. P. Amorim *B. B. N. França*

Relatório Técnico - IC-PFG-20-41
Projeto Final de Graduação
2020 - Dezembro

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

Análise de dependências assíncronas em arquiteturas baseadas em microsserviços

João Pedro de Amorim* Breno Bernard Nicolau de França*

Resumo

Em diferentes cenários, o estilo arquitetural em microsserviços vem ganhando notoriedade como referência de arquitetura a ser seguida em aplicações modernas, baseadas em núvem computacional, que buscam escalabilidade como requisito fundamental. Embora essa arquitetura promova uma série de benefícios, ela também traz desafios como o gerenciamento de sua evolução e a complexidade operacional. Nesse contexto, o uso de *frameworks* baseados em sistemas de mensagens (como o *Kafka*, por exemplo) tornou-se um padrão na indústria, sendo um método confiável e eficiente para estabelecer dependências assíncronas entre serviços. Assim, em uma arquitetura distribuída como a de microsserviços, é esperado um aumento de complexidade e consequentemente, perder a referência de como os componentes do sistema se comunicam entre si, ocasionando serviços com muitas dependências. Neste trabalho, visamos explorar a questão do gerenciamento de dependências, desenvolvendo uma abordagem conceitual e uma implementação prática dessa para mapear dependências assíncronas via *Kafka* no caso do projeto *SiteWhere*.

Após analisar o repositorio (código-fonte) do projeto, fomos capazes de desenvolver um algoritmo, além de realizar sua implementação e aplicação na arquitetura projeto e, por fim, comparamos os resultados obtidos com as relações obtidas via inspeção manual do projeto e a partir da análise de sua documentação. O algoritmo foi capaz de mapear corretamente 12 das 16 relações previstas pela inspeção manual e pela documentação. São apresentados também limitações e oportunidades de melhoria do algoritmo, além de uma análise de acoplamento do próprio *SiteWhere* a partir dos resultados obtidos.

*Instituto de Computação, Universidade Estadual de Campinas, 13081-970 Campinas, SP.

1 Introdução

No contexto atual de desenvolvimento de software, o estilo arquitetural baseado em microsserviços vem se tornando cada vez mais utilizado como uma decisão de projeto de sistemas no intuito de prover manutenibilidade, rápida entrega e alta escalabilidade [1]. Não é por acaso que gigantes da tecnologia adotarão esse padrão para seus sistemas e um número alto de sistemas considerados "monolíticos" (isto é, que funcionam apenas como uma grande unidade lógica em um único processo) vêm sendo reestruturados em múltiplos serviços, executados em diferentes processos, a fim de adotar esse padrão [2].

Concomitante com a alta dos microsserviços na indústria, uma solução muito adotada no contexto da comunicação assíncrona entre serviços é a utilização de sistemas de mensageria [3], como *Apache Kafka*[1] e RabbitMQ[2], implementando o estilo publisher-subscriber.

Devemos destacar que, ao se adotar uma estrutura inerentemente distribuída, como é o caso do padrão de microsserviços, há um risco de um aumento significativo da complexidade do sistema. Isso permite o surgimento de características detrimentais à manutenibilidade do sistema - e uma dessas características é o forte acoplamento entre serviços, que possui um impacto negativo direto na manutenção do sistema [4].

Assim, em uma arquitetura baseada em microsserviços, existe a necessidade de mapear as interdependências entre serviços de uma forma eficaz, a fim de se ter um uma visão geral de como se estrutura o sistema quanto à comunicação (e consequentemente, o acoplamento) entre serviços.

Recentemente, existem soluções para identificação de dependências entre microsserviços, como o GMAT [5] e SYMBIOTE [6], sendo este último um método capaz de analisar a evolução do acoplamento ao longo de inúmeras liberações. Todavia, não identificamos, até a conclusão deste trabalho, uma abordagem e/ou ferramenta de análise automatizada para identificação de dependências assíncronas, uma vez que os trabalhos anteriormente citados foram fundamentalmente na comunicação síncrona.

Na ausência de tal solução, torna-se complexo ter uma visão geral de como os componentes de sistema baseado em microsserviços concretamente se comunicam. Com pouca visibilidade do sistema, é possível que problemas em nível de projeto arquitetural (como o forte acoplamento entre serviços) passem desapercebidas e tomem proporções descontroladas na sua evolução. Ainda, não ter uma visão clara de como os componentes de um sistema se relacionam facilita a tomada de decisões equivocadas e/ou redundantes de desenvolvimento, o que impacta de forma negativa o projeto.

Assim, neste trabalho, propõe-se o desenvolvimento de uma abordagem conceitual tal como uma implementação concreta de uma ferramenta de análise estática de código capaz de mapear dependências assíncronas entre serviços e denotar tais resultados visualmente, por meio de um grafo de dependências.

Considerando o amplo escopo do problema, delimitamos para este trabalho o objetivo factível de desenvolver a tal solução considerando uma ferramenta específica de mensageria, sendo o *Apache Kafka*, dada sua ampla adoção no mercado e em sistemas de software livre. Para isso, adotamos uma abordagem incremental, onde uma solução específica foi proposta

[1] https://kafka.apache.org
[2] https://www.rabbitmq.com

visando o projeto *SiteWhere*, uma plataforma *IoT* de código aberto que adota o padrão arquitetural baseado em microsserviços e possui comunicação via *Apache Kafka*.

Ainda que tenhamos desenvolvido uma solução para um caso particular, o processo de entendimento do problema em um caso específico e identificação das limitações fornecem uma contribuiçao inicial para soluções mais genéricas de mapeamento de comunicações entre microsserviços, uma vez que será possível utilizá-la como base para indicar o que é específico da implementação do *SiteWhere* e o que pode ser considerado genérico à uma arquitetura qualquer baseada em microsserviços.

O restante deste trabalho está organizado da seguinte forma. Na Seção 2, apresentamos a fundamentação teórica com os principais conceitos envolvidos nesse trabalho. Em seguida, na Seção 3, apresentamos o método utilizados para explorar o problema e encaminhar a solução. Na Seção 4, é descrita a solução pelo viés conceitual e de implementação prática. Na Seção 5, trazemos os resultados do nossa solução e por fim, na Seção 6, avaliamos o desempenho obtido e denotamos pontos de gargalo e melhoras para próximas iterações.

2 Fundamentação Teórica

Nesta seção, apresentamos os principais conceitos no trabalho, de acordo com as subseções a seguir.

2.1 Microsserviços

James Lewis [7] define o estilo arquitetural baseado em microsserviços como "*uma abordagem para se desenvolver uma única aplicação como um conjunto de pequenos serviços, cada um rodando em seu próprio processo e se comunicando por mecanismos leves, comumente a partir de uma API por meio do protocolo HTTP.*"

Esse estilo arquitetural, pelo menos em relação à sua definição e categorização, é bem recente e cada vez mais vem sendo adotado frente à abordagem monolítica, uma vez que provê uma série de vantagens sobre a mesma. A abordagem monolítica, como o próprio nome indica, não oferece uma modularização apropriada em serviços: nela, uma única unidade lógica é responsável por todo o processamento pelo lado do servidor - ou seja, ela receberá requisições de clientes, executará a lógica pertinente ao domínio da requisição, se comunicará com um banco de dados e devolverá as informações pertinentes à cada requisição.

Um exemplo prático é de bom grado para entendermos bem a distinção entre as arquiteturas: considere o caso de um cliente que realiza seu cadastro e em seguida, realiza um clique no ícone que representa seu *feed* de notícias em um aplicativo de uma rede social.

Em uma arquitetura monolítica, um único servidor processará tanto a requisição referente ao cadastro quanto a referente ao *feed*, executará a lógica pertinente ao domínio do cadastro (por exemplo, a autenticação do usuário, verificando se suas credenciais estão de acordo e/ou analisando cabeçalhos da requisição, como *tokens*) e a lógica pertinente ao domínio do *feed* (por exemplo, requisitar ao banco de dados as notícias que são recomendadas àquele usuário em específico e para as notícias que possuem algum conteúdo de mídia, requisitar os *assets* pertinentes à *Content Delivery Network* [8] do aplicativo), e por fim, responder à cada requisição com as informações obtidas.

Já em uma arquitetura baseada em microsserviços, é esperado que a requisição seja primeiramente direcionada à um *API Gateway* [9], que por sua vez, direciona a requisição de cadastro para um serviço (ou conjunto de serviços) feito especificamente para o cadastro de usuários (rodando em seu próprio servidor) e analogamente, a requisição do *feed* de notícias será direcionado para um serviço feito para atender esse tipo específico de requisição.

Note que a arquitetura monolítica não possui uma segmentação de domínio (o que vai contra o proposto pela abordagem *Domain Driven Design* [10]) e além disso, uma mudança especificamente no domínio de cadastro afetar, por exemplo, o *feed* de notícias, uma vez que para essa mudança ser integrada à aplicação, todo o projeto necessita ser reconstruído para uma nova implantação.

A arquitetura baseada em microsserviços, por sua vez, promove melhorias sobre os pontos negativos mencionados no parágrafo anterior: ela é segmentada por domínio (um serviço ou um conjunto de serviços para cada domínio) e, em relação ao aspecto de implantação da aplicação, é fracamente acoplada - utilizando o exemplo anterior, uma mudança no domínio do cadastro implica na reconstrução e implantação apenas dos serviços relacionados a esse

domínio. Porém, tais vantagens vem ao custo de se manter e escalar uma aplicação que se torna intrinsecamente distribuída, o que é notoriamente mais complexo [11].

2.2 Serviços de Mensageria e o Estilo Publisher-Subscriber

Um cenário comum em uma arquitetura baseada em microsserviços é a de que informações contidas no domínio de um serviço impactem diretamente outros domínios e consequentemente, os serviços pertinentes àquele domínio. Dentro do contexto do exemplo apresentado na subseção anterior, imagine que o usuário altere suas preferências na aplicação da rede social, de modo que essa informação é processada pelo serviço de cadastro. Porém, essa mesma informação é pertinente ao serviço responsável pelo *feed* de notícias, uma vez que ele constrói o *feed* do usuário de acordo com suas preferências. É notório que precisamos estabelecer algum mecanismo de comunicação que atenda a essa necessidade - uma solução para tal é o estilo arquitetural *publisher-subscriber* [12].

O estilo arquitetural publisher-subscriber é utilizado por padrões de integração entre sistemas chamados de Mensageria, onde os componentes publishers são responsáveis por publicar mensagens em um canal ou tópico (comumente em um *broker* de mensagens ou um barramento de eventos) e, por sua vez, os subscribers são responsáveis por se inscrever em um tópico de acordo com suas necessidades. Assim que há uma atualização em um determinado tópico (a partir da publicação de uma mensagem nesse pelo publisher), os subscribers são notificados e realizam as alterações que à eles são pertinentes de acordo com o conteúdo da mensagem publicada.

No nosso exemplo, é fácil perceber que o serviço de cadastro pode ser mapeado para um publisher, de modo que quando o usuário alterar seus gostos/preferências, o serviço enviará uma mensagem para um tópico designado para tal, de modo que o serviço responsável pelo *feed* de notícias será notificado da atualização e realizará as alterações necessárias em seu domínio.

Uma implementação do estilo publisher-subscriber comumente utilizada é o ***Apache Kafka*** [13] (comumente referenciado apenas por *Kafka*). Desenvolvido pelo *LinkedIn* em 2010, trata-se de uma solução escalável, durável, robusta e com tolerância a falhas para o padrão de publish-subscribe e também para *streaming* de eventos.

No *Kafka*, quem assume o papel dos publishers são os chamados *producers* (ou produtores) e analogamente, o papel dos subscribers é assumido pelos chamados *consumers* (ou consumidores). Logo, produtores escrevem mensagens (unidade básica de dados no *Kafka*) para um *Kafka cluster*, que é composto por *brokers* - responsáveis por receber e guardar mensagens publicadas pelos produtores em tópicos. Consumidores consomem dados de um *broker* em posições específicas, correspondente aos tópicos aos quais eles se inscreveram.

Ao analisarmos uma relação entre um produtor, o tópico ao qual ele produz e um consumidor que consome desse tópico, estamos estabelecendo uma relação de *dependência assíncrona* entre esses componentes. Ao analisarmos tal relação em uma arquitetura de microsserviços, em que cada serviço pode conter múltiplos produtores/consumidores *Kafka*, estamos então analisando dependências entre serviços.

2.3 Análise Estática de Código e Reflexão Computacional

Outro conceito importante neste trabalho é o de análise estática de código. A análise estática de código é a análise feita de modo automática no código-fonte do programa e/ou em seu *bytecode* após ser compilado. Ela é utilizada majoritariamente para encontrar *bugs* no código (como é, por exemplo, o caso da ferramenta *FindBugs* [14]), reforçar padrões de código a fim de manter a qualidade desse (aqui, *Linters* [15] são o maior exemplo) e gerar métricas baseadas em elementos detectáveis via o código fonte. Seguiremos com a última abordagem citada, visto que queremos gerar métricas sobre a comunicação assíncrona entre serviços por meio da análise automatizada do código fonte.

Na implementação do algoritmo que analisa as dependências entre serviços via *Kafka*, é utilizado o conceito de **reflexão** [16]. A reflexão em linguagens de programação é a capacidade dessas de modificar, realizar introspecções e modificar sua estrutura e comportamento. No contexto da linguagem de programação Java, utilizada no trabalho, tal capacidade permite a inspeção e instanciação de classes, objetos e métodos cujo os nomes são descobertos em tempo de execução.

2.4 O Projeto *SiteWhere*

Neste trabalho, a análise das dependências assíncronas, via *Kafka*, é realizada especificamente no contexto do projeto ***SiteWhere*** [17]. *SiteWhere* é uma plataforma de código aberto voltada para *Internet of Things* (ou *IoT*) totalmente estruturada em microsserviços e que utiliza *Kafka* como sistema de mensageria. Em sua arquitetura 2.0, o *SiteWhere* utiliza estritamente o modelo publisher-subscriber do *Kafka* (isto é, há uma ausência de *Streams* [18]) descrito acima.

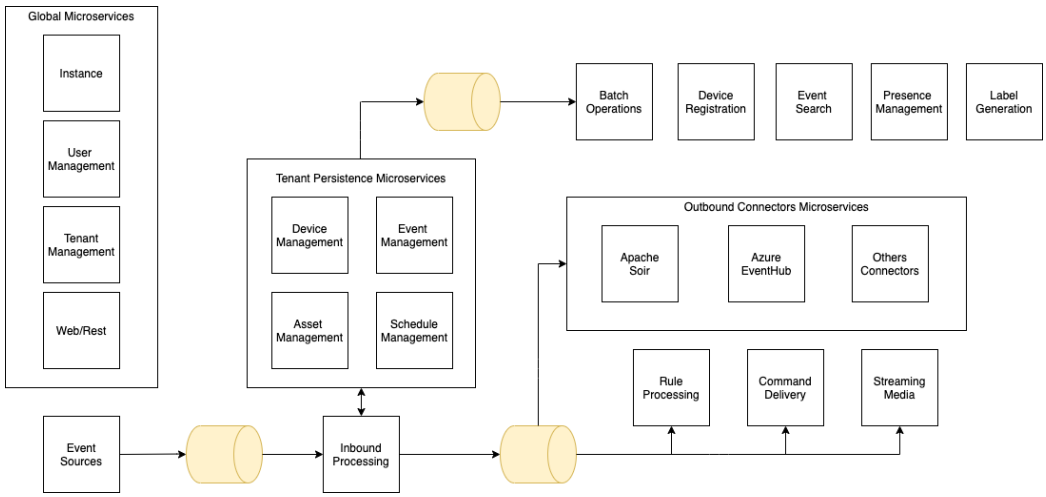

Figura 1: Arquitetura 2.0 do SiteWhere. Adaptado de [17]

Na figura 1, temos o diagrama para a arquitetura 2.0 do *SiteWhere*. Aqui, os microsserviços são delineados por quadrados e os tópicos *Kafka* por cilindros amarelos. Os produtores

Kafka são os microsserviços que possuem arestas que saem deles em direção à tópicos e os consumidores *Kafka* são microsserviços que possuem arestas que advindas dos tópicos e direcionadas a ele. Vale destacar que esse diagrama é a base para a arquitetura 2.0, mas se difere do que de fato se dá em *minor releases*, como é o caso da arquitetura 2.2.0.

3 Método de Trabalho

Para atingir o objetivo de desenvolver uma abordagem conceitual e prática para identificação de dependências assíncronas, via Apache Kafka, no projeto SiteWhere, tomamos os seguintes passos:

- Estudo do repositório, incluindo o código-fonte, existente do projeto em seu repositório do *GitHub*. A forma que o estudo foi conduzido é descrito na Seção 4.
- Estudo das tecnologias envolvidas no projeto *SiteWhere*. Etapa descrita na Seção 4.
- Desenvolvimento de uma solução de análise estática capaz de inspecionar o código-fonte do projeto e estabelecer as relações de dependências entre microsserviços. O viés conceitual da solução é descrito na Seção 4.
- Implementação da solução descrita na etapa anterior, de modo que sua saída seja a representação visual das relações de dependência por meio de um grafo. O detalhamento da implementação se dá na Seção 4.
- Análise dos resultados obtidos com a solução, comparando-os com as relações obtidas na documentação oficial do projeto *SiteWhere* e inspeção manual do código-fonte. Esse processo é descrito na Seção 5.

4 Solução Proposta

Primeiramente, estudamos o repositório do *SiteWhere* buscando identificar e relacionar manualmente produtores e consumidores *Kafka*, visando o reconhecer um padrão estrutural. Concomitantemente, estudamos as tecnologias existentes no projeto, com destaque ao *Apache Kafka*. Após isso, construímos o algoritmo representado pelo pseudocódigo descrito no Algoritmo 1.

Antes de detalhar a implementação do algoritmo, é importante descrever sobre a escolha da abordagem de análise estática frente à análise dinâmica. Optamos pela abordagem estática visto a inerente dificuldade (principalmente em termos de recursos computacionais e dependências) de fazer a implantação de um projeto da complexidade do *SiteWhere*. Ao tomarmos tal decisão (de escolher uma abordagem estática), é importante denotar que aqui passamos a analisar a arquitetura lógica do projeto no lugar de sua arquitetura concreta, isto é, sua "real" arquitetura em tempo de execução.

Para a primeira etapa do algoritmo, há uma inicialização de $Grafo$ com todos os tópicos presentes na classe responsável por conter os tópicos do projeto *SiteWhere*, isto é, a classe `KafkaTopicNaming`. Aqui, cada tópico é colocado na estrutura como "candidato" a aresta, e o mesmo só se torna uma aresta efetiva do grafo se, durante a execução do algoritmo, for relacionado a esse tópico algum microsserviço como vértice, isto é, a ele for relacionado um produtor ou consumidor *Kafka*. Um ponto importante é que o fato dos tópicos estarem contidos em uma classe é claramente uma decisão particular do *SiteWhere*- em outros projetos, é mais comum que os tópicos estejam, por exemplo, em um arquivo de configurações e variáveis de ambiente. Portanto, ao se desenvolver uma futura abordagem genérica, é importante que a estrutura que contenha os tópicos seja fornecida como um parâmetro do algoritmo.

Cada microsserviço no projeto é organizado em um diretório específico. Assim, para os microsserviços em L, buscamos seu diretório e, em seguida, o algoritmo procura um diretório *Kafka*. Essa etapa do algoritmo é relativamente simples, uma vez que os diretórios de serviços seguem o padrão `service-nome-do-serviço` e os diretórios *Kafka* que neles podem estar contidos possuem, em seu nome, `kafka`. Como cada diretório de um microsserviço pode conter vários diretórios aninhados, a busca por um diretório *Kafka* é feita recursivamente.

Ao detectarmos um diretório *Kafka*, analisamos seu arquivos por meio de uma rotina recursiva a fim de determinar se o mesmo é, de fato, um produtor ou consumidor *Kafka*. Nessa rotina, dado um arquivo, obtemos o nome de sua classe e o instanciamos via reflexão computacional. Em seguida, para cada propriedade nele contido, verificamos se alguma delas é uma instância de `KafkaProducer` ou de `KafkaConsumer` (o que ocorre pelo método `Class.isAssignableFrom()`, que funciona como o operador `instanceof` em reflexão) classes base fornecidas pelo *Kafka* para implementação e produtores e consumidores, respectivamente. Caso nenhuma das propriedadas da instância analisada é uma instância de `KafkaProducer` ou de `KafkaConsumer`, chamamos a rotina recursivamente na superclasse da instância analisada. O fato de podermos utilizar da reflexão computacional nessa rotina foi o fator responsável pela escolha da linguagem Java para a implementação do algoritmo.

Uma vez que temos uma classe que de fato é um produtor ou consumidor *Kafka*, analisamos se há uma relação entre os nomes desse e de um tópico presente na estrutura do

$Grafo$ como uma potencial aresta. Isso é feito ao compararmos, palavra a palavra, o nome do produtor/consumidor com as palavras existentes no nome do tópico - e se ao menos 75% das palavras no nome do consumidor/produtor estiverem contidas no nome do tópico, estabelecemos que há uma relação entre esses. Aqui, cada palavra possui como separador uma letra maiúscula, assim como prevê o padrão de nomeação de classes em Java. No caso do tópicos, o separador utilizado pelo *SiteWhere* foi o caractere "-". Também não consideramos o sufixo "Producer" ou "Consumer" na análise. Por exemplo, no microsserviço de `event sources`, há o produtor cujo a classe é chamada `DecodedEventsProducer` e por sua vez, existe o tópico chamado `event-source-decoded-events` - como as palavras `Decoded Events` estão inteiramente contidas no nome do tópico, estabelecemos que ambos possuem uma relação.

Estabelecida uma relação, o tópico deixa de ser um "candidato" a aresta e se torna uma aresta de fato, uma vez que ele contém vertices associados ao mesmo. Apenas arestas "efetivas" são consideradas na impressão do $Grafo$.

Por fim, para a rotina responsável por imprimir as relações do $Grafo$ produzido pelo algoritmo, optou-se por realizar sua implementação na linguagem DOT [19], que se trata de uma linguagem de descrição de grafos em texto puro. Para renderizar o texto gerado em DOT como imagens, a ferramenta GraphVIZ [20] foi escolhida.

Um ponto de destaque é que na implementação do algoritmo, optamos por realizar a estruturação desse a partir de uma classe abstrata (`AbstractParser`), que contém métodos que são comuns às diferentes versões arquiteturais do projeto *SiteWhere* e, por sua vez, classes concretas que são responsáveis por implementar aspectos particulares à cada versão - um exemplo é a classe `ConcreteParserForSecondArchitecture`, implementação concreta de (`AbstractParser` utilizada para analisar a arquitetura 2.2.0 do *SiteWhere*, que é a principal versão analisada no trabalho.

O código-fonte da implementação da solução e sua documentação está disponível em: `https://github.com/JPedroAmorim/pfg`

Entrada P**:** Estrutura de diretórios do projeto *SiteWhere*.
Entrada L**:** Lista de microsserviços aos quais devemos determinar as relações de dependências.

Saída: *Um grafo direcionado, no qual cada vértice representa um microsserviço que possui alguma dependência via Kafka. Cada aresta (arco) representa o tópico pelo qual os serviços se relacionam. Dada uma relação, um arco é direcionado de um microsserviço x para um microsserviço y, onde x é o produtor para o tópico que o arco representa, e y é o consumidor desse mesmo tópico.*

```
Gera-Relações(P, L):
    Grafo ← inicializado com os tópicos presentes na classe de tópicos como
     candidatos à arestas;
    para cada microsserviço presente em L faça
        Identifique seu diretório em P;
        para cada diretório existente dentro do diretório do microsserviço faça
            se houver um diretório de Kafka então
                para cada arquivo dentro do diretório faça
                    se o arquivo é um produtor ou consumidor então
                        se existe alguma relação entre os nomes do produtor ou
                         consumidor com um tópico presente como candidato a aresta
                         no grafo então
                            Grafo ← efetive o tópico como aresta do grafo ao tornar o
                             microsserviço um de seus vértices;
                        fim
                    fim
                fim
            fim
        fim
    fim
    retorna Grafo;
fim

Imprime(P, L):
    Grafo ← Gera-Relações(P, L);
    para cada aresta efetiva do grafo faça
        imprima a aresta e seus vértices em linguagem DOT;
    fim
fim
```

Algoritmo 1: Geração e impressão do grafo de dependências

5 Avaliação

A implementação da solução descrita na seção anterior foi avaliada com base na arquitetura 2.2.0 do projeto *SiteWhere*. Os grafos de dependência produzidos são apresentados nas figuras 2 e 3. A figura 2 possui os microsserviços detectados pelo algoritmo como vértices do grafo, e as arestas são direcionadas de um produtor *Kafka* para um consumidor *Kafka*. Já a figura 3 possui a mesma estrutura, porém, aqui as arestas são rotuladas com os nomes dos tópicos que elas representam.

A solução, executada na IDE (*Integrated Development Environment*) *IntelliJ Idea*, utilizando o *Gradle* como gerenciador de dependências e em uma máquina MacBook Pro 2018 com 8GB de RAM DDR3 e um processador Intel i5 Quad-Core de 2,3 GHz demorou cerca de 42 segundos, sendo esse valor uma média de três execuções. Devemos explicitar que há uma carga considerável em termos computacionais ao realizar a compilação dos serviços do projeto, o que é necessário para a etapa reflexiva do algoritmo.

Na tabela 1, há um resumo das relações *Kafka* que o algoritmo foi capaz de constatar, as relações obtidas na inspeção manual da *codebase* e as relações previstas na documentação do projeto *SiteWhere*. Nela, se a relação foi constatada pelo método há um símbolo de *checkmark* indicando a constatação e um espaço em branco caso contrário. Há um destaque para o serviço de Device State, cujo o algoritmo não foi capaz de constatar porém estava presente na inspeção manual e na documentação do projeto e para o serviço de Device Management,

Serviço	Solução Automatizada	Inspeção Manual	Documentação SiteWhere 2.0
Asset Management			
Batch Operations	✓	✓	✓
Command Delivery	✓	✓	✓
Device Management		✓	
Device Registration	✓	✓	✓
Device State		✓	✓
Event Management	✓	✓	✓
Event Search			
Event Sources	✓	✓	✓
Inbound Processing	✓	✓	✓
Instance Management			
Label Generation			
Outbound Connectors		✓	✓
Rule Processing		✓	✓
Streaming Media			
Schedule Management			

Tabela 1: Tabela com as relações encontradas via inspeção manual, solução automatizada e o que estava previsto na documentação.

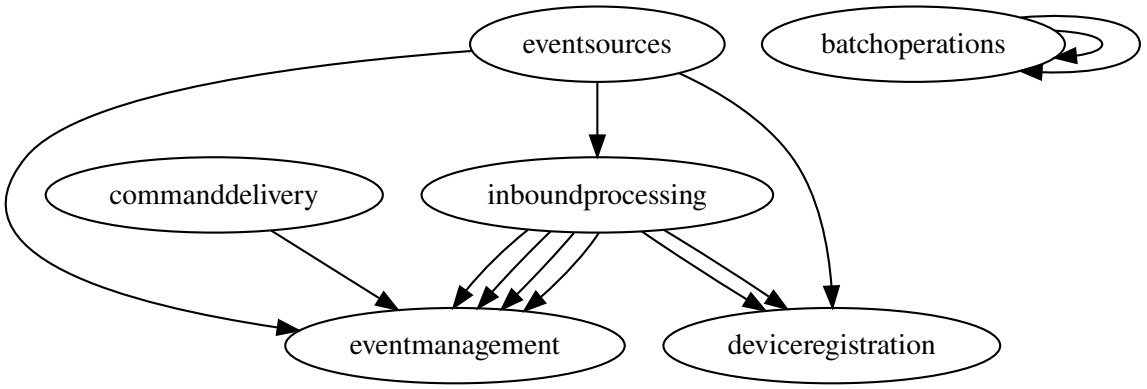

Figura 2: Grafo produzido pelo algoritmo com arestas não rotuladas.

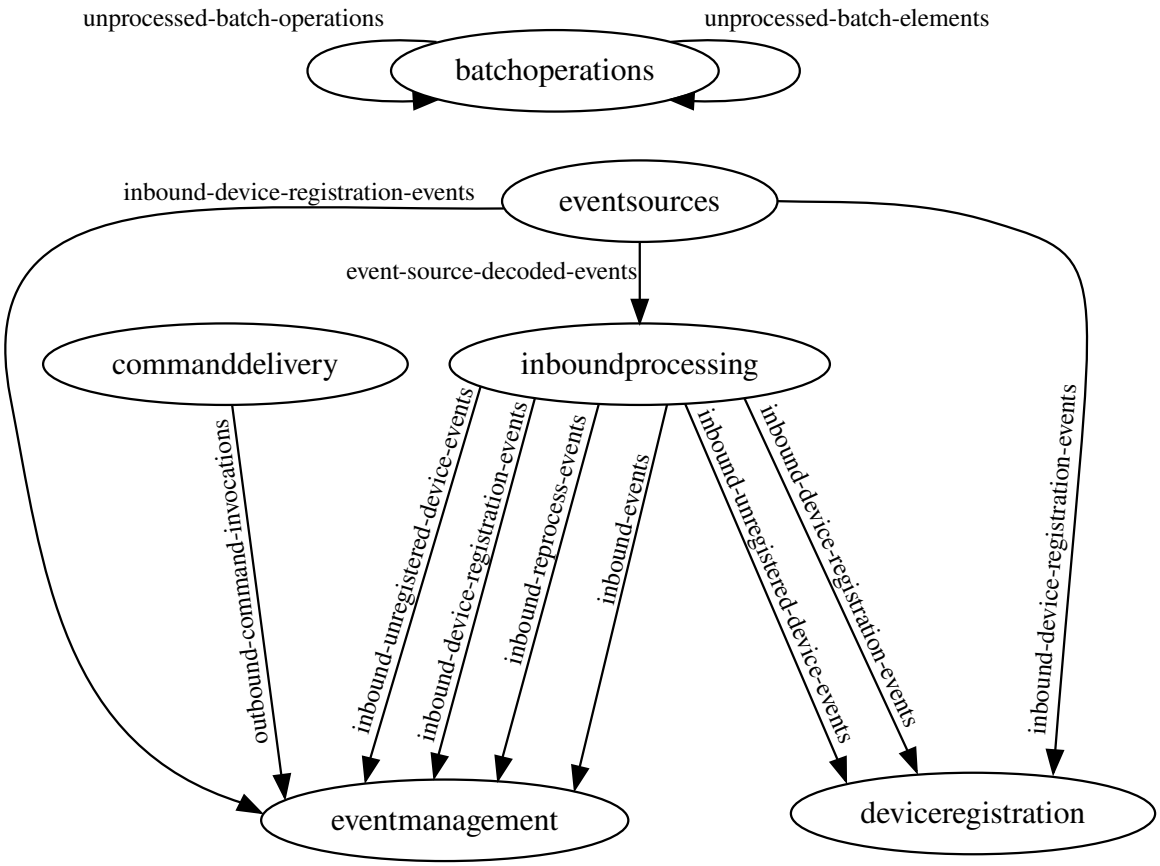

Figura 3: Grafo produzido pelo algoritmo com arestas rotuladas com os tópicos.

que por sua vez, foi constatado apenas pela inspeção manual - ele não estava previsto como um elemento *Kafka* na documentação e tampouco foi identificado pelo algoritmo.

Também temos que dar uma atenção aos serviços de Outbound Connectors e Rule Processing, que por sua vez, fogem do padrão verificado no resto do projeto de consumidor/produtor (o que é constatado na própria documentação) porém ainda possuem módulos *Kafka*. Ambos serviços não foram identificados pelo algoritmo, porém foram verificados manualmente e estavam na documentação.

Em seguida, detalhamos os casos relevantes via inspeção manual e o resultado produzido pelo algoritmo.

Batch Operations

Inspeção do código: Possui módulo *Kafka*.

- Produz para Failed Batch Elements Topic. Esse tópico, pela inspeção de código, não possui consumidores.
- Produz para Unprocessed Batch Elements Topic. Esse tópico possui um único consumidor, sendo ele o próprio serviço de Batch Operations.
- Consome Unprocessed Batch Elements Topic. Esse tópico possui o próprio Unprocessed Batch Elements como produtor.
- Produz para Unprocessed Batch Operations Topic. Esse tópico possui um único consumidor, sendo ele o próprio serviço de Batch Operations.
- Consome Unprocessed Batch Operations Topic. Esse tópico possui o próprio Unprocessed Batch Elements como produtor.

Resultado do algoritmo: Consta no grafo como um vértice que possui arestas que apontam para ele mesmo.

Command Delivery

Inspeção do código: Possui módulo *Kafka*.

- Produz para Undelivered Command Invocations Topic. Esse tópico, pela inspeção de código, não possui consumidores.
- Produz para Outbound Command Invocations Topic. Esse tópico, pela inspeção de código, possui o serviço Event Management como consumidor.

Resultado do algoritmo: Consta no grafo como um vértice que possui arestas apontando para o vértice do serviço de Event Management.

Device Management

Inspeção do código: Possui módulo *Kafka*.

- Produz para Inbound Events Topic. Esse tópico, pela inspeção de código, possui o serviço de Inbound Processing como outro produtor e o serviço de Event Management como consumidor.

Resultado do algoritmo: O algoritmo não foi capaz de relacionar o nome da classe que representa o produtor *Kafka* desse microsserviço com o tópico de Inbound Processing. Portanto, ele não aparece no grafo como um vértice.

Device Registration

Inspeção do código: Possui módulo *Kafka*.

- Consome Device Registration Events Topic. Esse tópico, pela inspeção de código, possui o serviço de Event Sources e o serviço de Inbound Processing como produtores.
- Consome Unregistered Device Events Topic. Esse tópico, pela inspeção de código, possui o serviço de Inbound Processing como produtor.

Resultado do algoritmo: Consta no grafo como um vértice que possui arestas vindo dos vértices que representam os serviços de Event Sources e Inbound Processing.

Device State

Inspeção do código: Possui módulo *Kafka*.

- Consome Outbound Events Topic. Esse tópico, pela inspeção de código, possui o serviço de Event Management como produtor.

Resultado do algoritmo: O algoritmo não foi capaz de relacionar o nome da classe que representa o produtor *Kafka* desse microsserviço com o tópico de Outbound Events. Portanto, ele não aparece no grafo como um vértice.

Event Management

Inspeção do código: Possui módulo *Kafka*.

- Consome Inbound Events Topic. Esse tópico, pela inspeção de código, possui o serviço de Inbound Processing como outro produtor.
- Consome Inbound Reprocess Events Topic. Esse tópico, pela inspeção de código, possui o serviço de Inbound Processing como outro produtor.
- Consome Unregistered Device Events Topic. Esse tópico, pela inspeção de código, possui o serviço de Inbound Processing como outro produtor.

- Consome Device Registration Events Topic. Esse tópico, pela inspeção de código, possui o serviço de Event Sources e o serviço de Inbound Processing como produtores.
- Consome Outbound Command Invocations Topic. Esse tópico, pela inspeção de código, possui o serviço Command Delivery como consumidor.

Resultado do algoritmo: Consta no grafo como um vértice que possui arestas vindo dos vértices que representam os serviços de Event Sources, Inbound Processing e Command Delivery.

Event Sources

Inspeção do código: Possui módulo *Kafka*.

- Produz para Event Sources Decoded Events Topic. Esse tópico, pela inspeção de código, possui o serviço de Inbound Processing como consumidor.
- Produz para Device Registration Events Topic. Esse tópico, pela inspeção de código, possui os serviços de Device Registration e Event Management como consumidores.
- Produz para Failed Decoded Events Topic. Esse tópico, pela inspeção de código, não possui consumidores.

Resultado do algoritmo: Consta no grafo como um vértice que possui arestas apontando para os vértices que representam os serviços de Device Registration, Event Management e Inbound Processing.

Inbound Processing

Inspeção do código: Possui módulo *Kafka*.

- Produz para Inbound Events Topic. Esse tópico, pela inspeção de código, possui o serviço de Event Management como consumidor.
- Produz para Inbound Reprocess Events Topic. Esse tópico, pela inspeção de código, possui o serviço de Event Management como consumidor.
- Produz para Unregistered Device Events Topic. Esse tópico, pela inspeção de código, possui os serviços de Event Management e Device Registration como consumidores.
- Produz para Device Registration Events Topic. Esse tópico, pela inspeção de código, possui os serviços de Device Registration e Event Management como consumidores.
- Consome Event Sources Decoded Events Topic. Esse tópico, pela inspeção de código, possui o serviço de Event Sources como produtor..

Resultado do algoritmo: Consta no grafo como um vértice que possui arestas apontando para os vértices que representam os serviços de Device Registration, Event Management. Há também uma aresta vinda do vértice que representa o serviço de Event Sources.

Outbound Connectors

Inspeção do código: Possui módulo *Kafka*, porém, os arquivos neles contidos não se enquadram estritamente como Produtores/Consumidores.

Resultado do algoritmo: Não consta no grafo como vértice.

Rule Processing

Inspeção do código: Possui módulo *Kafka*, porém, os arquivos neles contidos não se enquadram estritamente como Produtores/Consumidores.

Resultado do algoritmo: Não consta no grafo como vértice.

Instance Management

Inspeção do código: Possui módulo *Kafka*, porém, os arquivos neles contidos não se enquadram estritamente como Produtores/Consumidores.

Resultado do algoritmo: Não consta no grafo como vértice.

6 Conclusão e Trabalhos Futuros

Podemos constatar a partir da tabela que o algoritmo foi capaz de efetivamente mapear a maioria das dependências via *Kafka* contidas no *SiteWhere*- das 16 relações previstas pela inspeção manual do código em conjunto com a documentação, fomos capazes de corretamente produzir 12 delas. Todavia, ainda há espaço para melhora a fim de aprimorar o algoritmo para que esses quatro casos passem a ser detectados.

Para os casos de falha de detecção do algoritmo nos serviços de Device Management e Device State, a falha ocorre na etapa em que se busca relacionar, de modo sintático, o nome do Produtor/Consumidor presente no arquivo com os tópicos existentes na classe de resolução de tópicos do projeto. É passível de, em uma nova iteração do algoritmo, tomarmos mais algum método de relacionar tais entidades sem ser a partir de uma simples correlação entre os nomes das mesmas.

Temos também que o algoritmo não é capaz de identificar estruturas que fogem do modelo Produtor/Consumidor - o que é perceptível ao analisarmos os serviços de Outbound Connectors e Rule Processing. Esse se torna um problema grave em novas versões do Apache *Kafka*, que cada vez mais vale-se da estrutura de Streams - um exemplo claro em que o algoritmo performa mal em mapear relações é o caso da arquitetura 3.0.0 do *SiteWhere* em diante, visto que a mesma pauta-se mais na análise de *Streams* do que o modelo Produtor/Consumidor.

Outro ponto a ser considerado para próximas iterações é o refinamento da estrutura Abstrata/Concreta - a estrutura abstrata, por mais que proveu genericidade entre versões do *SiteWhere*, ainda está muito acoplada à própria organização do projeto, visto que a mesma (em termos da organização de diretórios e padrão de nomeação das classes/tópicos) não se altera drasticamente entre as versões arquiteturais do projeto.

Quanto à análise do acoplamento da versão 2.2.0 do *SiteWhere*, podemos constatar que o serviço de Inbound Processing contém um número expressivo de relações - ele está presente em 7 das 16 relações previstas. Pelo grafo gerado, é fácil perceber um alto grau de acoplamento a esse serviço, com destaque ao serviço de Event Management - que é fortemente dependente do consumo de tópicos produzidos pelo Inbound Processing. Tal situação é um forte *architecture smell* [21] do ponto de vista do acoplamento, e deve ser considerada uma refatoração desse serviço a fim de melhorar a manutenibilidade do sistema como um todo.

Referências

[1] Pooyan Jamshidi et al. "Microservices: The journey so far and challenges ahead". Em: *IEEE Software* 35.3 (2018), pp. 24–35.

[2] *Microservices Adoption in 2020*. O'Reilly. URL: https://www.oreilly.com/radar/microservices-adoption-in-2020/.

[3] Gregor Hohpe e Bobby Woolf. *Enterprise integration patterns: Designing, building, and deploying messaging solutions*. Addison-Wesley Professional, 2004.

[4] Justus Bogner, S. Wagner e Alfred Zimmermann. "Automatically measuring the maintainability of service- and microservice-based systems: a literature review". Em: *Proceedings of 27th International Workshop on Software Measurement and 12th International Conference on Software Process and Product Measurement*. Out. de 2017. DOI: 10.1145/3143434.3143443.

[5] Shang-Pin Ma et al. "Using service dependency graph to analyze and test microservices". Em: *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*. Vol. 2. IEEE. 2018, pp. 81–86.

[6] Daniel Rodrigo de Freitas Apolinário e Breno Bernard Nicolau de França. "Towards a method for monitoring the coupling evolution of microservice-based architectures". Em: *Proceedings of the 14th Brazilian Symposium on Software Components, Architectures, and Reuse*. 2020, pp. 71–80.

[7] James Lewis e Martin Flower. *Microservices, a definition of this new architectural term*. Mar. de 2014. URL: https://martinfowler.com/articles/microservices.html.

[8] *O que é uma CDN?* Cloudfare. URL: https://www.cloudflare.com/pt-br/learning/cdn/what-is-a-cdn/.

[9] *API Gateway Pattern. Backends for Frontends*. URL: https://microservices.io/patterns/apigateway.html.

[10] Martin Flower. *DomainDrivenDesign*. Abr. de 2020. URL: https://martinfowler.com/bliki/DomainDrivenDesign.html.

[11] Anand Ranganathan e Roy H. Campbell. "What is the complexity of a distributed computing system?" Em: *Complexity* 12.6 (jul. de 2007), pp. 37–45. DOI: 10.1002/cplx.20189.

[12] *What is Pub/Sub Messaging?* Amazon. URL: https://aws.amazon.com/pt/pub-sub-messaging/.

[13] *Apache Kafka*. Apache Software Foundation. URL: https://kafka.apache.org/.

[14] *FindBug Static Code Analysis tool*. University of Maryland. URL: http://findbugs.sourceforge.net/.

[15] *What is a Linter and why your team should use it*. Sourcelevel. URL: https://sourcelevel.io/blog/what-is-a-linter-and-why-your-team-should-use-it.

[16] Glen McCluskey. *Using Java Reflection*. Oracle. Jan. de 1998. URL: https://www.oracle.com/technical-resources/articles/java/javareflection.html.

[17] *SiteWhere*. SiteWhere LLC. URL: https://sitewhere.io/en/.

[18] *Kafka Streams*. Apache Software Foundation. URL: https://kafka.apache.org/documentation/streams/.

[19] *The DOT Language*. Graphviz. URL: https://graphviz.org/doc/info/lang.html.

[20] *Graph Visualization Software*. Graphviz. URL: https://graphviz.org/.

[21] Joshua Garcia et al. "Toward a Catalogue of Architectural Bad Smells". Em: *Proceedings of the Fifth International Conference on Quality of Software Architectures: Architectures for Adaptive Software Systems*. Vol. 5581. Lecture Notes in Computer Science. Springer International Publishing, 2009, pp. 146–162. ISBN: 978-3-642-02350-7. DOI: 10.1007/978-3-642-02351-4_10.