Pontifícia Universidade Católica
do Rio de Janeiro

Pedro Felipe Santos Magalhães

Uma biblioteca para testes determinísticos em sistemas distribuídos com comunicação assíncrona

Dissertação de Mestrado

Dissertação apresentada como requisito parcial para a obtenção do grau de Mestre pelo Programa de Pós–graduação em Informática, do Departamento de Informática da PUC-Rio .

Orientador : Prof. Markus Endler
Co-orientador: Dra. Maria Julia Dias de Lima

Rio de Janeiro
Abril de 2023

Pontifícia Universidade Católica
do Rio de Janeiro

Pedro Felipe Santos Magalhães

Uma biblioteca para testes determinísticos em sistemas distribuídos com comunicação assíncrona

Dissertação apresentada como requisito parcial para a obtenção do grau de Mestre pelo Programa de Pós–graduação em Informática da PUC-Rio . Aprovada pela Comissão Examinadora abaixo:

Prof. Markus Endler
Orientador
Departamento de Informática – PUC-Rio

Dra. Maria Julia Dias de Lima
TECGRAF-PUC-Rio

Profª. Noemi de La Rocque Rodriguez
Pesquisadora Autônoma

Profª. Silvana Rossetto
UFRJ

Rio de Janeiro, 20 de Abril de 2023

Pedro Felipe Santos Magalhães

Graduado em ciência da computação pela Universidade PUC-Rio.

Ficha Catalográfica

Santos Magalhães, Pedro Felipe

Uma biblioteca para testes determinísticos em sistemas distribuídos com comunicação assíncrona / Pedro Felipe Santos Magalhães; orientador: Markus Endler; co-orientador: Maria Julia Dias de Lima. – 2023.

67 f: il. color. ; 30 cm

Dissertação (mestrado) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática, 2023.

Inclui bibliografia

1. Informática – Teses. 2. Testes. 3. Sistemas Distribuídos. 4. Microsserviços. 5. Go. 6. Fila de Mensagens. I. Endler, Markus. II. Dias de Lima, Maria Julia. III. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. IV. Título.

CDD: 004

Aos meus pais, a minha irmã, minha esposa e minha família
pelo apoio e encorajamento.

Agradecimentos

Às minhas orientadoras Noemi Rodriguez e Maria Julia de Lima por todo apoio e parceria na a realização deste trabalho.

À CAPES e à PUC-Rio, pelos auxílios concedidos, sem os quais este trabalho não poderia ter sido realizado.

Aos meus amigos por todo apoio, paciência e compreensão.

Aos meus pais, pela educação, atenção e carinho de todas as horas.

Aos professores Alessandro Garcia, Luís Fernando Seibel, Marco Molinaro, Marcelo Gattas, Noemi Rodriguez que desde a graduação me mantiveram empolgado com os desafios da informática.

Aos meus colegas do Tecgraf em especial Carla, Carlos, Daniel, Felipe, Isabella, Rafael e Rodnei. Por toda a paciência e preocupação em ajudar no meu desenvolvimento com desenvolvedor.

Aos meus colegas da PUC-Rio em especial Bianca, Daniel, Hugo e Maurício.

Aos professores que participaram da Comissão examinadora.

A todos os professores e funcionários do Departamento pelos ensinamentos e pela ajuda.

A todos os amigos e familiares que de uma forma ou de outra me estimularam ou me ajudaram.

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001.

Resumo

Santos Magalhães, Pedro Felipe; Endler, Markus; Dias de Lima, Maria Julia. **Uma biblioteca para testes determinísticos em sistemas distribuídos com comunicação assíncrona**. Rio de Janeiro, 2023. 67p. Dissertação de Mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Observamos que cada vez mais desenvolvedores estão adotando a arquitetura de microsserviços para o desenvolvimento de sistemas distribuídos. Usualmente nesse tipo de arquitetura há um serviço de fila de mensagens que fica responsável em fazer a comunicação assíncrona entre os microsserviços, um serviço bastante utilizado para isso é o Kafka. Nesse ambiente assíncrono, os testes de integração de um determinado serviço ficam complexos pela dificuldade de criar cenários reprodutíveis. No nosso trabalho propomos e avaliamos o uso de uma biblioteca em Go que ajuda no desenvolvimento de testes de integração para microsserviços que utilizam Docker e Kafka, garantindo a ordenação de eventos nos cenários de teste desenvolvidos.

Palavras-chave

Testes; Sistemas Distribuídos; Microsserviços; Go; Fila de Mensagens.

Abstract

Santos Magalhães, Pedro Felipe; Endler, Markus (Advisor); Dias de Lima, Maria Julia (Co-Advisor). **A library for deterministic tests in distributed systems with asynchronous communication**. Rio de Janeiro, 2023. 67p. Dissertação de Mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Nowadays more and more developers are adopting the microservices architecture for the development of distributed systems. Usually in this type of architecture there is a message queue service that is responsible for asynchronous communication between microservices; a service that is widely used for this is the Apache Kafka. In this asynchronous environment, integration tests for a given service become complex due to the difficulty of creating reproducible scenarios. In our work, we propose and evaluate the use of a library we developed in Go for the construction of integration tests for microservices that use Docker and Kafka, guaranteeing in the ordering of events as described in the test script.

Keywords

Tests; Distributed Systems; Microservices; Go; Message Queuing.

Sumário

Lista de figuras

Lista de Códigos

Tell me and I forget; teach Me and I may remember; involve me and I learn

Xun Kuang, *The Achievements of the Ru.*

1
Introdução

Nos últimos anos acompanhamos um grande crescimento no número de sistemas distribuídos. Isso porque, dentre outros fatores, estamos vendo a popularização da computação na nuvem, o aumento de poder computacional dos dispositivos móveis, o aumento do número de pessoas conectadas à internet, além da contínua expansão e melhoria da infraestrutura da rede mundial de computadores. Ano após ano a latência na comunicação entre dispositivos distantes se torna menor e a capacidade de transferir dados se torna maior.

Outro fator que pode estar contribuindo para o crescimento de sistemas distribuídos é a adoção de uma arquitetura chamada de microsserviços (NAMIOT; SNEPS-SNEPPE, 2014). Nessa arquitetura, cada parte do sistema tem uma fronteira bem definida e executa de forma independente dos outros componentes, facilitando assim que eles possam ser replicados de forma individual e fiquem distribuídos em um ambiente de rede. Por exemplo, em um sistema em que ocorrem muito mais leituras do que escritas, se o serviço responsável pela escrita for um microsserviço independente do de leitura, podemos criar mais réplicas apenas do serviço de leitura para atender a demanda sem precisar aumentar as réplicas do serviço de escrita. Além disso, em alguns cenários é possível distribuir as réplicas geograficamente para atender mais rapidamente usuários de diferentes localidades.

Nesses tipos de sistemas distribuídos baseados em microsserviços, é comum que parte da comunicação entre os microsserviços seja feita através de *frameworks* de filas de mensagem (STEEN; TANENBAUM, 2017a), comumente usando o paradigma conhecido como *Publish/Subscribe* (EUGSTER et al., 2003). Nessa arquitetura, cada serviço pode ler e escrever na fila de mensagem de forma independente, permitindo assim que os serviços interajam uns com os outros de forma assíncrona. No contexto deste trabalho, estamos interessados em sistemas que utilizam a comunicação através de fila de mensagens, usando plataformas de mensageria, com a arquitetura *Publish/Subscribe*.

Os testes desse tipo de sistema geralmente são bastante complexos devido ao grande número de variáveis envolvidas. Um dos desafios ocorre por causa da comunicação dos serviços via mensageria ser feita de forma assíncrona, tornando complicado garantir a ordem de determinados eventos do sistema. Essa assincronia de produção e de consumo de mensagens acaba tornando complicado reproduzir uma sequência de eventos de forma consistente em um sistema distribuído. Outra questão decorre do fato de que cada microsserviço

pode falhar de forma independente dos outros por problemas de software ou de hardware podendo, portanto, aumentar consideravelmente o número de cenários de testes de um sistema de acordo com o número de microsserviços.

Um outro problema nessa arquitetura de sistemas distribuídos com comunicação assíncrona é determinar o estado geral de um sistema no momento que um erro ocorre (CHANDY; LAMPORT, 1985). Essa questão não será tratada no nosso trabalho; vamos partir de um estado inicial conhecido e gerar eventos ordenados definidos pelo desenvolvedor através do *script* de teste.

Existem muitas ferramentas e metodologias para testes desse tipo de sistema. Uma das técnicas utilizadas para garantir a repetibilidade de cenários de testes é o uso de scripts programáveis (JIANG; HASSAN, 2015), em que o desenvolvedor pode definir o comportamento dos testes utilizando a linguagem de programação oferecida pelo *driver* de teste utilizado.

Em sistemas distribuídos, podem ocorrer diversos tipos de falha no sistema, sendo elas nos próprios serviços, por falha de software, ou na infraestrutura de hardware como problemas de rede. É comum que nesse tipo de sistemas existam mecanismos para lidar com falhas denominadas *fail-stop*, em que o sistema, ou parte dele, tem uma falha completa e para de executar. Neste tipo de falha, se a parte do sistema que falhou volta a funcionar, ela é tratada como uma nova instância.

No nosso trabalho criamos uma biblioteca de apoio a testes buscando facilitar a ordenação de estados do sistema durante a execução de um teste. Nossa biblioteca procura ajudar o desenvolvedor a rodar o sistema sob teste fornecendo mecanismos para que o desenvovedor consiga criar uma determinada ordenação em seu script de testes. Com isso, buscamos permitir que seja possível testar determinados requisitos de um serviço, por exemplo, em que o desenvolvedor podería definir no script de testes uma determinada ordem de eventos do sistema e verificar que o comportamento do serviço sob teste é o esperado. Além disso, precisamos que os testes sejam sempre reproduzíveis, ou seja, sempre produzindo o mesmo resultado a cada vez que é executado. Para alcançar isso, buscamos um modelo de programação apropriado para permitir ao desenvolvedor estabelecer essa ordem de eventos. Vamos avaliar o uso da biblioteca para criação de testes em cenários de dois estudos de caso que adotam a arquitetura de microsserviços. Essa análise busca identificar em quais cenários a biblioteca traz benefícios ao desenvolvedor e em quais ela não é capaz de ajudar. Além disso, vamos avaliar o uso da linguagem de programação escolhida, Go[1], para implementar a biblioteca.

No capítulo 2, vamos caracterizar melhor o tipo de sistemas em que

[1]<https://go.dev/>

estamos interessados, mostrando um pouco mais sobre a arquitetura de microsserviços e sobre filas de mensagens. Vamos falar também, sobre testes de sistemas distribuídos e sobre os tipos de falhas. Em seguida, no capítulo 3 falaremos sobre ferramentas de testes, comentando sobre as implicações da escolha de uma ferramenta e da linguagem utilizada. Ao final do capítulo, falaremos sobre a biblioteca desenvolvida. A seguir, no capítulo 4 avaliamos nossa biblioteca realizando testes de microsserviços em projetos sendo desenvolvidos no Tecgraf PUC-Rio. Por fim, temos as conclusões no 5.

2
Fundamentos

No nosso trabalho, estamos interessados em sistemas distribuídos formados por conjuntos de microsserviços com comunicação assíncrona. Em especial, neste trabalho, vamos utilizar sistemas que se comunicam via filas de mensagens para nossos testes. Esse tipo de arquitetura tem sido bastante utilizada por delegar questões complexas de comunicação, como confiabilidade e segurança, para serviços especializados (WISKA et al., 2016; DU et al., 2018; BANO et al., 2022b; BANO et al., 2022a). Neste capítulo descrevemos os conceitos de microsserviços, na seção 2.1, e de fila de mensagens, na seção 2.2. Em seguida, na 2.3, falamos sobre testes de sistemas distribuídos. Por fim, na seção 2.4 apresentamos uma taxonomia de falhas.

2.1
Aplicações baseadas em microsserviços

A arquitetura de microsserviços é um estilo de arquitetura de software em que uma aplicação é construída a partir de pequenos serviços independentes, cada um executando um processo específico e se comunicando com outros serviços por meio de uma rede. Esses serviços são projetados para serem escaláveis, resilientes e altamente disponíveis, o que permite que as aplicações possam ser facilmente adaptadas às mudanças de requisitos e condições de carga.

Uma das características comuns das aplicações que usam a arquitetura de microsserviços é o uso de serviços de mensageria para a comunicação entre os serviços. Esses serviços de mensageria permitem que os serviços possam se comunicar de forma assíncrona, sem depender da disponibilidade do serviço de destino, e garantindo uma entrega confiável de mensagens.

Em geral os maiores benefícios desse tipo de arquitetura são: escalabilidade, resiliência, flexibilidade e manutenibilidade. Ao utilizar serviços independentes e assíncronos, os desenvolvedores podem escalar serviços de forma mais eficiente e lidar com falhas de forma mais resiliente, enquanto mantêm a flexibilidade de atualizar e gerenciar serviços individualmente (DRAGONI et al., 2017).

A arquitetura de microsserviços permite que os serviços possam ser escalados independentemente, o que significa que os serviços mais demandados podem ser escalados horizontalmente para lidar com a carga adicional, enquanto

os serviços menos demandados podem ser mantidos com menor número de instâncias.

Além disso, a arquitetura também permite que os serviços possam ser desenvolvidos, implantados e gerenciados independentemente, o que agiliza as entregas pelas equipes de desenvolvimento.

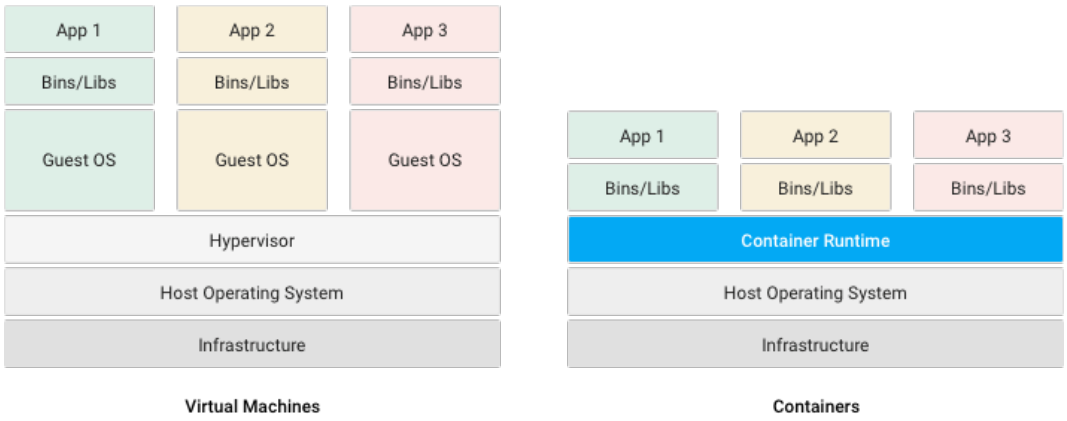
Como os serviços são independentes uns dos outros, a manutenção e atualização de um serviço não afeta os outros serviços da aplicação, o que torna a manutenção e atualização mais fácil e menos arriscada.

Serviços desse tipo são tipicamente executados em *containers* (BERNSTEIN, 2014) para possibilitar o isolamento dos recursos utilizados e facilitar a implantação de cada microsserviço em uma ou mais máquinas. Em infraestruturas que utilizam gerenciadores de *containers*, como o *Kubernetes* (BERNSTEIN, 2014) a replicação desses serviços que compõem o sistema fica simplificada. Atualmente, uma implementação de *containers* muito usada é a do *Docker*[1].

Containers

Os *containers* oferecem um mecanismo de empacotamento de aplicativos que permite abstrair o ambiente em que o aplicativo vai ser executado, desacoplando assim, a plataforma de execução do desenvolvimento do software. São muito similares às máquinas virtuais, mas são projetados para ter um menor impacto de desempenho na máquina física que está rodando os *containers*.

Figura 2.1: Máquina Virtual x Container



Dessa forma, basta o desenvolvedor definir o *container* de sua aplicação para que ela possa executar em qualquer outra máquina que utilize essa tecnologia. Ao iniciar a aplicação, a máquina executa o *container* e não a aplicação diretamente. O *container* de uma aplicação inclui todas as suas

[1] <https://www.docker.com/>

dependências, como pacotes e bibliotecas, representadas na figura 2.1 como 'Bins/Libs'.

A tecnologia de *containers* está sendo muito usada em aplicações baseadas em microsserviços. No nosso trabalho, a utilização dessa tecnologia foi importante para evitar dificuldades com o ambiente de execução necessário para iniciar cada serviço sob teste. Cada serviço que utilizamos como exemplo foi desenvolvido em linguagens diferentes, possuem dependências diversas e mesmo assim durante o teste bastou executar o *container* de cada um deles.

2.2 Filas de mensagens

A comunicação por fila de mensagens utiliza uma arquitetura em que os produtores e consumidores de mensagens não se comunicam diretamente. Ao invés disso, utilizam um serviço intermediário que centraliza a comunicação e fica responsável por receber mensagens dos produtores e encaminhá-las para os consumidores. Esse tipo de arquitetura traz diversos benefícios, principalmente o desacoplamento e a assincronia entre consumidores e produtores.

O desacoplamento entre produtores e consumidores permite que os produtores não conheçam as aplicações consumidoras para produzir as mensagens, independentemente do número ou da localização dos consumidores. O produtor interage apenas com o serviço responsável pela fila que é também o ponto de acesso aos consumidores. Outra vantagem dessa abordagem é o assincronismo entre a produção e o consumo de mensagens, permitindo que os produtores não fiquem limitados pela velocidade de consumo. Como a mensagem é armazenada na fila de mensagens, o produtor pode produzir mensagens mesmo que nenhum consumidor esteja conectado.

Tipicamente o serviço de mensagens permite que os consumidores e produtores usem canais de comunicação independentes para criar uma separação entre as mensagens. Assim, consumidores cadastrados em um canal de comunicação só recebem as mensagens produzidas para aquele canal e não são notificados sobre mensagens em outros canais.

A comunicação assíncrona com serviços de mensageria permite que os serviços possam lidar com falhas de forma mais resiliente, pois a mensagem será armazenada em uma fila e será entregue assim que o serviço estiver disponível novamente.

Eugster(EUGSTER et al., 2003) afirma que o uso do modelo *Publish-Subscribe*, intimamente ligado a arquiteturas de fila de mensagens, traz um desacoplamento em três níveis. Como as aplicações que produzem e consomem mensagens não precisam se conhecer para trocarem mensagens, o autor fala

que existe um *desacoplamento espacial*. Além disso, como o produtor e o consumidor executam de forma independente e possivelmente em momentos diferentes, dependendo apenas do serviço de mensagens, o produtor pode produzir todas as mensagens enquanto os consumidores não estão executando, em seguida parar de executar e mesmo assim todas as mensagens estarão prontas para serem enviadas aos consumidores que comecem a executar. Para o autor esse seria o *desacoplamento no tempo*. Por fim, ele fala sobre o *desacoplamento de sincronização*, mostrando que nessa arquitetura o produtor não é bloqueado ao produzir mensagens, ou seja, ainda que seja necessário lidar com mensagens produzidas de forma paralela entre produtores diferentes, quem vai tratar isso é o serviço de mensagens sem bloquear os produtores. Da mesma forma, os consumidores não ficam bloqueados aguardando que mensagens sejam produzidas. Em geral, as APIs permitem o cadastro de *callbacks* que são acionadas apenas quando alguma mensagem chega no canal de interesse do consumidor.

Kafka

O *Kafka* é um sistema de fila de mensagens criado no LinkedIn para fazer o processamento de logs (KREPS et al., 2011). Nesse sistema, os dados são armazenados na forma de tópicos. Aplicações produtoras podem escrever mensagens nesses tópicos e essas são repassadas para as aplicações consumidoras que estão cadastradas no tópico apropriado. O *Kafka* permite que consumidores e produtores se cadastrem nos tópicos de forma dinâmica, lidando bem com a conexão e desconexão de qualquer um deles. É possível definir diferentes tipos de configurações para cada tópico para ajustá-lo a diferentes cenários de comunicação. Podemos definir o nível de confirmação de entrega para balancear o desempenho e a confiabilidade da entrega das mensagens dos produtores. Por exemplo, em um cenário em que a perda de algumas mensagens em um determinado tópico não importa, o usuário pode optar por nenhuma confirmação de entrega nas mensagens, obtendo assim uma entrega mais rápida, enquanto em um cenário mais crítico podemos optar pela confirmação completa em que o *Kafka* informa ao produtor que o tópico recebeu a mensagem apenas quando todas as réplicas existentes tiverem armazenado a mensagem.

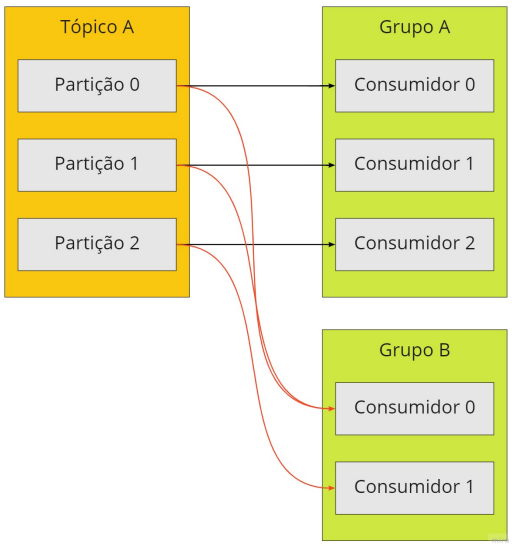
Para permitir um melhor balanceamento de carga no consumo de mensagens em um determinado tópico, o *Kafka* permite a criação de *grupos de consumidores*, onde cada mensagem enviada no tópico é recebida por apenas um membro desse grupo. Dessa forma, o trabalho pode ser distribuído entre os diferentes consumidores de um grupo permitindo um paralelismo no trata-

mento das mensagens. Para fazer parte de um determinado grupo basta que o consumidor indique o identificador do grupo ao se cadastrar em um tópico. O *Kafka* permite a entrada e saída de consumidores do grupo a qualquer momento.

Para fazer a divisão de carga, o *Kafka* subdivide os tópicos em partições e essas partições são divididas entre os consumidores de um grupo; cada mensagem produzida é enviada para apenas uma partição e apenas o consumidor atribuído àquela partição recebe a mensagem correspondente. Dessa forma, o número de partições de um tópico define o número máximo de consumidores ativos em um grupo de consumidores. O número de partições de um tópico é definido em sua criação. Quando um consumidor entra ou sai do grupo, o *Kafka* redistribui as partições entre os membros. O produtor pode especificar uma partição destino para cada mensagem ou definir uma chave de distribuição para que o próprio *Kafka* fique responsável pela distribuição das mensagens para as partições.

Na figura 2.2 temos um exemplo de um tópico com três partições e dois grupos de consumidores. Nesse exemplo, o Grupo A tem um consumidor para cada partição e o Grupo B tem um consumidor com duas partições e um outro com uma partição. Caso uma mensagem chegue ao Tópico A e seja alocada na partição 0, a mensagem será encaminhada apenas para o Consumidor 0 de cada grupo enquanto os outros consumidores permanecerão em estado de espera.

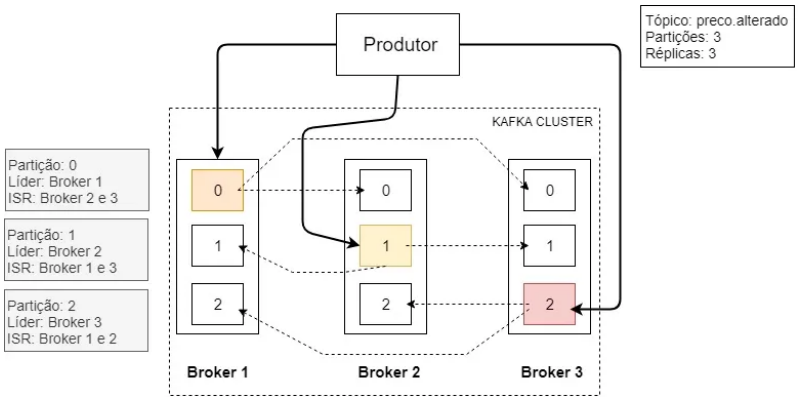
Figura 2.2: Grupos de consumidores do Kafka



Além disso, é possível que os tópicos estejam distribuídos em um ou mais servidores, chamados de *brokers*. Os *brokers* são instâncias do servidor do Kafka, usualmente chamados de *nós*, e são responsáveis por armazenar

os dados das partições dos tópicos, permitindo a escrita pelos produtores e a leitura por consumidores. Quando existe mais de um *broker*, os dados das partições podem ser replicados. Na figura 2.3 temos o exemplo de um tópico com três partições usando um fator de replicação três. O *Kafka* coordena os *brokers* de forma que, no caso de indisponibilidade de uma ou mais instâncias, o sistema continue funcionando. Nesse exemplo, cada *broker* é responsável por uma partição e possui uma cópia das outras 2 partições. Dessa forma, o serviço pode perder até dois de seus *brokers* e continuar operando normalmente.

Figura 2.3: Replicação de partições



Observamos que o *Apache Kafka* vem sendo muito utilizado nas aplicações com a arquitetura em que estamos interessados (BANO et al., 2022a; BANO et al., 2022b; DU et al., 2018; WISKA et al., 2016). Os sistemas que vamos usar como base para os testes de aplicações baseadas em microsserviço utilizam o *Kafka* para fazer a comunicação entre os serviços.

Nos trabalhos de Saira Bano(BANO et al., 2022a; BANO et al., 2022b), os autores avaliam o uso do *Kafka* em um sistema em que os produtores e consumidores podem sofrer constantes desconexões. Eles apontam como benefícios de usar o *Kafka* e da arquitetura de mensageria o fato deste tipo de sistema resolver os problemas de acoplamento entre produtores e consumidores e conseguir lidar com a conexão e desconexão de produtores ou consumidores. Isso tira da aplicação a responsabilidade de tratar esses casos.

Já Yuheng Du(DU et al., 2018) traz em seu artigo o uso da escalabilidade do *Kafka* com uma análise detalhada de latência e de taxa de transferência com diferentes cenários de replicação e quantidade de clientes e produtores conectados ao Kafka. Nesse artigo o autor usa uma aplicação de veículos conectados que enviam constantemente informações de seus sensores para servidores utilizando o *Kafka* que precisa lidar com a grande quantidade de dados produzida por cada veículo. Além da escalabilidade e desempenho, os autores comentam terem resolvido um problema de replicação de dados ao

usarem o Kafka, uma vez que toda a persistência de mensagens agora era feita pelo Kafka.

Em outro trabalho, Rindra Wiska e outros(WISKA et al., 2016) avalia o uso do *Kafka* em uma aplicação de monitoramento do nível de CO^2 atmosférico com alto volume de mensagens trocadas entre produtores e consumidores. Nesse trabalho eles também avaliam o desempenho do *Kafka* ao lidar com o grande volume de dados.

2.3 Testes de aplicações distribuídas

Em sistemas distribuídos apenas a corretude de cada componente não garante o funcionamento do sistema, uma vez que diversas variáveis externas ao código são introduzidas aos sistemas quando os tornamos distribuídos. A rede, por exemplo, se torna uma variável importante que pode alterar o comportamento do sistema em caso de falha ou lentidão, ou a configuração incorreta de determinados serviços do sistema, além de possíveis incompatibilidades de versões entre os componentes do sistema. Outras características podem aumentar a complexidade do teste de sistemas, como a existência de réplicas em diversos microsserviços independentes que são escalonados em máquinas diferentes. Essas características fazem com que exista um não determinismo no estado do sistema e torna mais difícil reproduzir de maneira consistente determinados cenários de testes. Por esse motivo, criar testes de sistema para esse tipo de software demanda um apoio de ferramentas e metodologias que ajudem o desenvolvedor a lidar com a complexidade de testes em sistemas distribuídos.

Uma abordagem de teste de sistema que surgiu nos últimos anos é a chamada Engenharia do Caos (*Chaos Engineering*) (BASIRI et al., 2016) que busca criar falhas aleatórias nos sistemas para verificar sua robustez. Algumas empresas que têm a viabilidade do negócio dependente da alta disponibilidade de seus sistemas passaram a rodar os testes também no ambiente de produção. Segundo eles, seria impraticável simular, em um ambiente de testes, todas as variáveis envolvidas do sistema inteiro. Por isso, deixam um serviço rodando em produção que, dentre outras coisas, aleatoriamente interrompe a execução de alguma parte do sistema, desliga máquinas ou causa lentidão na rede. Fazendo a medição constante do sistema, eles conseguem avaliar o impacto de cada problema criado e detectar possíveis problemas em seus serviços. Segundo o artigo, esse tipo de prática tem aumentado a tolerância às falhas dos serviços do sistema. Outro ponto interessante que o artigo de Basiri traz é a ideia de que em um sistema distribuído complexo, os engenheiros de software, ao invés de avaliar se o sistema desenvolvido funciona exatamente como o especificado,

passam a avaliar outras métricas do sistema e verificar se o sistema se comporta de forma satisfatória mesmo em caso de falhas em partes do sistema.

Em contrapartida, o nosso trabalho tenta criar cenários determinísticos para que todas as falhas sejam reproduzíveis de forma consistente. Não estamos interessados em falhas aleatórias e sim em permitir a reprodução de testes idênticos, de maneira determinística, controlando a ordem de eventos da aplicação e de falhas em seus componentes. Queremos que o desenvolvedor consiga gerar um script de teste determinando a ordenação dos eventos para validar um determinado requisito do sistema ou de um serviço.

Jiang e Hassan apresentam um *survey* sobre testes de carga em sistemas distribuídos complexos (JIANG; HASSAN, 2015). Nele os autores caracterizam três momentos no desenvolvimento de testes. Primeiro definem a etapa de design e geração da massa de dados que serão utilizados nos testes. Nesta etapa o desenvolvedor deve definir e gerar os dados que serão utilizados nos testes para que fiquem o mais próximo de uma execução real do sistema. Em seguida falam sobre a fase de execução dos testes, em que os desenvolvedores geram os testes, os executam e coletam os dados de resultado. Por fim, na fase de análise do resultado dos testes, os desenvolvedores usam os dados coletados no passo anterior para verificar se o resultado está satisfatório ou não.

Apesar de Jiang e Hassan terem colocado o foco em testes de de carga, o trabalho tem muitas questões que servem para os testes de tolerância a falhas neste tipo de sistema. Das três fases descritas, gostaríamos de focar mais na etapa de execução dos testes de sistemas distribuídos. Estamos especialmente interessados no que os autores chamam de *Programable configuration* em que o *framework* de testes permite que, além da configuração dos parâmetros de um teste, o desenvolvedor possa inserir sua própria lógica que vai executar durante o teste.

Nosso objetivo é permitir que o desenvolvedor programe os testes podendo definir a ordem da a ocorrência de determinados eventos durante a execução do teste. Com isso, o desenvolvedor pode criar cenários de testes de forma reprodutível tornando a execução dos testes determinística.

H. Cui e J. Chen (CUI; CHEN, 2005) discutem esse problema de não determinismo de testes em sistemas distribuídos e propõem uma solução que usa um *middleware* para interceptar a comunicação no sistema em teste e controlar o fluxo de mensagens, gerando testes com comportamento determinístico. Essa abordagem seria bastante intrusiva nos ambientes que temos em mente mas tem a mesma preocupação de nosso trabalho de garantir a repetibilidade dos testes.

2.4 Taxonomia de falhas

Na literatura temos diversas classificações de falhas em sistemas distribuídos. Por exemplo, Steen e Tanenbaum (2017b) cita as seguintes: falha de parada, falha de omissão, falha temporal, falha na resposta e falha arbitrária. Simplificadamente, em um sistema distribuído assíncrono, podemos falar em falhas de parada e falhas arbitrárias.

As falhas de parada ocorrem quando um microsserviço está operando de forma correta e subitamente para de funcionar completamente. Enquanto ele está nesse estado nenhuma interação é possível. Nesse tipo de falha o restante do sistema consegue perceber que o microsserviço está com problema. Esse é o tipo de falha em que estamos interessados.

Já nas falhas arbitrárias, também conhecidas como falhas bizantinas (LAMPORT; SHOSTAK; PEASE, 2019), o serviço em falha passa a se comportar de forma incorreta mas isso pode ou não ser percebido pelos outros serviços do sistema e as informações retornadas por ele podem ser tratadas como válidas ainda que estejam incorretas. Nesse trabalho não vamos tratar esse tipo falha.

No nosso trabalho, não iremos tratar falhas ocorridas dentro do sistema de mensagens. Falhas nesse sistema podem gerar consequências para todos os serviços que o utilizam para comunicação. Mayer et al. (2011) fala sobre a robustez de sistemas *Publish/Subscribe* mostrando os tipos de falhas mais comuns e as medidas que podem ser adotadas para contorná-las. Com a arquitetura descrita por Kreps et al. (2011), o *Kafka* possui seus próprios mecanismos de tolerância a falhas e por isso, não trataremos de falhas no serviço de fila de mensagens. Vamos assumir o comportamento conforme especificado em sua documentação.

O ambiente em que as nossas aplicações de interesse estão inseridas utilizam um sistema de gerenciamento de *containers*, o *Kubernetes*, para fazer a implantação dos microsserviços. Esse tipo de sistema permite que as falhas de paradas sejam identificadas e que o microsserviço que apresente falha seja substituído por outro. Em geral, para determinar se um microsserviço está funcionando corretamente, os gerenciadores de *containers* usam o conceito de *Health Check*.

Health Check é um mecanismo que permite aos desenvolvedores de um microsserviço criar uma lógica que indica se ele está funcionando de forma correta ou não. Isso permite que o gerenciador de *containers* determine que um serviço, apesar de estar rodando, não está mais preenchendo os critérios de bom funcionamento definidos pelos desenvolvedores. Nesse momento, o

desenvolvedor pode definir uma ação a ser executada nesse caso, como por exemplo, disparar uma nova instância para substituir a defeituosa. A lógica que define o bom funcionamento do serviço fica a cargo dos desenvolvedores e, em geral, essa definição é feita através de um *endpoint HTTP* exposto pelo serviço. Com o *health check* definido, o gerenciador fica periodicamente checando a saúde dos serviços. Se nada for definido pelo desenvolvedor, o serviço só é identificado como em falha se sua execução for interrompida.

Para aplicações que não armazenam estado, um gerenciador de *containers* com *health checks* e uma política de substituição dos *containers* defeituosos já é o suficiente para garantir o bom funcionamento no caso de falhas. No entanto, em serviços que guardam estado em geral é necessário garantir que a nova instância seja capaz de recuperar o estado da que falhou para que não ocorra perda de dados em caso de falha.

3
Uma ferramenta de testes para aplicações distribuídas

Nesse capítulo, na seção 3.1, falaremos sobre o uso de bibliotecas e *frameworks* para ajudar no desenvolvimento dos testes. Por fim, na 3.2, introduziremos a biblioteca proposta para auxiliar em testes de sistemas distribuídos.

3.1
Ferramentas de testes

Como observamos, os testes de sistemas distribuídos apresentam um desafio para os desenvolvedores. Com isso, ferramentas e bibliotecas de apoio a testes se tornam extremamente úteis para reproduzir, em testes automatizados, os mesmos cenários complexos desses sistemas.

Dependendo da ferramenta ou biblioteca, o desenvolvedor pode usar uma linguagem específica para testes, que chamamos de linguagem de domínio específico, ou uma linguagem de propósito geral. A escolha da linguagem traz benefícios e desvantagens que devem ser analisadas na hora de escolher que tipo de ferramentas e bibliotecas serão usadas.

Liu e Richardson (LIU; RICHARDSON, 1999) debatem os problemas de utilizar as linguagens de propósito geral para programar testes. O autor discute alguns pontos como a falta de clareza nos testes quando usamos esse tipo de linguagem e a dificuldade na manutenção dos testes.

De acordo com Mernik (MERNIK; HEERING; SLOANE, 2005), linguagens de domínio específico, quando usadas no domínio para qual foram projetadas, apresentam maior expressividade e facilidade de uso quando comparadas com linguagens de propósito geral. O autor fala também na maior clareza e facilidade de manutenção dos testes que utilizam a linguagem de domínio.

Schieferdecker, Din e Apostolidis (2005) e Mladenov, Winsen e Mavrakis (2017) descrevem o uso da linhagem de domínio específico TTCN-3 (WILLCOCK et al., 2011), muito usada em testes de sistemas de telecomunicação. Mladenov mostra também o uso de um *framework* de testes chamado Eclipse Titan[1], que utiliza a TTCN-3 para o teste de sistemas de mensageria.

Apesar da maior expressividade, as linguagens de domínio específico exigem que os programadores aprendam uma nova linguagem para os testes. Em contra partida, as linguagens de uso geral tem a vantagem de ter um maior número de desenvolvedores familiarizados, exigindo menos esforço de aprendi-

[1] <https://projects.eclipse.org/projects/tools.titan>

zado da equipe. Muitas vezes também já estão disponíveis, para linguagens de desenvolvimento geral, bibliotecas e *frameworks* que podem facilitar a geração de script testes.

Uma solução intermediária é o uso de uma linguagem de propósito geral com APIs voltadas para testes. Um exemplo dessa abordagem é a do *framework* K6[2] desenvolvido para testes de carga. Nesse *framework*, a linguagem utilizada para gerar os scripts de testes é a linguagem de propósito geral Javascript. O *framework* provê interfaces para que o desenvolvedor consiga executar testes distribuídos de forma simplificada, podendo por exemplo, rodar diversos testes concorrentes com pouco esforço.

A escolha da linguagem e de *frameworks* também impõe um estilo de programação de testes de acordo com o modelo de programação que é adotado por eles. Com isso, acaba sendo importante avaliar que tipo de testes pretende-se realizar antes de definir a linguagem ou os *frameworks*.

3.2 A nossa Biblioteca de apoio a testes

Neste trabalho investigamos a criação e o uso de uma biblioteca para a criação de testes para aplicações baseadas em microsserviços e que ofereça aos desenvolvedores um modelo de programação voltado para o controle da ordem de eventos. Nosso objetivo é facilitar a criação de cenários de testes em que o desenvolvedor consiga controlar a ordem de eventos no sistema durante a execução do script de teste.

Como a nossa biblioteca trabalha bastante com questões de assincronismo e sincronização decidimos trabalhar com a linguagem Go. Um primeiro ponto que motivou a escolha foram as *goroutines* que são *threads* controladas pelo *runtime* de Go e são criadas de forma simples. Elas são gerenciadas automaticamente pela linguagem, o que significa que não precisamos nos preocupar com a alocação de memória ou gerenciamento de *threads*. Dessa forma, fica mais fácil criar novas *goroutines* sempre que precisarmos executar uma tarefa assíncrona.

Os canais em Go são outra ferramenta interessante para a programação assíncrona. Eles permitem que você envie e receba valores entre *goroutines*, sem precisar se preocupar com a sincronização de *threads* ou bloqueios manuais. Os canais em Go são tipados, e permitem que o tipo de dado enviado no canal seja definido pelo desenvolvedor.

Além disso, Go possui recursos para controlar a concorrência, como *WaitGroup* e *Select*. *WaitGroup* permite esperar que um grupo de *goroutines*

[2] <https://k6.io/docs/>

termine de executar antes de continuar o código principal, enquanto o *Select* permite aguardar vários canais diferentes ao mesmo tempo, facilitando a sincronização entre *goroutines*.

Por causa dessas facilidades da linguagem para trabalhar com código assíncrono, achamos que seria interessante avaliar como o uso dessa linguagem pode facilitar o desenvolvimento da nossa biblioteca.

3.2.1 Modelo de programação

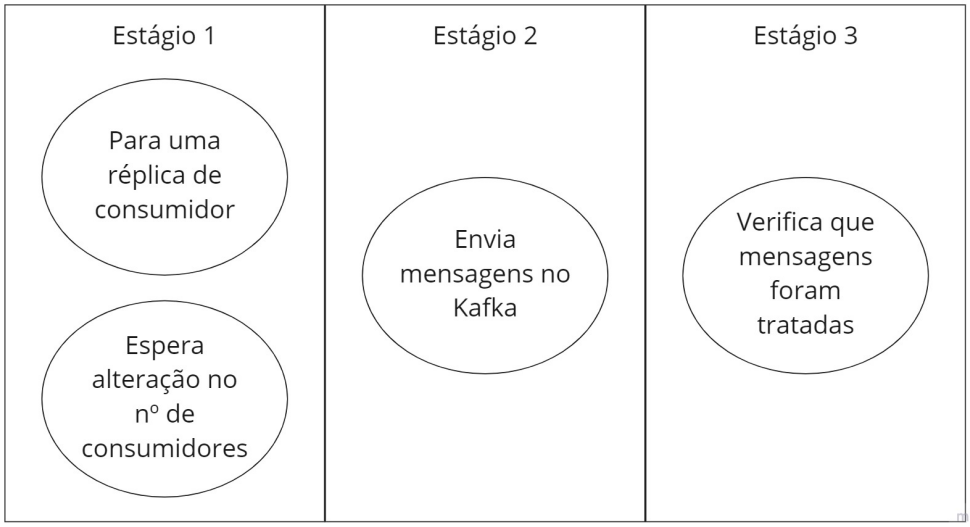
Um conceito primordial na nossa biblioteca é o do *estágio*, para a nossa biblioteca cada estágio representa uma etapa de um script de teste. Cada uma dessas etapas é executada de forma sequencial e em nenhum momento da execução do script mais de um estágio pode estar sendo executado, ou seja, cada estágio executa de forma síncrona. Na figura 3.1, que discutiremos um pouco mais a frente, para que o 'Estágio 2' seja iniciado os eventos definidos no 'Estágio 1' devem ter terminado.

Cada estágio está associado um conjunto de *Jobs*, ou comandos, disparados em *goroutines* no início do estágio. Por padrão, cada um aguarda todos os *Jobs* vinculados a ele terminarem para que o próximo estágio seja iniciado. No entanto, é possível que o desenvolvedor opte por um *Job* que rode durante mais de um estágio, e para isso, fizemos com que seja possível definir um estágio de início e um de término para o *Job*. Nesse caso, o estágio em que o *Job* iniciou não aguarda o término dele para passar a execução para o próximo estágio. O estágio que é definido como terminal vai ficar responsável por finalizar esse *Job*. A API da biblioteca provê funções para adicionar cada um desses tipos de *Jobs*.

3.2.2 Implementação

A criação de um *Job* é feita através da nossa API, utilizando um método do estágio. A chamada desse método vai definir a função que deverá ser executado no *Job* e vinculá-lo ao estágio. Como em Go funções são consideradas de primeira classe, a API recebe como parâmetro a função que será executada pela biblioteca quando o *Job* for iniciado. Para que seja possível determinar o fim de um estágio, cada *Job* deve chamar a função 'Done' quando tiver terminado. O estágio aguarda o final de todos os *Jobs*. Dessa forma, conseguimos garantir que o próximo estágio só vai iniciar depois que todos os *Jobs* estejam indicados como finalizados pelo desenvolvedor.

Figura 3.1: Estágios na biblioteca



Usando a nossa biblioteca conseguimos de forma facilitada criar diversos cenários, como por exemplo o envio de uma mensagem ao *Kafka* logo após a execução de uma réplica de um serviço consumidor ser parada ou mais réplicas entrarem e execução. O desenvolvedor pode criar um teste como o exemplificado na figura 3.1, onde em um primeiro estágio temos dois *Jobs*. No primeiro *Job* o serviço consumidor é parado através do *Docker*. O segundo *Job* do mesmo estágio termina quando o número de consumidores do tópico que o serviço estava cadastrado diminui. No estágio seguinte o *Job* envia uma mensagem para o *Kafka* e em um último estágio um *Job* verifica o funcionamento correto do sistema

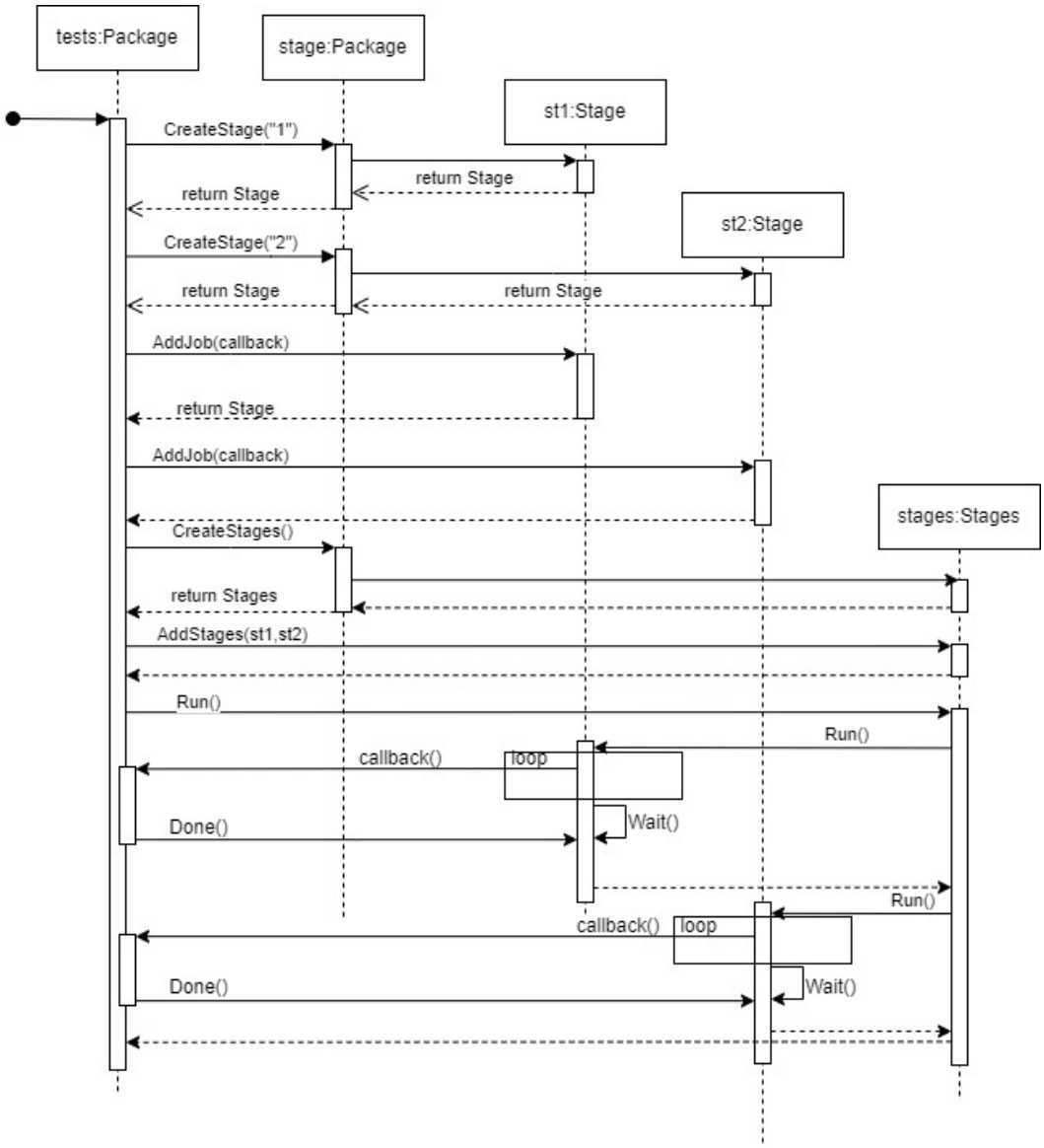
Fizemos um diagrama de sequência simplificado, figura 3.2, que mostra a criação e a execução de estágios durante um script de teste. No diagrama omitimos a abstração do *Job*. Em um diagrama completo, teríamos uma indireção de estágio para *Job* na hora da chamada 'AddJob()' e na chamada 'Run()' do estágio. Ao invés de chamar a *callback* diretamente, o estágio usaria o método 'Run()' do *Job*. O código com os detalhes de implementação estão no apêndice 10.

Código 1: Exemplo de criação de estágios

```
1 stages := stage.CreateStages()
2 st := stage.CreateStage("Primeiro")
3 st2 := stage.CreateStage("Segundo")
4 st3 := stage.CreateStage("Terceiro")
```

Além do diagrama, fizemos um exemplo simplificado de script usando a nossa biblioteca. O exemplo é meramente ilustrativo mas mostra algumas características da biblioteca. Nele criamos três estágios, sendo os dois primeiros com um *Job* e último com dois *Jobs*. Inicialmente, no código 1, fazemos a

Figura 3.2: Diagrama de sequência criação e execução de estágios



criação dos estágios. Guardamos eles em variáveis para vincular os *Jobs* à eles. Com os estágios criados, passamos a vincular os *Jobs* a cada um deles.

Já no trecho de código 2, temos exemplos de definição dos *Jobs* dos estágios 1 e 2. No *Job* do primeiro estágio estamos executando um *container* que a cada segundo escreve em sua saída padrão a string 'oi'; esse *Job*, e por consequência o primeiro estágio, termina quando o *container* é iniciado. Aqui já podemos observar duas características da biblioteca. Em primeiro lugar vemos que no início do *Job* fazemos explicitamente o 'Done' e é dessa maneira que a biblioteca fica sabendo que ele terminou. Repare que aqui usamos o *defer* que coloca a chamada do método para o final da função. Isso é obrigatório para todos os *Jobs* que não são multiestágio. Mostraremos como tratar os *Jobs* multiestágio no capítulo 4. Outra característica é que permitimos que a função

receba um parâmetro enviado pelo desenvolvedor na hora de criação do *Job*, esse parâmetro é de tipo genérico e é necessário fazer o *typecast* para o tipo correto na hora do uso. Esse parâmetro é útil para passar cópia de variáveis e evitar possíveis problemas de concorrência. No estágio seguinte o *Job* utiliza uma função da biblioteca para ficar aguardando que a string 'oi' apareça por três vezes no log do *container* ou ocorra o *timeout* de cinco segundos.

Código 2: Jobs estágios 1 e 2

```
const comando = "while sleep 1; do echo 'oi'; done"
st.AddJob(func(dcag stage.DoneCancelArgGet) {
    defer dcag.Done()
    arg := dcag.GetFuncArg()
    cmd, _ := arg.(string) //typecast
    container, _ = testcontainers.GenericContainer(context.Background(),
        testcontainers.GenericContainerRequest{
            ContainerRequest: testcontainers.ContainerRequest{
                Image: "ubuntu",
                Cmd:   []string{"sh", "-c", cmd},
            },
            Started: true,
        })
}, comando)

st2.AddJob(func(dcag stage.DoneCancelArgGet) {
    defer dcag.Done()
    err := dockertest.WaitForLogMessage("oi", 3, time.Second*5, container)
    if err != nil {
        t.Fatal(err) // timeout
    }
}, nil)
```

Já nos códigos 3 e 4, temos a definição dos Jobs do estágio 3. Nesse estágio, temos exemplos de uso de rotinas e canais em Go. Inicialmente, código 3 fizemos um servidor HTTP que responde às chamadas após três segundos que é usado pelo primeiro *Job* do estágio. Nesse *Job* mostramos um exemplo de criação de uma rotina para fazer uma chamada HTTP de forma assíncrona e do uso de um canal para aguardar o término da rotina. Essa forma de uso do canal é comum para evitar problemas de sincronização entre código síncrono e assíncrono. Nesse caso a função principal do *Job* fica aguardando uma mensagem chegar no canal e só retorna e chama o 'Done' após receber a resposta da chamada HTTP. Já no segundo *Job* do terceiro estágio, código 4, temos outro exemplo de uso de canais. Dessa vez, ao invés de bloquear por completo aguardando o canal como feito no anterior, usamos a primitiva *select* que bloqueia até que qualquer canal dos *cases* seja liberado. Nesse *Job* usamos o mesmo canal do anterior e um canal que é liberado a cada 500ms dentro de um laço infinito. Como o *select* suspende a execução até que algum canal seja liberado, cada iteração do laço só executa quando um canal é liberado.

Código 3: Job com mock de servidor HTTP

```
mockServer := httptest.NewServer(http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
    t.Log("mockServer got request")
    time.Sleep(time.Second * 3)
    w.Write([]byte("ok"))
}))

respChan := make(chan interface{})
st3.AddJob(func(dcag stage.DoneCancelArgGet) {
    defer dcag.Done()
    t.Log("Job 3")
    go func() {
        r, _ := http.Get(mockServer.URL)
        body, _ := io.ReadAll(r.Body)
```

```
14          t.Log("Job 3 Got response: ", string(body))
15          close(respChan) // fecha e todos ouvindo recebem nil
16      }()
17      <-respChan
18  }, nil)
```

Código 4: Job com sincronização via canal

```
1  st3.AddJob(func(dcag stage.DoneCancelArgGet) {
2      defer dcag.Done()
3      t.Log("Job 4")
4      ticker := time.NewTicker(time.Millisecond * 500)
5      defer ticker.Stop()
6      for {
7          select {
8          case <-respChan:
9              t.Log("Job4 end")
10             return
11         case tick := <-ticker.C: // ocorre varias vezes até o caso de cima ocorrer
12             t.Log("Job4 tick", tick.Second())
13         }
14     }
15 }, nil)
```

No código 5 mostramos como adicionar os estágio à uma coleção e fazer a execução de todos os estágios. É só a partir desse momento que a execução dos estágios começa. As etapas anteriores foram apenas de definição. O código completo do exemplor está no código 14 do apêndice.

Código 5: Exemplo de execução de estágios

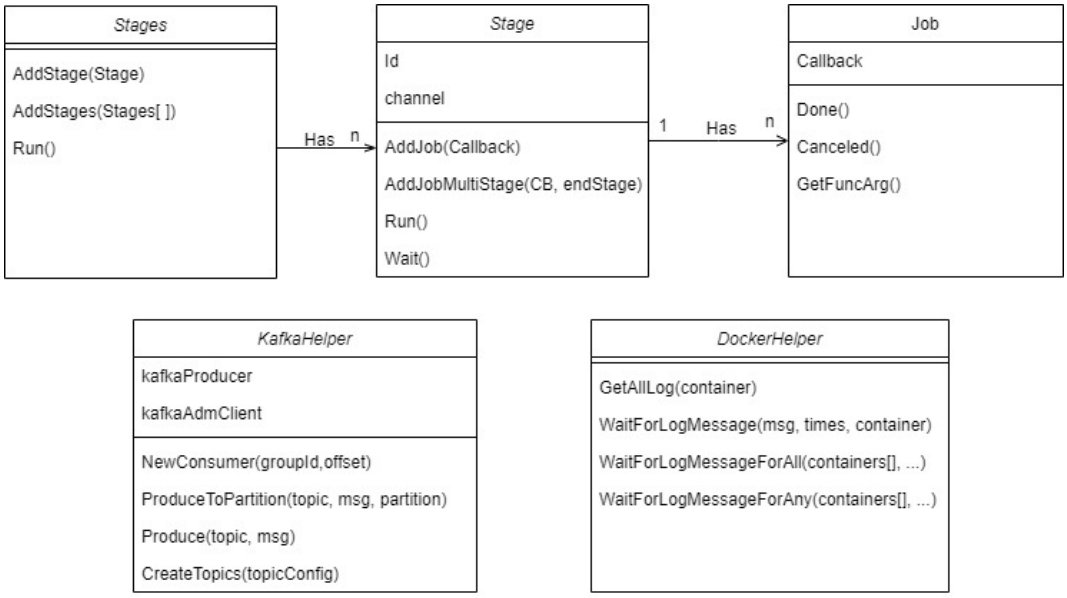
```
1  stages.AddStages([]*stage.Stage{st, st2, st3})
2  stages.Run()
```

Nossa biblioteca inclui APIs para acesso ao *Kafka* e ao *Docker*. Para o uso do *Kafka* nós utilizamos a biblioteca oficial do *Apache* em Go e criamos métodos que simplificam a interação do desenvolvedor com a biblioteca do *Kafka*. Tentamos simplificar, por exemplo, a criação de produtores, consumidores, tópicos e mensagens. Já para o *Docker*, nós utilizamos uma outra biblioteca chamada *Testcontainers*[3] que facilita bastante a criação e execução de *container*. Fizemos então, alguns métodos auxiliares como, por exemplo, uma função que aguarda uma determinada string aparecer no log de um *container*. Podemos ver uma versão simplificada das APIs da biblioteca no modelo descrito na figura 3.3.

Apesar de focarmos no uso da biblioteca para a criação de scripts de testes de sistemas distribuídos que utilizam o *Docker* e o *Kafka*, ela não se limita a esse uso. De forma geral, ela poderia se aplicar a qualquer cenário em que desejamos usar código concorrente com pontos de sincronização, sendo durante testes ou não. Estamos restringindo o estudo para esse tipo de sistema porque será o ambiente em que vamos validar o uso da biblioteca.

[3] <https://golang.testcontainers.org/>

Figura 3.3: Modelo simplificado das APIs



4
Experimentos

Para avaliar o uso de nossa biblioteca, optamos por realizar testes para dois serviços que fazem parte de aplicações distintas em desenvolvimento no Tecgraf e usam a arquitetura de microsserviços. Com esse experimento, queremos verificar em que medida a nossa biblioteca pode ajudar na ordenação de eventos nos testes em sistemas complexos. Vamos avaliar se conseguimos implementar testes reproduzíveis para os cenários de testes não triviais que definimos. Neste capítulo vamos descrever brevemente o contexto de cada um desses serviços e seu funcionamento e apresentar os cenários de testes desenvolvidos explorando o uso da biblioteca. Por fim, vamos fazer uma avaliação sobre a biblioteca.

4.1
Monitor de Execução Remota no SOMA

Desenvolvemos o monitor de execução para o SOMA na disciplina de Projeto Final de Software. O SOMA - evolução do *framework* CSBASE desenvolvido pelo Tecgraf PUC-Rio (LIMA et al., 2015) - é uma plataforma para execução remota de programas com uma arquitetura de microsserviços que usam o *Kafka* na comunicação assíncrona entre eles.

No SOMA, o usuário pode disparar execuções de programas em máquinas de um *cluster* e acompanhar -ou *monitorar*- essas execuções através dos arquivos de log em tempo real. O monitor que desenvolvemos utiliza o *Kafka* para implementar o monitoramento de arquivos, a comunicação com os outros microsserviços do sistema e a coordenação entre réplicas.

O monitor fica conectado ao *Kafka* aguardando mensagens informando qual arquivo deve ser monitorado. A partir de uma mensagem requisitando uma monitoração, o monitor passa a observar o arquivo informado e envia todo o conteúdo do arquivo para um tópico específico para esse arquivo. A cada escrita no arquivo de log, o conteúdo adicionado é enviado. O monitor fica observando o arquivo até receber uma mensagem cancelando a observação. Como a monitoração ocorre em arquivos de log, o monitor só está preparado para tratar arquivos que sofrem edição do tipo *append*, ou seja, toda edição somente adiciona conteúdo ao arquivo.

Durante seu funcionamento normal, o serviço se conecta como consumidor no tópico 'monitor_interesse' para receber requisições de monitoramento. Neste tópico ele recebe mensagens em formato *JSON*, contendo uma *string*

com o caminho do arquivo (Path) e um booleano que indica se o arquivo deve ser monitorado ou não (início ou cancelamento do monitoramento).

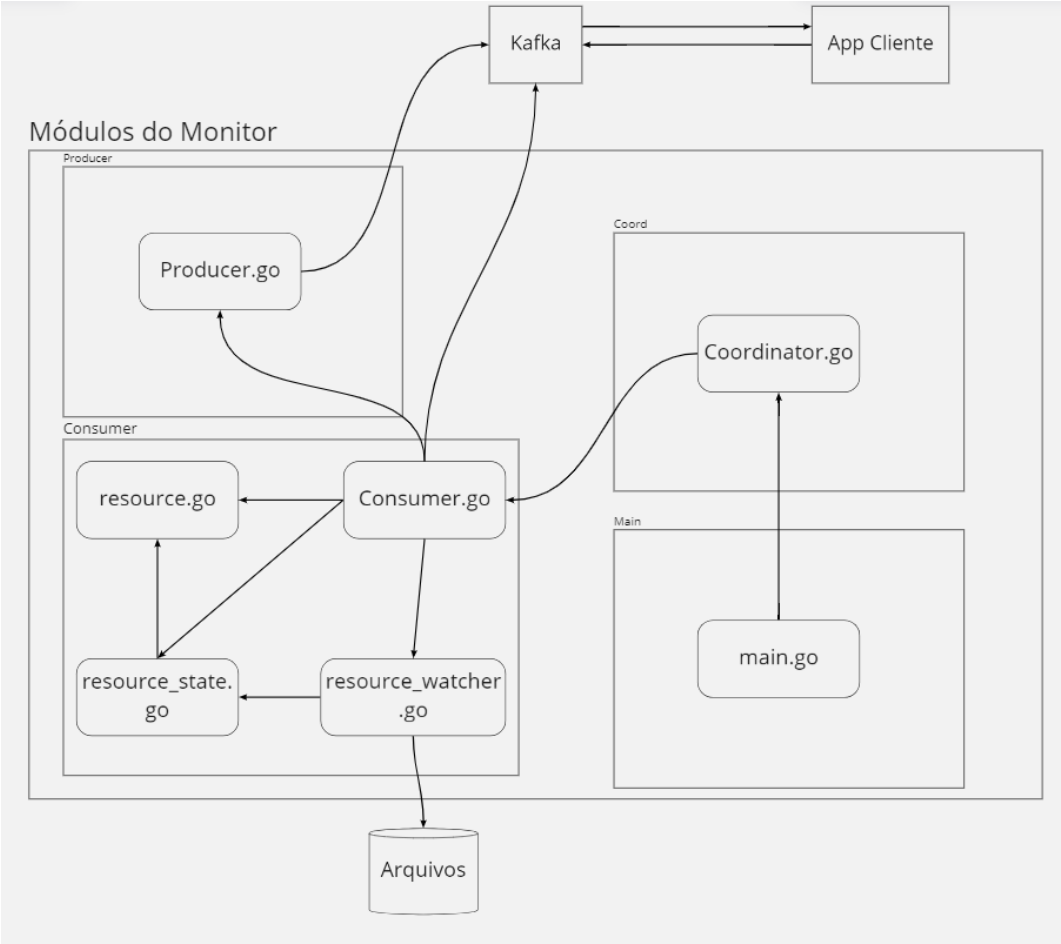
O monitor foi construído para poder ser replicado, desacoplado dos consumidores da monitoração e tolerante a falhas do tipo *fail-stop* em uma ou mais de suas réplicas. Fizemos uso do conceito de grupo de consumidores do *Kafka* para que a coordenação das réplicas ficasse a cargo do Kafka. Uma característica do serviço de monitoração é que ele precisa manter estado (arquivos monitorados e última posição lida em cada arquivo) e em caso de falha é necessário que a réplica que assumir o trabalho consiga recuperar esse estado. Assim, criamos um tópico especifico no *Kafka* que guarda o estado dos monitores, que pode ser recuperado por outras réplicas durante a execução.

Os módulos implementados estão indicados na figura 4.1. O módulo principal do monitor é o 'Consumer' que se conecta ao *Kafka* como consumidor e recebe mensagens para iniciar ou parar o monitoramento de arquivos. Além de mensagens no *Kafka*, o 'Consumer' também reage a mudanças nos arquivos monitorados. Nesse caso o 'resource_watcher' informa a mudança e o conteúdo que foi adicionado para o 'Consumer' que repassa para o 'Producer'. O 'Producer' fica responsável por produzir as mensagens com o conteúdo recebido. Temos um exemplo, na figura 4.2, da sequência de acontecimentos quando uma réplica do monitor recebe uma mensagem do *Kafka* para monitorar um novo arquivo.

Para guardar o estado, aproveitamos o fato de que, dentro de um grupo de consumidores, uma partição é designada a no máximo um membro do grupo e passamos a guardar o estado de cada partição do tópico de entrada do monitor em um tópico com o mesmo número de partições que o de entrada. Cada réplica posta a alteração de estado de cada partição (alteração em arquivos monitorados ou na última posição lida de um arquivo) na partição correspondente do tópico de estado. Assim, quando o monitor responsável pela partição 0 do tópico de entrada recebe uma mensagem para monitorar um determinado arquivo, ele inicia a monitoração e posta no tópico de estado, na partição 0, o novo estado da partição. Dessa forma, a última mensagem enviada em cada partição do tópico de estado contém os arquivos e a posição de leitura de cada arquivo que monitor responsável pela mesma partição no tópico de entrada está monitorando. Sempre que algum monitor tiver uma falha, o *Kafka* redistribui as partições entre as réplicas. Cada réplica saudável restante recebe a lista de novas partições pelas quais ficou responsável. A réplica pode então consultar o tópico de estado, nas partições correspondentes, para recuperar a última mensagem recebida para cada uma dessas partições.

Durante o desenvolvimento inicial desse serviço foi possível observar a

Figura 4.1: Módulos do monitor



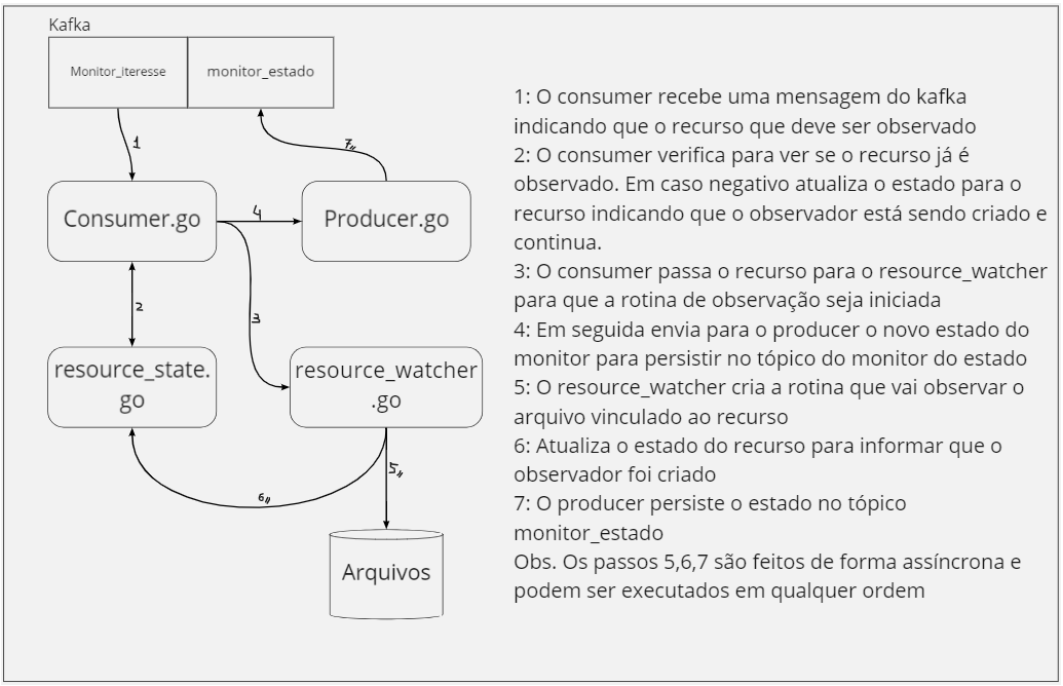
complexidade de gerar testes automatizados para aplicações com esse tipo de arquitetura. Vimos que o uso de uma biblioteca que pudesse nos ajudar a ordenar os eventos, auxiliar na comunicação com outros serviços e gerenciar os *containers* poderia ser de grande valia para o desenvolvedor desse tipo de aplicação.

4.2 Definição dos cenários de testes do monitor

Nossa motivação para o uso dessa aplicação para avaliar o uso da biblioteca foi por ela guardar estado dos arquivos que está monitorando e em caso de falha de uma réplica, pode não ser trivial que as outras continuem o trabalho de forma correta, sem perda ou duplicação de dados. Além disso, esse serviço usa uma abordagem de delegar a coordenação das réplicas para o *Kafka* através do uso de grupo de consumidores. Isso cria uma dependência que pode tornar o teste e a ordenação de eventos mais complexos.

Inicialmente criamos um teste para verificar o funcionamento correto do serviço. Aqui buscamos testar um requisito do serviço, ao iniciar a monitoração

Figura 4.2: Observando um novo recurso



de um arquivo que já possuí conteudo o serviço deve enviar esse conteúdo para o*Kafka*. O monitor vai receber uma mensagem pedindo a monitoração de um arquivo e deve enviar mensagens ao *Kafka* com o conteúdo desse arquivo. Ao desenvolvermos esse cenário encontramos um erro no monitor. Quando o serviço começa a observar um arquivo pela primeira vez, ele só começa a enviar o conteúdo do arquivo após ocorrer uma mudança no arquivo. Assim, se todo o conteúdo do arquivo for escrito antes da monitoração ser iniciada, o serviço não vai enviar nada para o *kafka*.

O código 6 mostra um cenário simplificado que reproduz o comportamento do erro encontrado. Inicialmente criamos os estágios e definimos o *container*. Na linha 12 informamos que queremos aguardar por uma ocorrência de *string* no log ou por dez segundos. O primeiro estágio tem apenas o *Job* da linha 37, que inicia o *container* e só termina quando o log for encontrado ou o tempo tiver passado. No segundo estágio temos dois *Jobs*: o primeiro, na linha 49, envia uma mensagem ao *Kafka* pedindo para que um arquivo que já tem uma linha escrita seja monitorado e aguarda a confirmação da entrega. O segundo *Job*, na linha 56, aguarda que uma entrada com o nome do arquivo que deve ser observado seja escrita no log do *container*. Por fim, no último estágio, o *Job* da linha 66 aguarda por uma mensagem no tópico em que o monitor deve enviar o conteúdo do arquivo e se assegura que a mensagem não é vazia. Com o erro que detectamos, esse *Job* vai falhar na linha 71 porque o monitor nunca vai enviar a mensagem.

Código 6: Teste monitor

```
// Reproduz bug do monitor que não envia msg até que tenha ocorrido mudança no arquivo
func TestFileWithInitialContent(t *testing.T) {
  var containerMonitor testcontainers.Container
  stages := stage.CreateStages()
  st := stage.CreateStage("inicia_container")
  st2 := stage.CreateStage("envia msg e aguarda o monitor receber")
  st3 := stage.CreateStage("aguarda msg no topico do arquivo")

  const monitorReadyLogMsg = "TIME SPENT WAITING FOR STATE RECOVERY"
  r := testcontainers.ContainerRequest{
    Image:      "monitor:0.0.1-snapshot",
    WaitingFor: wait.ForLog(monitorReadyLogMsg).WithOccurrence(1).WithStartupTimeout(time.Second *
        10),
    Mounts:     monitorMounts,
    HostConfigModifier: func(hc *container.HostConfig) {
      hc.NetworkMode = "host"
    },
  }

  // inicializa helper do kafka
  k, err := kafkatest.NewKafka(kafka.ConfigMap{"bootstrap.servers": "localhost:9092",
    "acks": "all"})
  if err != nil {
    t.Fatal(err)
  }
  // Cria os tópicos
  k.CreateTopics(&tConfig)
  // Cria o consumidor e subscreve ao tópico de interesse
  consumer, err := k.NewConsumer("test-group", kafkatest.OffsetEarliest)
  if err != nil {
    t.Fatal(err)
  }
  err = consumer.SubscribeTopics([]string{oneLineFileTopic}, nil)
  if err != nil {
    t.Fatal(err)
  }

  st.AddJob(func(dcag stage.DoneCancelArgGet) {
    defer dcag.Done()
    var err error
    containerMonitor, err = testcontainers.GenericContainer(context.Background(), testcontainers.
        GenericContainerRequest{
      ContainerRequest: r,
      Started:          true,
    })
    if err != nil {
      t.Fatal(err)
    }
  }, nil)

  st2.AddJob(func(dcag stage.DoneCancelArgGet) {
    defer dcag.Done()
    ch := make(chan kafka.Event)
    k.Produce("monitor_interesse", fmt.Sprintf(`{ "path": "%s", "project": "p1", "watch": true}`,
        oneLineFile), ch)
    <-ch // Aguarda confirmação de entrega
  }, nil)

  st2.AddJob(func(dcag stage.DoneCancelArgGet) {
    defer dcag.Done()
    log.Println("Waiting for the monitor to recieve the msg")
    err := dockertest.WaitForLogMessage(oneLineFileName, 1, time.Second*3, containerMonitor)
    if err != nil {
      t.Fatal(err) // timeout
    }
    log.Println("Monitor got the msg")
  }, nil)

  st3.AddJob(func(dcag stage.DoneCancelArgGet) {
    defer dcag.Done()
    const timeout = time.Second * 5
    m, err := consumer.ReadMessage(timeout)
    if err != nil {
      t.Fatal(err) // timeout
    }
    assert.NotEmpty(t, m)
  }, nil)

  stages.AddStages([]*stage.Stage{st, st2, st3})
  stages.Run()
```

Nesse cenário a nossa biblioteca ajudou para fazer a comunicação com o *Kafka* criando o estado inicial dos tópicos, além de permitir o consumo e envio de mensagens de forma facilitada. Nossa API do *Docker* foi bem útil para verificar que o monitor recebeu uma mensagem do *Kafka* e que contém o arquivo que queremos de fato observar. A utilização dos estágios e *Jobs* foi suficiente para garantir a ordenação dos eventos e reproduzir o problema da escrita no arquivo ocorrer antes do monitor começar a monitoração.

Em um segundo cenário buscamos explorar um exemplo de falha *fail-stop* com uma ordenação de eventos mais complexa em que determinados eventos devem ocorrer durante a ocorrência do rebalanceamento do grupo de consumidores do Kafka. A ideia é verificar que a monitoração de um arquivo

continua mesmo em caso de falha da instância que estava responsável por ele. Para esse teste, pretendíamos executar duas instâncias do monitor, começar a monitoração de um arquivo, simular uma falha do tipo *fail-stop* no monitor que ficar responsável por ele, fazer escritas no arquivo monitorado durante a etapa de redistribuição das partições feita pelo *Kafka* e, em seguida, verificar se o monitor que continuou executando passa a monitorar o arquivo. Nesse cenário era necessário garantir que a sequência de eventos fosse consistente para que a escrita ocorresse durante o momento que o monitor está fazendo a recuperação do estado da partição.

Esse cenário mostrou a limitação da biblioteca quando tentamos fazer uma sincronização com eventos que dependem de uma comunicação direta entre dois serviços nos quais não podemos interferir. Não conseguimos, de forma consistente, reproduzir o cenário de envio de mensagens enquanto o rebalanceamento de partições do tópico está ocorrendo. Conseguimos saber que o protocolo de rebalanceamento havia sido disparado pelo *Kafka*, mas não conseguimos controlar o teste para garantir que a mensagem enviada ao *Kafka* chegasse enquanto o rebalanceamento estava ocorrendo. Mesmo assim, vimos alguns pontos positivos da nossa biblioteca. Por exemplo, o método 'waitForAnyLog' funcionou bem para descobrir qual *container* escreveu em seu no log que passou a monitorar o arquivo que requisitamos e assim sabemos qual réplica devemos interromper para simular um *fail-stop* forçando o rebalanceamento.

4.3 Aplicação +Acordo do TJRJ

Além do serviço descrito anteriormente, buscamos um outro serviço que fizesse parte de uma sistema baseado em microsserviços e que utilizasse o *Kafka* para a comunicação. O serviço escolhido faz parte de um sistema que está sendo desenvolvido pelo Tecgraf em conjunto com outros departamentos e é fruto de uma parceria entre a PUC-Rio, o Tribunal de Justiça do estado do Rio (TJRJ) e a Light. A plataforma que está sendo desenvolvida se chama +Acordo e é uma solução pré-processual para resolução de disputas online. O projeto, inicialmente, está voltado para questões do consumidor e busca resolver os problemas dos usuários de forma ágil e acessível, minimizando custos e burocracia. O sistema combina técnicas de *machine learning* e processamento de linguagem natural para gerar, automaticamente, propostas de acordo com base em dados fornecidos pelas partes.

O sistema utiliza a arquitetura de microsserviços e o serviço que vamos utilizar para os nossos testes é o Complementação Judicial. Esse serviço recebe

uma demanda via mensagem no *Kafka* e usa uma API do TJRJ para buscar informações sobre processos anteriores entre o demandante e a demandada. Após a complementação, envia a demanda com as novas informações em um tópico de saída do serviço para que ela continue sendo processada pelo sistema. A comunicação entre o serviço e o TJRJ é feita através de uma API HTTP fornecida pelo tribunal. O serviço de complementação fica conectado ao *Kafka* ouvindo o tópico 'demanda-submetida'. Sempre que uma mensagem é recebida nesse tópico o serviço faz a complementação da demanda e a envia no tópico 'resposta-complementacao-judicial'.

Para esse microsserviço a política adotada para uma falha *fail-stop* é de disparar uma nova instância e remover a que está em falha. Como as instâncias não guardam estado, não é necessária nenhuma lógica para a recuperação do trabalho da que falhou. Nesse ambiente o *Kubernetes* é usado para gerenciar os *containers* dos microsserviços e cada um deles provê um *endpoint* de *health check* para informar a saúde do serviço para que o *Kubernetes* possa identificar o bom funcionamento de cada parte do sistema.

4.4 Cenários de teste do serviço Complementação Judicial

A motivação para o uso dessa aplicação foi o uso de comunicação sincrona e assíncrona durante seu funcionamento. Esse serviço recebe e envia mensagens assíncronas para outros serviços do sistema e consulta um serviço do sistema do TJRJ de forma síncrona. Além disso, ele tem mais depêndencias que o Monitor para seu funcionamento.

O primeiro cenário que decidimos testar foi o de uma falha bizantina em que o serviço recebe mensagens fora do padrão esperado no tópico de entrada. Com esse teste detectamos um erro na aplicação quando uma mensagem recebida no tópico 'demanda-submetida' não estava no modelo esperado, o serviço abortava a execução. Para reproduzir esse cenário subimos o serviço usando o *docker*, enviamos uma mensagem que não segue o padrão esperado, verificamos que o *health check* do serviço deve permanecer como *healthy*, verificamos que o tópico 'demanda-falhada' recebeu uma mensagem e, por fim, verificamos que a mensagem do *Kafka* foi consumida (o *offset* do tópico foi alterado).

O *script* de teste é mostrado de forma simplificada no código 7. Nesse trecho retiramos a maioria das verificações de erro para deixar código mais limpo e usamos uma versão simplificada da verificação do recebimento da mensagem no tópico de falhas.

O código do teste mostra alguns detalhes do uso da biblioteca. Inicial-

mente criamos os estágios (linhas 5 a 8), nas linhas 11 a 21 inicializamos o *helper* do *Kafka*, criamos os tópicos com as partições e as mensagens iniciais, que nesse caso estão vazios, e por fim fazemos a subscrição aos tópicos de saída do serviço. Das linhas 39 até a 73 estamos adicionando os *Jobs* que serão executados em cada estágio do teste. O primeiro *Job* (linha 34) fica responsável por subir o serviço sob teste e, no fluxo normal, ele só termina após o serviço responder '200' no *endpoint* de *Health Check*. Em seguida, criamos um *Job* que será executado durante todos os estágios (linha 40), que fica responsável por verificar as mensagens recebidas no *Kafka*. No estágio 2 o *Job* adicionado envia uma mensagem mal formada para o tópico de entrada do serviço e aguarda a confirmação de envio através de um canal. Por fim, no terceiro estágio, criamos um *Job* que verifica até um determinado tempo de *timeout*, se o tópico de 'demanda-falhada' recebeu mensagem, o que indicaria que o comportamento do serviço está correto. Até esse ponto do código nenhum *Job* está em execução. Estamos apenas criando as *callbacks* que serão executadas em cada estágio. A execução só é iniciada após adicionarmos os estágios ao *stages* e chamarmos o *Run()* (linhas 75 e 76). Os estágios são executados na ordem que foram adicionados.

Código 7: Teste simplificado mensagem inválida

```
1  // ...
2
3  func SampleTestEmptyMsgOdj(t *testing.T) {
4    topicMessages := make(map[string][]string)
5    stages := stage.CreateStages()
6    st1 := stage.CreateStage("Inicializa serviço sob teste")
7    st2 := stage.CreateStage("Envia mensagem mal formatada ao kafka")
8    st3 := stage.CreateStage("Verifica se o tópico demanda-falhada recebeu uma mensagem")
9
10   // inicializa helper do kafka
11   k, _ := kafkatest.NewKafka(kafka.ConfigMap{"bootstrap.servers": "localhost:9092",
12     "acks": "all"})
13   // Cria os tópicos vazios
14   k.CreateTopics(&topic.TopicConfig{Topics: []topic.Topics{
15     {Name: "demanda-submetida", NumPartitions: 1, Messages: []topic.Messages{}},
16     {Name: "resposta-complementacao-judicial", NumPartitions: 1, Messages: []topic.Messages{}},
17     {Name: "demanda-falhada", NumPartitions: 1, Messages: []topic.Messages{}},
18   }})
19   // Cria o consumidor e subscreve aos tópiocs de interesse
20   consumer, _ := k.NewConsumer("empty-msg-test-group", kafkatest.OffsetEarliest)
21   _ = consumer.SubscribeTopics([]string{"demanda-submetida", "demanda-falhada"}, nil)
22
23   // Inicia o serviço de complementação e aguarda resposta do health check para continuar, falha em
         caso de timeout
24   st1.AddJob(func(dcag stage.DoneCancelArgGet) {
25     defer dcag.Done()
26     _, err := testcontainers.GenericContainer(context.Background(), testcontainers.
           GenericContainerRequest{
27       ContainerRequest: testcontainers.ContainerRequest{
28         Image:      "repo.tecgraf.puc-rio.br:18089/odrtj/odr-complementacao-tj:master",
29         WaitingFor: wait.ForHTTP("/q/health/live").WithPort("8080/tcp").WithStartupTimeout(time.
               Second * 10),
30         Env:        map[string]string{"ODR_KAFKA_HOST": "localhost:9092"},
31       },
32       Started: true,
33     })
34     if err != nil {
35       t.Fatal(err)
36     }
37   }, nil)
38
39   // Job multi estágio que checa periodicamente os tópicos subscritos
40   st1.AddJobMultiStage(func(dcag stage.DoneCancelArgGet) {
41     ticker := time.NewTicker(time.Millisecond * 100)
42     defer ticker.Stop()
43     for {
44       select {
45       case <-dcag.Canceled(): // cancelado ao final do estagio st3
46         return
47       case <-ticker.C:
48         // faz poll no nos tópicos e caso receba mensagem adiciona ao mapa 'topicMessages'
49         checkMessages(consumer, topicMessages, t)
50       }
51     }
52   }, st3, nil)
53
54   // produz uma mensagem mal formatada para o tópico de entrada do serviço
55   st2.AddJob(func(dcag stage.DoneCancelArgGet) {
56     produceChan := make(chan kafka.Event)
57     _ = k.Produce("demanda-submetida", "{}", produceChan)
```

```
58      // aguarda confirmação da entrega da mensagem, falha se der timeout
59      select {
60      case <-time.After(5 * time.Second):
61        t.Fatal("Não recebeu confirmação de mensagem enviada")
62      case <-produceChan:
63        dcag.Done()
64      }
65
66    }, nil)
67
68    // Job para verificar a cada 500ms se que o tópico demanda-falhada recebeu uma nova mensagem
69    // e que o health check continua retonando 200 (código omitido para simplificar)
70    st3.AddJob(func(dcag stage.DoneCancelArgGet) {
71      // Verifica se recebeu mensagem a cada 500ms com timeout de 5 segundos
72      // Em caso de timeout falha
73    }, nil)
74
75    stages.AddStages([]*stage.Stage{st1, st2, st3})
76    stages.Run()
77  }
```

Esse primeiro cenário foi interessante porque nos permitiu encontrar um problema no serviço de complementação do TJRJ ao tratar uma mensagem em um formato não esperado. Verificamos que ao receber uma mensagem com informações faltando o serviço abortava e não consumia a mensagem que causou o problema. Com isso, toda réplica que subia para substituir a que falhou, também consumia a mensagem errada e acabava falhando. Aqui observamos algumas questões que impactaram na nossa capacidade de sincronizar de forma consistente o teste. O *health check* e o log do serviço não informam se ele está pronto para receber mensagens. O *health check* indica apenas que a aplicação está rodando, mas não aguarda toda a inicialização e o log possui apenas informações de que vai se cadastrar nos tópicos, mas não diz se está responsável por alguma partição. Dessa forma, passamos a ter que usar algumas pausas arbitrárias para garantir que o serviço já estava pronto para receber mensagens.

Um outro cenário é o de indisponibilidade da API do tribunal, que simula uma falha *fail-stop* em uma dependência do servido de complementação. Nesse cenário enviamos uma mensagem bem formada para o tópico de entrada mas não subimos a API do TJ que precisaria ser consultada para o funcionamento normal do serviço. Em seguida, vamos verificar que o tópico 'demanda-falhada' recebeu uma mensagem, o *health check* continua como *healthy* e que a mensagem do tópico de entrada foi consumida. O código desse cenário pode ser encontrado no apêndice (linha 262 do código 16).

Nesse segundo cenário, tivemos que lidar com a complexidade do ambiente. O serviço de complementação depende de um microsserviço de autenticação chamado *keycloak*. Antes de se comunicar com a API do TJRJ ele se autentica usando o *keycloak*, pegando um *token* que será usado para validar o acesso à API do TJRJ. Com isso, foi necessário fazer um *mock* do *keycloak*. Tivemos um problema aqui para verificar que a mensagem foi consumida no tópico, conseguimos apenas verificar no log do serviço testado que ele havia recebido a mensagem. O plano inicial era verificar o próprio *offset* do tópico através do *Kafka*, mas a biblioteca de Go que usamos para a comunicação com o *Kafka* não dá suporte para essa consulta. Mesmo assim, conseguimos fazer

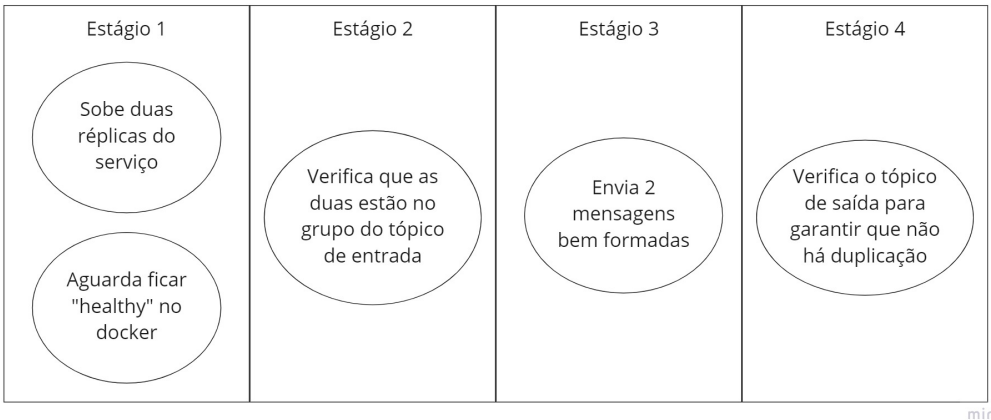
o teste corretamente ao verificar que o serviço enviou uma mensagem para o tópico que é utilizado para informar falhas e permaneceu com o *health check* informando o funcionamento normal.

Aqui também temos um exemplo de uso do *Job* multiestágio que fica consumindo mensagens do *Kafka* e é interrompido quando o estágio final do *Job* termina. Para ser interrompido, o *Job* deve ficar constantemente verificando se recebeu alguma mensagem no canal de cancelamento. Esse é um padrão que é necessário poque não é possível, em Go, interromper as *goroutines* por 'fora' (linha 326 do código16).

Por último fizemos um teste para verificar o comportamento da aplicação quando há replicação. Para isso, vamos subir duas réplicas do serviço, aguardar o *docker* reconhecer que elas estão saudáveis e em seguida verificar que as réplicas estão no mesmo grupo de consumidores do tópico de entrada. Após isso, vamos enviar duas mensagens bem formadas e verificar no tópico de saída que as duas mensagens foram tratadas sem duplicação. A figura 4.3 mostra como é esse cenário no modelo da biblioteca. Este não é um teste de comportamento em caso de falha, mas nos pareceu interessante investigar o uso de nossa biblioteca também para um teste de correção (linha 515 do código 16).

No último cenário tivemos dificuldades, mais uma vez, pela limitação da biblioteca do *Kafka* em Go e não foi possível verificar se as réplicas estavam no mesmo grupo. Para verificar que as replicas estavam trabalhando corretamente, criamos o tópico de entrada com duas partições, escrevemos uma mensagem em cada partição e verificamos através do log que cada réplica recebeu uma mensagem e que ao todo duas mensagens foram enviadas para o tópico de saída do serviço.

Figura 4.3: Modelo para cenário de réplica



4.5 Discussão

Depois de desenvolvermos o a biblioteca e os testes para as aplicações que usamos como exemplo, vamos avaliar o uso da linguagem Go tanto para o desenvolvimento de testes quanto para o desenvolvimento de código assíncrono com sincronizações. Além disso, avaliaremos também os pontos positivos e negativos do uso da nossa biblioteca.

Uso de Go

Sobre o uso da linguagem Go, tivemos uma experiência positiva principalmente na facilidade do uso de *goroutines*. Na nossa biblioteca fizemos com que o código de cada *Job* execute em uma *goroutine* para que eles executassem de forma concorrente, permitindo que um *Job* execute uma ação bloqueante, como aguardar uma mensagem do *Kafka*, enquanto outro *Job* permanece executando de forma concorrente. O estágio desses *Jobs* fica aguardando a finalização de cada um deles para que o próximo estágio seja iniciado. Isso é feito usando bibliotecas padrão de Go para sincronização.

No código 8 mostramos como a nossa biblioteca faz a sincronização utilizando apenas funcionalidades de Go. Cada *Stage* tem um *WaitGroup* (linha 5) e toda vez que é adicionado um *Job* que começa e termina no estágio fazemos um 'Add(1)' no *WaitGroup* (linha 32). Quando o desenvolvedor informa o fim de um *Job* chamando o método do *Job* 'Done' nós fazemos um 'Done' no *WaitGroup* do estágio (linha 16). Quando rodamos o estágio criamos uma *goroutine* (linha 42) que vai fazer a execução de todos os *jobs*.[1] Em seguida aguardamos o final do estágio usando a função 'Wait' do *WaitGroup*. Internamente o *WaitGroup* é um contador seguro para concorrência e a chamada 'Wait' fica presa até que o contador fique menor ou igual a zero.

Código 8: Uso de sincronização de goroutine

```
1  import "sync"
2
3  type Stage struct {
4    Id           string
5    WaitGroup    *sync.WaitGroup
6    ... // omitido para simplificar exemplo
7  }
8
9  type Job struct {
10   Begin   *Stage
11   End     *Stage
12   Work    func(DoneCancelArgGet)
13   ...
14 }
15
16 func (j Job) Done() { // Deve ser chamado pelo desevolvedo para indicar o fim do Job
17   if j.Begin == j.End {
18     j.End.WaitGroup.Done()
19   }
20 }
21
22 func (s *Stage) Wait() {
```

[1] A criação de uma *goroutine* é feita usando a palavra reservada 'go' antes da chamada de uma função.

```
23    s.WaitGroup.Wait()
24  }
25
26  func (st *Stage) AddJobMultiStage(work func(DoneCancelArgGet), endStage *Stage, workArg interface{})
        *Job {
27    job := Job{
28      ...
29    }
30    if st == endStage { // Começa e termina no mesmo estágio
31      st.Jobs = append(st.Jobs, job)
32      endStage.WaitGroup.Add(1)
33    } else {
34      endStage.JobsToFinish = append(endStage.JobsToFinish, job)
35    }
36
37    return &job
38  }
39
40  func (st *Stage) Run() {
41    for _, job := range append(st.Jobs, st.JobsToFinish...) {
42      go st.runWork(job)
43    }
44    st.Wait()
45    ...
46        close(st.channel)
47  }
```

Achamos muito prático criar um código que executa de forma concorrente e conseguimos fazer a criação e sincronização de forma fácil usando apenas bibliotecas da própria linguagem. Outro ponto bem interessante é o uso de canais em conjunto com o 'select' que pode ser usando tanto para comunicação quanto para a sincronização, como vimos anteriormente no código **??**.

Outra vantagem é que Go possui um sistema de gerenciamento de pacotes fácil de usar, o que torna o processo de adicionar e gerenciar bibliotecas externas bem simples. Além disso, Go tem uma comunidade ativa e colaborativa, que produz e mantém bibliotecas para uma ampla gama de necessidades de desenvolvimento. Na nossa biblioteca, por exemplo, usamos a biblioteca oficial da *Apache* para a comunicação com o *Kafka* e a *TestContainers* para lidar com *containers*.

Um ponto que não gostamos no uso de Go foi que o tratamento de erro em Go pode ser um pouco complicado para programadores acostumados com outras linguagens. Em vez de usar exceções, como em algumas outras linguagens, Go utiliza valores de erro para indicar falhas na execução do programa. Quando uma função ou método em Go é chamado, ela geralmente retorna dois valores: um valor de resultado e um valor de erro. Se a função ou método executou com sucesso, o valor de erro é nulo (nil) e o valor de resultado contém o resultado da operação. Se ocorreu um erro, o valor de erro contém uma descrição do erro e o valor de resultado é geralmente nulo. Com isso, o código sempre mistura a lógica com o tratamento de erro como podemos ver no exemplo 9.

Código 9: Tratamento de erro em Go

```
1  k, err := kafkatest.NewKafka(kafka.ConfigMap{"bootstrap.servers": "localhost:9092",
2    "acks": "all"})
3  if err != nil {
4    t.Fatal(err)
5  }
6  k.CreateTopics(&topic.TopicConfig{Topics: []topic.Topics{
7    {Name: entryTopic, NumPartitions: 1, Messages: []topic.Messages{}},
8    {Name: outTopic, NumPartitions: 1, Messages: []topic.Messages{}},
9    {Name: errorTopic, NumPartitions: 1, Messages: []topic.Messages{}},
10 }})
11
12 consumer, err := k.NewConsumer("test-group", kafkatest.OffsetEarliest)
13 if err != nil {
14   t.Fatal(err)
15 }
```

```
16
17 err = consumer.SubscribeTopics([]string{entryTopic, errorTopic}, nil)
18 if err != nil {
19   t.Fatal(err)
20 }
```

Uso da nossa biblioteca

Nossa biblioteca se mostrou útil para ajudar a criar testes que dependem da ordenação de eventos. O uso de estágios para agrupar funções que executam de forma concorrente permite a construção de testes complexos sem que o desenvolvedor tenha que lidar diretamente com a parte de sincronização. A biblioteca permite também a criação de *Jobs* que executam por mais de um estágio. Isso se mostrou útil para criar um teste do monitor que ficava periodicamente escrevendo em um arquivo, e parece ser interessante ter essa possibilidade, principalmente quando lidamos com eventos onde podemos querer ter um *Job* coletando mensagens do *Kafka*, ou queremos ter um *mock* HTTP rodando durante alguns estágios.

Com o uso da biblioteca conseguimos pegar erros nas duas aplicações que testamos e foi possível gerar scripts de teste que reproduzem de forma consistente os casos de erro, permitindo que após a correção do código dos serviços os mesmos testes sejam executados para verificar que o comportamento foi ajustado.

Para o uso do *Kafka*, nossa biblioteca facilitou a criação de produtores e consumidores. Além de permitir que o desenvolvedor crie um estado inicial dos tópicos para o teste, a biblioteca cria os tópicos informados pelo programador e os preenche com as mensagens informadas. O código 9 mostra a inicialização do consumidor e produtor e a criação dos tópicos que serão usados nos testes. Além do método de criação da linha 6, a biblioteca permite que o desenvolvedor informe um arquivo JSON. Conseguimos criar facilmente *Jobs* que produzem e consomem mensagens do *Kafka*. Esses *Jobs* foram úteis para sincronizar a execução do teste em alguns cenários. O uso do *Kafka*, para esses cenários, com a nossa biblioteca ficou mais simples do que usar diretamente a biblioteca oficial do *Kafka*.

Já no uso de *containers* inicialmente implementamos um código que permitia iniciar, parar e obter logs de *containers* usando a biblioteca oficial do *Docker* para Go. Entretanto passamos a usar a biblioteca *Testcontainers* que já faz essa interface com o *Docker* de forma simplificada. Decidimos então, criar funções auxiliares de sincronização usando os *containers* gerados pelo *Testcontainers*. Criamos por exemplo uma função para aguardar que uma determinada string apareça no log de um *container* e outra que passa uma

lista de *containers* e aguarda até que a string passada apareça no log de qualquer um dos *containers*. Como subimos o serviço via *container* durante o teste, foi possível criar cenários de teste em que há mais de uma instância do microsserviço executando, simulando falhas do tipo *fail-stop*.

Essas funções de sincronização foram muito úteis para os testes do monitor, por exemplo, para aguardar o monitor receber uma mensagem do *Kafka* na linha 59 do código 6. Usamos esse mecanismo também para identificar o momento em que um monitor recebe novas partições após simularmos uma falha de outro monitor.

Acreditamos que alguns pontos deixaram a desejar, por exemplo o usuário fica obrigado a chamar a função 'Done' do *Job* para informar que ele terminou e se a função não for chamada o estágio vai ficar travado aguardando. Já nos *jobs* que executam em mais de um estágio o desenvolvedor tem que fazer uma construção que utiliza o *select* para ficar verificando se o estágio final do *Job* terminou e a execução deve ser cancelada. Se o desenvolvedor não fizer isso, o *Job* vai ficar rodando até o final do teste. Essa limitação é decorrente do modelo adotado pela biblioteca que roda os *Jobs* de forma concorrente em rotinas independentes.

Além disso, com o assincronismo verificamos que alguns cenários são muito difíceis de serem reproduzidos de forma consistente. Isso porque a biblioteca ajuda na orquestração e ordenação dos eventos, mas não interfere no código ou no comportamento dos serviços executados. Dessa forma, procuramos utilizar o log dos *containers* para detectar a ocorrência de determinados eventos nos serviços. Um cenário em particular que quisemos reproduzir, foi o envio de uma mensagem exatamente no período em que o *Kafka* está fazendo o rebalanceamento de partições de um tópico. Queríamos identificar o momento que o *Kafka* entra em modo de rebalanceamento das partições após a falha de um Monitor, mas conseguimos apenas identificar que o rebalanceamento já havia ocorrido usando o log do Monitor. Nesse cenário, a abordagem de detecção via log acaba se tornando problemática. Conseguimos detectar apenas que o rebalanceamento havia ocorrido, mas não o momento que ele foi iniciado. Acreditamos que essa seja uma limitação da biblioteca: os estados transitórios das aplicações podem causar problemas, especialmente os que ocorrem por pouco tempo.

Essa abordagem funciona bem quando o serviço escreve no log informações de sua execução. No monitor, por exemplo, apesar do problema do evento de rebalanceamento, o log contém informações sobre mensagens recebidas e encaminhadas para o *Kafka* e conseguimos verificar a ocorrência desses eventos apenas analisando o log do *container*. Dessa forma, o uso do log para a sin-

cronização também acaba sendo útil apenas se o serviço sendo testado escreve no log informações relevantes para o cenário que queremos testar. Se o serviço não logar a informação que precisamos, só conseguimos testar o resultado final sem verificar os eventos intermediários.

No serviço do TJRJ que testemos, por exemplo, em alguns momentos tivemos que utilizar esperas arbitrárias porque determinados eventos não constavam no log do serviço. Essas esperas acabam fazendo com que o teste fique mais demorado, além de poder criar problemas de inconsistência se a máquina que rodar os testes for mais lenta ou estiver mais sobrecarregada.

Uma questão que pode ajudar bastante nos testes é a possibilidade de configuração dos serviços que vamos testar. O serviço de complementação do TJRJ possuía diversas maneiras de configuração, podendo alterar as *URLS* dos serviços que ele depende, o nome dos tópicos do *Kafka* usados, dentre outras coisas. Uma configuração que ele não possibilita e acabou nos atrapalhando bastante é a de definir o identificador do grupo. Isso acabava gerando uma intermitência ao rodar um teste logo após o outro porque o *Kafka* podia identificar que o *container* do teste anterior estava apenas demorando a responder gerando problemas de distribuição das partições. Para resolver isso foi necessário alterar as configurações do *Kafka* para reduzir bastante o tempo em que ocorre a desconexão de consumidores que não estão respondendo. Mesmo assim, em alguns casos precisamos colocar algumas esperas no scrip de teste com tempo suficiente para que os *containers* anteriores fossem removidos do grupo antes de enviar messagens para um tópico.

Acreditamos que para facilitar o teste de serviços desses tipos de sistemas, eles devem intrumentar adequadamente o log do serviço com o máximo de infomações para permitir que os eventos ocorridos durante sua execução possam ser detectados durante a execução de testes. Uma boa prática seria o uso de níveis de log definidos na hora de executar a aplicação permitindo alternar entre modos mais e menos verbosos.

5
Conclusão

Neste trabalho investigamos como lidar com a complexidade de testar sistemas distribuídos, em especial sistemas baseados em microsserviços com comunicação assíncrona através de filas de mensagens. Um dos principais fatores que contribuem para essa complexidade é a natureza não determinística desses sistemas. Experimentamos desenvolver uma biblioteca que garantisse a ordenação dos eventos em sistemas baseados em microsserviços com comunicação usando filas de mensagens. O objetivo da biblioteca foi de auxiliar o desenvolvedor a criar scripts de teste em que ele conseguisse, baseado no estado do sistema, definir o momento em que cada etapa de seu código deveria rodar. Usamos essa biblioteca para criar testes de serviços reais que estão em uso no Tecgraf.

Criamos uma biblioteca que apesar de suas limitações se mostrou muito útil na criação de testes de microsserviços com comunicação assíncrona. Nas duas aplicações testadas fomos capazes de encontrar erros com o testes desenvolvidos e criar scripts de teste que reproduziam de forma consistente os eventos que causavam o problema. Nossa API para o *Docker* facilitou o uso de *containers* e fomos capazes de gerar testes que rodavam com mais de uma instância do serviço sob teste executando e simular falhas do tipo *fail-stop*, conseguindo sincronizar os eventos através do log desses serviços. Além disso, a API para o *Kafka* simplificou o uso da fila de mensagem e facilitou a criação de tópicos com configuração e conteúdo definidos pelo desenvolvedor, e também a produção e consumo de mensagens. Acreditamos que o uso da biblioteca pode simplificar a criação de testes para aplicações baseadas em microsserviço, permitindo que os eventos sejam ordenados da maneira que o desenvolvedor deseja durante o teste.

Os cenários em que a biblioteca não foi bem sucedida, foram aqueles em que não conseguimos detectar a ocorrência de eventos sem nos intrometermos no funcionamento do sistema. Um exemplo é o caso do rebalancing do *Kafka* que só conseguimos detectar pelo log do monitor. Não conseguimos impedir ou atrasar o final do rebalacing para gerar, de forma consistente, eventos enquanto ele estava acontecendo.

Além disso, discutimos sobre os tipos de linguagens utilizadas para testes e e quais as implicações dessa escolha. Acreditamos que, na maioria das vezes, uma linguagem de proposito geral que seja de conhecimento dos desenvolvedores torna-se mais adequada para o desenvolvimento dos testes,

uma vez que eles vão ter uma curva de aprendizado menor. Talvez uma exceção pode ser feita para desenvolvedores especializados em testes que, apesar de uma maior curva de aprendizado, podem se beneficiar de linguagens específicas para testes. Um outro ponto em favor do uso da linguagem geral, é fato de que ainda que o *framework* ou biblioteca não dê suporte para determinada operação, o desenvolvedor pode usar a linguagem para implementar a operação que deseja.

Quanto ao uso da linguagem Go, avaliamos que seu uso para o desenvolvimento da ferramenta facilitou bastante a execução e a sincronização de código concorrente na nossa biblioteca. Não tivemos dificuldade em criar mecanismos para executar o código dos *Jobs* em *goroutines* e fazer a sincronização para a execução de cada estágio usando apenas funcionalidades providas pela própria linguagem. O uso de canais em conjunto com o *select* permite criar códigos que lidam muito bem com execução assíncrona. Além disso, foi muito fácil encontrar e usar bibliotecas de terceiros para ajudar no desenvolvimento. Os pontos negativos ficam por conta do tratamento de erro que, por vezes, deixa o código embolado e pela biblioteca do *Kafka* que tem funcionalidades limitadas. Outro benefício do uso de Go é que a ferramenta de linha de comando facilita a execução de testes, permitindo por exemplo rodar testes especificos, arquivos especificos ou todos os arquivos em um projeto. Além disso, essa ferramenta facilita o download de bibliotecas externas e a publicação de bibliotecas desenvolvidas.

Apesar disso, outras linguagens que possuam mecanismos para a criação, sincronização e comunicação de código concorrente, poderiam utilizar o mesmo modelo que utilizamos na nossa biblioteca e obter resultados similares. Se Java fosse utilizado, por exemplo, teríamos o beneficio de ter uma biblioteca mais completa para a interação com o *Kafka* e o tratamento de erros através de exceções.

Uma melhoria que poderia ajudar a observar melhor o estado do *Kafka*, seria adicionar a nossa biblioteca métodos para usar a API *REST*[1] [2] do *Kafka* que parece ter mais informações do que a biblioteca de *Kafka* para Go.

Uma abordagem interessante, que pode ajudar na monitoração e nos testes nesse tipo de sistemas, é o padrão de projeto usado no *Kubernetes* chamado de *Operator*[3], que permite ao desenvolvedor acoplar uma lógica desenvolvida por ele quando um determinado evento ocorre no *Kubernetes*. Por exemplo, o desenvolvedor pode inserir um código que vai rodar quando o *Kubernetes* identifica a falha de um determinado serviço. Se o *Kafka*, por

[1] <https://developer.mozilla.org/pt-BR/docs/Glossary/REST>
[2] <https://docs.confluent.io/platform/current/kafka-rest/>
[3] <https://kubernetes.io/docs/concepts/extend-kubernetes/operator/>

exemplo, permitisse esse tipo de construção, seria possível inserir código para controlar de forma mais fácil os eventos durantes os testes.

Outro caminho que não exploramos no nosso trabalho mas pode ser interessante, é o de usar teste *Fuzz* (KLEES et al., 2018) para gerar eventos aleatórios no sistema e, na ocorrência de uma falha, conseguir capturar a sequência de eventos que a originou.

6
Referências bibliográficas

BANO, S. et al. A novel approach to distributed model aggregation using apache kafka. In: **Proceedings of the 2nd Workshop on Flexible Resource and Application Management on the Edge**. [S.l.: s.n.], 2022. p. 33–36.

BANO, S. et al. Kafkafed: Two-tier federated learning communication architecture for internet of vehicles. In: **2022 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)**. [S.l.: s.n.], 2022. p. 515–520.

BASIRI, A. et al. Chaos engineering. **IEEE Software**, IEEE, v. 33, n. 3, p. 35–41, 2016.

BERNSTEIN, D. Containers and cloud: From lxc to docker to kubernetes. **IEEE cloud computing**, IEEE, v. 1, n. 3, p. 81–84, 2014.

CHANDY, K. M.; LAMPORT, L. Distributed snapshots: Determining global states of distributed systems. **ACM Transactions on Computer Systems (TOCS)**, ACM New York, NY, USA, v. 3, n. 1, p. 63–75, 1985.

CUI, H.; CHEN, J. Test control via dos middleware instrumentation. In: **2005 IEEE Instrumentationand Measurement Technology Conference Proceedings**. [S.l.: s.n.], 2005. v. 3, p. 2344–2348.

DRAGONI, N. et al. Microservices: yesterday, today, and tomorrow. **Present and ulterior software engineering**, Springer, p. 195–216, 2017.

DU, Y. et al. A distributed message delivery infrastructure for connected vehicle technology applications. **IEEE Transactions on Intelligent Transportation Systems**, v. 19, n. 3, p. 787–801, 2018.

EUGSTER, P. T. et al. The many faces of publish/subscribe. **ACM computing surveys (CSUR)**, ACM New York, NY, USA, v. 35, n. 2, p. 114–131, 2003.

JIANG, Z. M.; HASSAN, A. E. A survey on load testing of large-scale software systems. **IEEE Transactions on Software Engineering**, IEEE, v. 41, n. 11, p. 1091–1118, 2015.

KLEES, G. et al. Evaluating fuzz testing. In: **Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security**. New York, NY, USA: Association for Computing Machinery, 2018. (CCS '18), p. 2123–2138. ISBN 9781450356930. Disponível em: <https://doi.org/10.1145/3243734.3243804>.

KREPS, J. et al. Kafka: A distributed messaging system for log processing. In: **Proceedings of the NetDB**. [S.l.: s.n.], 2011. v. 11, p. 1–7.

LAMPORT, L.; SHOSTAK, R.; PEASE, M. The byzantine generals problem. In: **Concurrency: the works of Leslie Lamport**. [S.l.: s.n.], 2019. p. 203–226.

LIMA, M. J. de et al. Csbase: 10 years of experience with a framework for batch processing in heterogeneous computing environments. **Technical Report, Tecgraf Institute/PUC-Rio**, 2015.

LIU, C.; RICHARDSON, D. J. Programming languages considered harmful in writing automated software tests. 1999.

MAYER, T. R. et al. Evaluating the robustness of publish/subscribe systems. In: IEEE. **2011 International Conference on P2P, Parallel, Grid, Cloud and Internet Computing**. [S.l.], 2011. p. 75–82.

MERNIK, M.; HEERING, J.; SLOANE, A. M. When and how to develop domain-specific languages. **ACM computing surveys (CSUR)**, ACM New York, NY, USA, v. 37, n. 4, p. 316–344, 2005.

MLADENOV, K.; WINSEN, S. V.; MAVRAKIS, C. Formal verification of the implementation of the MQTT protocol in iot devices. **SNE Master Research Projects 2016–2017**, 2017.

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

SCHIEFERDECKER, I.; DIN, G.; APOSTOLIDIS, D. Distributed functional and load tests for web services. **International Journal on Software Tools for Technology Transfer**, Springer, v. 7, n. 4, p. 351–360, 2005.

STEEN, M. V.; TANENBAUM, A. S. **Distributed systems**. [S.l.]: Maarten van Steen Leiden, The Netherlands, 2017. 206–210 p.

STEEN, M. V.; TANENBAUM, A. S. **Distributed systems**. [S.l.]: Maarten van Steen Leiden, The Netherlands, 2017. 429–433 p.

WILLCOCK, C. et al. **An introduction to TTCN-3**. [S.l.]: John Wiley & Sons, 2011.

WISKA, R. et al. Big sensor-generated data streaming using kafka and impala for data storage in wireless sensor network for co2 monitoring. In: **2016 International Workshop on Big Data and Information Security (IWBIS)**. [S.l.: s.n.], 2016. p. 97–102.

A
Código da biblioteca

O código desenvolvido está no GitHub no endereço: https://github.com/Pedro-Magalhaes/async-microservice-test

A.1
Stage

Código principal da biblioteca que coordena a execução de estágios e *Jobs*.

Código 10: Módulo Stage

```
1  package stage
2
3  import (
4    "log"
5    "sync"
6  )
7
8  type DoneCancelArgGet interface {
9    Done()
10   GetFuncArg() interface{}
11   Canceled() chan interface{}
12 }
13
14 type Stage struct {
15   Id           string
16   WaitGroup    *sync.WaitGroup
17   Jobs         []*Job
18   channel      chan interface{}
19 }
20
21 type Stages struct {
22   stages      map[string]*Stage
23   stagesArray []*Stage
24 }
25
26 type Job struct {
27   Begin   *Stage
28   End     *Stage
29   Work    func(DoneCancelArgGet)
30   funcArg interface{}
31   hasFinished bool
32 }
33
34 /* Job */
35
36 func (j *Job) GetFuncArg() interface{} {
37   return j.funcArg
38 }
39
40 // Informa ao estágio que o Job terminou
41 // Sempre deve ser chamada ao final do Job de estágio único
42 // o estágio não termina até que todos os Jobs tenham chamado essa função
43 func (j *Job) Done() {
44   if j.Begin == j.End && !j.hasFinished {
45     j.hasFinished = true //evita que o mesmo job chame done() mais de 1 vez
46     j.End.WaitGroup.Done()
47   }
48 }
49
50 // Retorna um canal que informa se o estágio final do Job terminou
51 // Deve ser usado em Jobs multiestágio para terminar quando seu estágio final terminar
52 func (j *Job) Canceled() chan interface{} {
53   return j.End.channel
54 }
55
56 /* Stage */
57
58 func CreateStage(id string) *Stage {
59   return &Stage{
60     Id:        id,
61     WaitGroup: &sync.WaitGroup{},
62     channel:   make(chan interface{}),
63     Jobs:      []*Job{},
64   }
65 }
66
67 func (s *Stage) Wait() {
68   s.WaitGroup.Wait()
69 }
70
71 // Cria um Job de um único estágio
72 func (st *Stage) AddJob(work func(DoneCancelArgGet), workArg interface{}) {
73   st.AddJobMultiStage(work, st, workArg)
74 }
75
76 // Adiciona um Job que inicia em um estágio e pode terminar em outro
77 func (st *Stage) AddJobMultiStage(work func(DoneCancelArgGet), endStage *Stage, workArg interface{})
       *Job {
78   job := Job{
79     Work:     work,
80     End:      endStage,
81     Begin:    st,
82     funcArg: workArg,
83     hasFinished: false,
84   }
85   if st == endStage {
```

```
86      log.Default().Println("Adicionando ao wait Group", st.Id)
87      endStage.WaitGroup.Add(1)
88    }
89    st.Jobs = append(st.Jobs, &job)
90    return &job
91  }
92
93  // Roda um estágio
94  func (st *Stage) Run() {
95    for _, job := range st.Jobs {
96      go st.runWork(job)
97    }
98    st.Wait()
99    log.Default().Println("Finalizando stage: ", st.Id)
100   close(st.channel)
101 }
102
103 func (st *Stage) runWork(j *Job) {
104   go j.Work(j)
105 }
106
107 /* Stages */
108
109 func CreateStages() *Stages {
110   return &Stages{
111     make(map[string]*Stage),
112     []*Stage{},
113   }
114 }
115
116 // Adiciona um estágio
117 func (s *Stages) AddStage(st *Stage) *Stages {
118   s.stagesArray = append(s.stagesArray, st)
119   s.stages[st.Id] = st
120   return s
121 }
122
123 // Adiciona um array de estágios
124 func (s *Stages) AddStages(stages []*Stage) *Stages {
125   s.stagesArray = append(s.stagesArray, stages...)
126   for _, st := range stages {
127     s.stages[st.Id] = st
128   }
129   return s
130 }
131
132 // retorna um estágio dado um id
133 func (s *Stages) GetStage(id string) *Stage {
134   return s.stages[id]
135 }
136
137 // Roda todos os estágios internos
138 func (s *Stages) Run() {
139   for _, s := range s.stagesArray {
140     log.Default().Println("Runnig stage: ", s.Id)
141     s.Run()
142   }
143 }
```

A.2 Kafka

Código para facilitar a interação com o *Kafka*.

Código 11: Módulo Kafkatest

```
1  package kafkatest
2
3  import (
4    "context"
5    "errors"
6    "log"
7    "strings"
8    "time"
9
10   "github.com/Pedro-Magalhaes/async-microservice-test/pkg/topic"
11   "github.com/confluentinc/confluent-kafka-go/kafka"
12 )
13
14 const (
15   OffsetEarliest = "earliest"
16   OffsetLatest   = "latest"
17 )
18
19 type KafkaHelper struct {
20   server         string
21   kConfig        kafka.ConfigMap
22   kafkaProducer  *kafka.Producer
23   kafkaConsumer  *kafka.Consumer
24   kafkaAdmClient *kafka.AdminClient
25   // Colocar array para mensagens de cada tópico? O que ocorreria em um topico muito movimentado
26 }
27
28 func NewKafka(kConfig kafka.ConfigMap) (*KafkaHelper, error) {
29   s, err := kConfig.Get("bootstrap.servers", "")
30   if err != nil {
31     return nil, err
32   }
33   server, ok := s.(string)
34
35   if !ok {
36     return nil, errors.New("incorrect config, check the bootstrap.servers atribute")
37   }
38
39   if strings.Compare(server, "") == 0 {
40     server = defaultServer()
41   }
42   k := KafkaHelper{
43     server:  server,
```

```
44      kConfig: kConfig,
45    }
46    k.kafkaAdmClient, err = kafka.NewAdminClient(&kConfig)
47    if err != nil {
48      log.Println("Erro criando adm do kafka")
49      return nil, err
50    }
51
52    k.kafkaProducer, err = kafka.NewProducer(&kConfig)
53    if err != nil {
54      log.Println("Erro criando producer do kafka")
55      return nil, err
56    }
57
58    return &k, nil
59  }
60
61  func (k *KafkaHelper) NewConsumer(groupId, offsetConfig string) (*kafka.Consumer, error) {
62    consumerConfig := cloneConfigMap(k.kConfig)
63    consumerConfig.SetKey("group.id", groupId)
64    consumerConfig.SetKey("auto.offset.reset", offsetConfig)
65    // auto.offset.reset
66    return kafka.NewConsumer(&consumerConfig)
67  }
68
69  func (k *KafkaHelper) Produce(topic, msg string, confirmDelivery chan kafka.Event) error {
70    return k.kafkaProducer.Produce(&kafka.Message{TopicPartition: kafka.TopicPartition{Topic: &topic},
          Value: []byte(msg)}, confirmDelivery)
71  }
72
73  func (k *KafkaHelper) ProduceToPartition(topic, msg string, partition int32, confirmDelivery chan
       kafka.Event) error {
74    return k.kafkaProducer.Produce(&kafka.Message{
75      TopicPartition: kafka.TopicPartition{
76        Topic: &topic, Partition: partition,
77      },
78      Value: []byte(msg)}, confirmDelivery)
79  }
80
81  func (k KafkaHelper) CreateTopics(t *topic.TopicConfig) {
82    var topicNames []string = make([]string, len(t.Topics))
83    var specifications []kafka.TopicSpecification = make([]kafka.TopicSpecification, len(t.Topics))
84    for i, v := range t.Topics {
85      topicNames[i] = v.Name
86      specifications[i] = kafka.TopicSpecification{
87        Topic:             v.Name,
88        NumPartitions:     v.NumPartitions,
89        ReplicationFactor: 1,
90      }
91    }
92    _, e := k.kafkaAdmClient.DeleteTopics(context.Background(), topicNames, kafka.
          SetAdminOperationTimeout(time.Second * 2))
93    if e != nil {
94      log.Println("Não foi possivel deletar os topicos")
95      log.Println(topicNames)
96      panic(e)
97    }
98
99
100   // Limitação da lib Kafka de Go: precisamos aguardar o broker reconhecer a deleção dos tópicos
101   time.Sleep(time.Second * 2)
102
103   _, err := k.kafkaAdmClient.CreateTopics(context.Background(), specifications)
104   if err != nil {
105     log.Println("Não foi possivel criar os tópicos")
106     log.Println(topicNames)
107     panic(err)
108   }
109
110   // Limitação da lib Kafka de Go: precisamos aguardar o broker reconhecer a criação dos tópicos
111   time.Sleep(time.Second * 2)
112
113   for _, v := range t.Topics {
114     if len(v.Messages) > 0 {
115       log.Println("Deveria colocar as mensagens: ")
116       log.Println(v.Messages)
117     }
118     deliveryChan := make(chan kafka.Event)
119     for _, m := range v.Messages {
120       err := k.kafkaProducer.Produce(&kafka.Message{
121         TopicPartition: kafka.TopicPartition{Topic: &v.Name, Partition: int32(m.Partition)},
122         Value:          []byte(m.Message),
123       }, deliveryChan)
124       if err != nil {
125         panic(err)
126       }
127       <-deliveryChan
128     }
129   }
130 }
131
132 func (k KafkaHelper) DeleteTopics(t []string) {
133   k.kafkaAdmClient.DeleteTopics(context.Background(), t, kafka.SetAdminOperationTimeout(time.Second *
          2))
134 }
135
136 func (k KafkaHelper) ResetOffsets() { // remove o topico que armazena ofsets
137   k.kafkaAdmClient.DeleteTopics(context.Background(), []string{"__consumer_offsets"})
138 }
139
140 func (k KafkaHelper) Close() {
141   if k.kafkaAdmClient != nil {
142     k.kafkaAdmClient.Close()
143   }
144   if k.kafkaConsumer != nil {
145     k.kafkaConsumer.Close()
146   }
147   if k.kafkaProducer != nil {
148     k.kafkaProducer.Close()
149   }
150
151 }
152
153 func defaultServer() string {
154   return "localhost:9092"
155 }
156
157 func cloneConfigMap(original kafka.ConfigMap) kafka.ConfigMap {
158   m2 := make(kafka.ConfigMap)
159   for k, v := range original {
160     m2[k] = v
161   }
```

```
162   return m2
163 }
```

Modulo auxiliar para carregar as configurações de tópico de um arquivo JSON

Código 12: Módulo Topic

```
1  package topic
2
3  import (
4    "encoding/json"
5    "io"
6    "os"
7  )
8
9  type Messages struct {
10   Partition int16  `json:"partition"`
11   Message   string `json:"message"`
12   Key       string `json:"key,omitempty"`
13 }
14 type Topics struct {
15   Name          string     `json:"name"`
16   NumPartitions int        `json:"numPartitions"`
17   Messages      []Messages `json:"messages"`
18 }
19 type TopicConfig struct {
20   Topics []Topics `json:"topics"`
21 }
22
23 func LoadTopicConfig(jsonFile string) (*TopicConfig, error) {
24   file, err := os.Open(jsonFile)
25   if err != nil {
26     return nil, err
27   }
28   bytes, err := io.ReadAll(file)
29   if err != nil {
30     return nil, err
31   }
32   conf := TopicConfig{}
33
34   err = json.Unmarshal(bytes, &conf)
35
36   if err != nil {
37     return nil, err
38   }
39   return &conf, nil
40 }
```

A.3 Docker

Código para facilitar a sincronização dos containers *Docker*.

Código 13: Módulo Dockertest

```
1  package dockertest
2
3  import (
4    "context"
5    "errors"
6    "io"
7    "sync"
8    "sync/atomic"
9    "time"
10
11   "github.com/testcontainers/testcontainers-go"
12   "github.com/testcontainers/testcontainers-go/wait"
13 )
14
15
16 type waitResponse struct {
17   err       error
18   container testcontainers.Container
19 }
20
21 type waitStrategy struct {
22   container testcontainers.Container
23   st        wait.Strategy
24 }
25
26
27 // Retorna todo o log de um container.
28 // Não deve ser usado por logs muito grandes
29 func GetAllLog(c testcontainers.Container) (string, error)  {
30   read, err := c.Logs(context.Background())
31   if err != nil {
32     return "", err
33   }
34   text, err := io.ReadAll(read)
35   if err != nil {
36     return "", err
37   }
38   return string(text), nil
39 }
40
41 // Aguarda por um container
42 func WaitForLogMessage(message string, occurrence int, timeout time.Duration, container
       testcontainers.Container) error {
43   st := wait.ForLog(message).WithOccurrence(occurrence).WithPollInterval(time.Second / 2).
       WithStartupTimeout(timeout)
44   ctx, cancel := context.WithTimeout(context.Background(), timeout)
```

```
45    defer cancel()
46    c := waitForStrategies(ctx, []waitStrategy{{container: container, st: st}})
47    resp := <-c
48    return resp.err
49  }
50
51  // Recebe um lista de containers e retorna quando econtrar "messsage" x vezes no log de todos os
        containers.
52  func WaitForLogMessageForAll(message string, occurrence int, timeout time.Duration, containers []
        testcontainers.Container) error {
53    var errCount uint64 // contador thread safe
54    wg := sync.WaitGroup{}
55    for _, v := range containers {
56      wg.Add(1)
57      go func(c testcontainers.Container) {
58        defer wg.Done()
59        err := WaitForLogMessage(message, occurrence, timeout, c)
60        if err != nil {
61          atomic.AddUint64(&errCount, 1)
62        }
63      }(v)
64    }
65    wg.Wait()
66    if errCount > 0 {
67      return errors.New("error waiting for logs")
68    }
69    return nil
70  }
71
72  // Recebe um lista de containers e retorna o primeiro container que a "message" for encontrada x
        vezes.
73  func WaitForLogMessageForAny(message string,
74                occurrence int,
75                timeout time.Duration,
76                containers [] testcontainers.Container) (testcontainers.Container,error) {
77    ctx, cancel := context.WithTimeout(context.Background(), timeout)
78    defer cancel()
79    var strategies []waitStrategy
80    for _, v := range containers {
81      strategy := wait.ForLog(message).WithOccurrence(occurrence).WithPollInterval(time.Second/2).
            WithStartupTimeout(timeout)
82      strategies = append(strategies, waitStrategy{ st: strategy, container: v})
83    }
84    c := <- waitForStrategies(ctx, strategies)
85    return c.container, c.err
86  }
87
88  // Cria rotina e faz o WaitUntilReady. Retorna via canal quando cada terminar
89  // Se ocorrer um timeout retornar um erro além do container
90  func waitForStrategies(ctx context.Context, strategies []waitStrategy) chan waitResponse {
91    c := make(chan waitResponse, len(strategies)) // canal com buffer pra não ficar preso se não for
          lido
92    wg := sync.WaitGroup{}
93    for _, strategy := range strategies {
94      wg.Add(1)
95      go func(s waitStrategy) {
96        defer wg.Done()
97        select {
98        case <-ctx.Done():
99          c <- waitResponse{err: errors.New("timeout"), container: s.container}
100       default:
101         c <- waitResponse{err: s.st.WaitUntilReady(ctx, s.container), container: s.container}
102       }
103
104     }(strategy)
105   }
106   return c
107 }
```

B
Experimentos

B.1
Exemplo de uso da biblioteca

Código com exemplo simples de uso da biblioteca.

Código 14: Teste mínimo

```
1  package simple
2
3  import (
4    "context"
5    "io"
6    "net/http"
7    "net/http/httptest"
8    "testing"
9    "time"
10
11   "github.com/Pedro-Magalhaes/async-microservice-test/pkg/dockertest"
12   "github.com/Pedro-Magalhaes/async-microservice-test/pkg/stage"
13   "github.com/testcontainers/testcontainers-go"
14 )
15
16 func TestSimpleDocker(t *testing.T) {
17   var container testcontainers.Container
18   stages := stage.CreateStages()
19   st := stage.CreateStage("Primeiro")
20   st2 := stage.CreateStage("Segundo")
21   st3 := stage.CreateStage("Terceiro")
22
23   const comando = "while sleep 1; do echo 'oi'; done"
24   st.AddJob(func(dcag stage.DoneCancelArgGet) {
25     defer dcag.Done()
26     arg := dcag.GetFuncArg()
27     cmd, _ := arg.(string) //typecast
28     container, _ = testcontainers.GenericContainer(context.Background(),
29       testcontainers.GenericContainerRequest{
30         ContainerRequest: testcontainers.ContainerRequest{
31           Image: "ubuntu",
32           Cmd:   []string{"sh", "-c", cmd},
33         },
34         Started: true,
35       })
36   }, comando)
37
38   st2.AddJob(func(dcag stage.DoneCancelArgGet) {
39     defer dcag.Done()
40     err := dockertest.WaitForLogMessage("oi", 3, time.Second*5, container)
41     if err != nil {
42       t.Fatal(err) // timeout
43     }
44   }, nil)
45
46   mockServer := httptest.NewServer(http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
47     t.Log("mockServer got request")
48     time.Sleep(time.Second * 3)
49     w.Write([]byte("ok"))
50   }))
51
52   respChan := make(chan interface{})
53   st3.AddJob(func(dcag stage.DoneCancelArgGet) {
54     defer dcag.Done()
55     t.Log("Job 3")
56     go func() {
57       r, _ := http.Get(mockServer.URL)
58       body, _ := io.ReadAll(r.Body)
59       t.Log("Job 3 Got response: ", string(body))
60       close(respChan) // fecha e todos ouvindo recebem nil
61     }()
62     <-respChan
63   }, nil)
64
65   st3.AddJob(func(dcag stage.DoneCancelArgGet) {
66     defer dcag.Done()
67     t.Log("Job 4")
68     ticker := time.NewTicker(time.Millisecond * 500)
69     defer ticker.Stop()
70     for {
71       select {
72       case <-respChan:
73         t.Log("Job4 end")
74         return
75       case tick := <-ticker.C: // ocorre varias vezes até o caso de cima ocorrer
76         t.Log("Job4 tick", tick.Second())
77       }
78     }
79   }, nil)
80   stages.AddStages([]*stage.Stage{st, st2, st3})
81   stages.Run()
82 }
```

B.2 Monitor

Testes para o serviço de monitoração.

Código 15: Testes do monitor

```go
package monitor

import (
  "context"
  "fmt"
  "log"
  "os"
  "testing"
  "time"

  "github.com/Pedro-Magalhaes/async-microservice-test/pkg/dockertest"
  "github.com/Pedro-Magalhaes/async-microservice-test/pkg/kafkatest"
  "github.com/Pedro-Magalhaes/async-microservice-test/pkg/kafkatest/topic"
  "github.com/Pedro-Magalhaes/async-microservice-test/pkg/stage"
  "github.com/confluentinc/confluent-kafka-go/kafka"
  "github.com/docker/docker/api/types/container"
  "github.com/stretchr/testify/assert"
  "github.com/testcontainers/testcontainers-go"
  "github.com/testcontainers/testcontainers-go/wait"
)

const (
  targetFolder     = "/go/src/app/test_files"
  oneLineFileName  = "file_with_one_line.txt"
  monitorReadyLogMsg = "Iniciando espera por mensagens"
  monitorRecoveredStateLogMsg = "TIME SPENT WAITING FOR STATE RECOVERY"
)

var (
  oneLineFileTopic = "test_files__" + oneLineFileName
  tConfig = topic.TopicConfig{Topics: []topic.Topics{
    {Name: "monitor_interesse", NumPartitions: 2, Messages: []topic.Messages{}},
    {Name: oneLineFileTopic, NumPartitions: 1, Messages: []topic.Messages{}},
    {Name: "job_info", NumPartitions: 1, Messages: []topic.Messages{}},
    {Name: "monitor_estado", NumPartitions: 2, Messages: []topic.Messages{
      {Partition: 0, Message: "[]", Key: ""}, // estado "vazio"
      {Partition: 1, Message: "[]", Key: ""}, // estado "vazio"
    }},
  }}
  monitorMounts = testcontainers.Mounts(
    testcontainers.ContainerMount{
      Source: testcontainers.GenericBindMountSource{HostPath: "/local/ProgramasLocais/Documents/
          pessoal/Puc/mestrado/async-microservice-test/examples/monitor/files"},
      Target: testcontainers.ContainerMountTarget(targetFolder),
    },
    testcontainers.ContainerMount{
      Source: testcontainers.GenericBindMountSource{HostPath: "/local/ProgramasLocais/Documents/
          pessoal/Puc/mestrado/async-microservice-test/examples/monitor/config-monitor.json"},
      Target: "/go/src/app/config.json",
    },
  )
  monitorCRequest = testcontainers.ContainerRequest{
    Image:      "monitor:0.0.1-snapshot",
    WaitingFor: wait.ForLog(monitorRecoveredStateLogMsg).WithOccurrence(1).WithStartupTimeout(time.
        Second * 10),
    Mounts:     monitorMounts,
    HostConfigModifier: func(hc *container.HostConfig) {
      hc.NetworkMode = "host"
    },
  }
)

// Reproduz bug do monitor que não envia msg até que tenha ocorrido mudança no arquivo
func TestFileWithInitialContent(t *testing.T) {
  var containerMonitor testcontainers.Container
  stages := stage.CreateStages()
  st := stage.CreateStage("inicia_container")
  st2 := stage.CreateStage("envia msg e aguarda o monitor receber")
  st3 := stage.CreateStage("aguarda msg no topico do arquivo")

  // inicializa helper do kafka
  k, err := kafkatest.NewKafka(kafka.ConfigMap{"bootstrap.servers": "localhost:9092",
    "acks": "all"})
  if err != nil {
    t.Fatal(err)
  }
  // Cria os tópicos
  k.CreateTopics(&tConfig)
  // Cria o consumidor e subscreve ao tópico de interesse
  consumer, err := k.NewConsumer("test-group", kafkatest.OffsetEarliest)
  if err != nil {
    t.Fatal(err)
  }
  err = consumer.SubscribeTopics([]string{oneLineFileTopic}, nil)
  if err != nil {
    t.Fatal(err)
  }

  st.AddJob(func(dcag stage.DoneCancelArgGet) {
    defer dcag.Done()
    var err error
    containerMonitor, err = testcontainers.GenericContainer(context.Background(), testcontainers.
        GenericContainerRequest{
      ContainerRequest: monitorCRequest,
      Started:          true,
    })
    if err != nil {
      t.Fatal(err)
    }
  }, nil)

  st2.AddJob(func(dcag stage.DoneCancelArgGet) {
    defer dcag.Done()
    ch := make(chan kafka.Event)
    k.Produce("monitor_interesse", fmt.Sprintf(`{ "path": "%s", "project": "p1", "watch": true}`,
        oneLineFileName), ch)
    <-ch // Aguarda confirmação de entrega
```

```
103   }, nil)
104
105   st2.AddJob(func(dcag stage.DoneCancelArgGet) {
106     defer dcag.Done()
107     log.Println("Waiting for the monitor to recieve the msg")
108     err := dockertest.WaitForLogMessage(oneLineFileName, 1, time.Second*3, containerMonitor)
109     if err != nil {
110       t.Fatal(err) // timeout
111     }
112     log.Println("Monitor got the msg")
113   }, nil)
114
115   st3.AddJob(func(dcag stage.DoneCancelArgGet) {
116     defer dcag.Done()
117     const timeout = time.Second * 5
118     m, err := consumer.ReadMessage(timeout)
119     if err != nil {
120       t.Fatal(err) // timeout
121     }
122     assert.NotEmpty(t, m)
123   }, nil)
124
125   stages.AddStages([]*stage.Stage{st, st2, st3})
126   stages.Run()
127 }
128
129
130 // Teste com 2 réplicas do monitor
131 func TestReplicas(t *testing.T) {
132   var containerMonitor, containerMonitor2 testcontainers.Container
133   stages := stage.CreateStages()
134   st := stage.CreateStage("inicia_container")
135   st2 := stage.CreateStage("envia msg e aguarda o monitor receber")
136   st3 := stage.CreateStage("aguarda msg no topico do arquivo")
137
138   monitorCRequest.WaitingFor = wait.ForLog(monitorRecoveredStateLogMsg).WithOccurrence(1).
        WithStartupTimeout(time.Second * 10)
139
140   // inicializa helper do kafka
141   k, err := kafkatest.NewKafka(kafka.ConfigMap{"bootstrap.servers": "localhost:9092",
142     "acks": "all"})
143   if err != nil {
144     t.Fatal(err)
145   }
146   // Cria os tópicos
147
148   k.CreateTopics(&tConfig)
149   // Cria o consumidor e subscreve ao tópico de interesse
150   consumer, err := k.NewConsumer("test-group", kafkatest.OffsetEarliest)
151   if err != nil {
152     t.Fatal(err)
153   }
154   err = consumer.SubscribeTopics([]string{oneLineFileTopic}, nil)
155   if err != nil {
156     t.Fatal(err)
157   }
158
159   st.AddJob(func(dcag stage.DoneCancelArgGet) {
160     defer dcag.Done()
161     var err error
162     containerMonitor, err = testcontainers.GenericContainer(context.Background(), testcontainers.
          GenericContainerRequest{
163       ContainerRequest: monitorCRequest,
164       Started:          true,
165     })
166     if err != nil {
167       t.Fatal(err)
168     }
169   }, nil)
170
171   st.AddJob(func(dcag stage.DoneCancelArgGet) {
172     defer dcag.Done()
173     var err error
174     containerMonitor2, err = testcontainers.GenericContainer(context.Background(), testcontainers.
          GenericContainerRequest{
175       ContainerRequest: monitorCRequest,
176       Started:          true,
177     })
178     if err != nil {
179       t.Fatal(err)
180     }
181   }, nil)
182
183   st2.AddJob(func(dcag stage.DoneCancelArgGet) {
184     defer dcag.Done()
185     ch := make(chan kafka.Event)
186     k.Produce("monitor_interesse", fmt.Sprintf(`{ "path": "%s", "project": "p1", "watch": true}`,
          oneLineFileName), ch)
187     <-ch // Aguarda confirmação de entrega
188   }, nil)
189
190   var monitorInCharge testcontainers.Container
191
192   st2.AddJob(func(dcag stage.DoneCancelArgGet) {
193     defer dcag.Done()
194     log.Println("Waiting for any monitor to recive the file monitoring msg")
195     ct, err := dockertest.WaitForLogMessageForAny(
196       oneLineFileName,
197       1,
198       time.Second*20,
199       []testcontainers.Container{containerMonitor, containerMonitor2},
200     )
201     if err != nil {
202       t.Fatal(err) // timeout
203     }
204     log.Println("Monitor got the msg", ct.GetContainerID())
205     monitorInCharge = ct // o monitor que recebeu a mensagem é o que vai observar o arquivo
206   }, nil)
207
208   st3.AddJob(func(dcag stage.DoneCancelArgGet) {
209     defer dcag.Done()
210     // aguarda o monitor que ficou responsavel pelo arquivo iniciar o watcher
211     dockertest.WaitForLogMessage("Iniciando loop do watcher", 1, time.Second * 10, monitorInCharge)
212     mySt := "uma escrita no arquivo \n com três linhas \n fim!"
213     e := os.WriteFile("./files/" + oneLineFileName, []byte(mySt), 0777)
214     if e != nil {
215       t.Fatal(e)
216     }
217     const timeout = time.Second * 10
218     msg, err := consumer.ReadMessage(timeout) // aguarda mensagem no tópico de
219     if err != nil {
220       t.Fatal(err) // timeout
```

```
221     }
222     assert.NotEmpty(t, msg.Value) // teste simples para ver que não é uma msg vazia
223   }, nil)
224
225   stages.AddStages([]*stage.Stage{st, st2, st3})
226   stages.Run()
227 }
```

B.3 Complementação Judicial

Testes para o serviço de Complementação Judicial do TJRJ.

Código 16: Teste do TJRJ

```
1  package odrj
2
3  import (
4    "context"
5    "fmt"
6    "log"
7    "math/rand"
8    "net/http"
9    "net/http/httptest"
10   "strings"
11   "testing"
12   "time"
13
14   "github.com/Pedro-Magalhaes/async-microservice-test/pkg/dockertest"
15   "github.com/Pedro-Magalhaes/async-microservice-test/pkg/kafkatest"
16   "github.com/Pedro-Magalhaes/async-microservice-test/pkg/kafkatest/topic"
17   "github.com/Pedro-Magalhaes/async-microservice-test/pkg/stage"
18   "github.com/confluentinc/confluent-kafka-go/kafka"
19   "github.com/docker/docker/api/types/container"
20   "github.com/stretchr/testify/assert"
21   "github.com/testcontainers/testcontainers-go"
22   "github.com/testcontainers/testcontainers-go/wait"
23 )
24
25 const (
26   entryTopic = "demanda-submetida"
27   errorTopic = "demanda-falhada"
28   outTopic   = "resposta-complementacao-judicial"
29   odrImage = "repo.tecgraf.puc-rio.br:18089/odrtj/odr-complementacao-tj:master"
30   defaultOdrPort = 8888
31   defaultGelfPort = 12201
32 )
33
34 var portOffset = 0
35
36 func getRandTopicName(baseName, testName string) string {
37   return fmt.Sprintf("%s-%s-%d", baseName, testName, rand.Uint32())
38 }
39
40 func getNewContainerDefinition(mockServerUrl, entryTopic string) map[string]string {
41   cDef := map[string]string{
42     "ODR_KAFKA_HOST":                "localhost:9092",
43     "ODR_TJRJ_PROCESSOS_RETRY_DELAY": "100",
44     "ODR_TJRJ_PROCESSOS_TIMEOUT":     "1000",
45     "ODR_HTTP_PORT":                  fmt.Sprint(defaultOdrPort + portOffset),
46     "ODR_GELF_PORT":                  fmt.Sprint(defaultGelfPort + portOffset),
47     "ODR_TJRJ_PROCESSOS_URI":         mockServerUrl,
48     "ODR_TJRJ_OIDC_URI":              mockServerUrl + "/auth/realms/homologacao",
49     "ODR_TJRJ_OIDC_CLIENT_ID":        "odr",
50     "ODR_TJRJ_OIDC_CLIENT_SECRET":    "fake-client-secret-for-test",
51     "ODR_TJRJ_OIDC_USERNAME":         "fake-user",
52     "ODR_TJRJ_OIDC_PASSWORD":         "1234",
53     "ODR_KAFKA_IN_TOPIC":         entryTopic,
54   }
55   portOffset++
56   return cDef
57 }
58
59 func checkMessages(kafkaConsumer *kafka.Consumer, messageMap map[string][]string, t *testing.T) {
60   // Process messages
61   ev, err := kafkaConsumer.ReadMessage(100 * time.Millisecond)
62   if err != nil {
63     // Errors are informational and automatically handled by the consumer
64     return
65   }
66
67   t.Logf("Consumed event from topic %s: key = %-10s value = %s\n",
68     *ev.TopicPartition.Topic, string(ev.Key), string(ev.Value))
69   // adiciona ao mapa de mensagens
70   messageMap[*ev.TopicPartition.Topic] = append(messageMap[*ev.TopicPartition.Topic], string(ev.Value
         ))
71 }
72
73 func createFakeServer(authStatus, tjStatus int) *httptest.Server {
74   return httptest.NewServer(http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
75
76     if strings.Contains(r.URL.Path, "auth") { // requisição token
77       w.WriteHeader(authStatus)
78       w.Write([]byte(`{"access_token": "token-de-acesso-para-chamada"}`))
79       return
80     }
81     w.Header().Set("content-type", "application/json")
82     w.WriteHeader(tjStatus)
83     w.Write([]byte("[\"111111111.2019.8.19.0206\", \"222222222.2021.81.9.0208\",
           \"333333333.2022.81.9.0029\"]"))
84   }))
85 }
86
87 func TestWrongMsgFormat(t *testing.T) {
88   var containerObj testcontainers.Container
89   stages := stage.CreateStages()
90   st := stage.CreateStage("Primeiro")
91   st2 := stage.CreateStage("Segundo")
92   st3 := stage.CreateStage("Terceiro")
```

```
93
94    mockTJsvr := createFakeServer(200,200)
95    defer mockTJsvr.Close()
96
97    env := getNewContainerDefinition(mockTJsvr.URL, entryTopic)
98    r := testcontainers.ContainerRequest{
99      Image: odrImage,
100     WaitingFor: wait.ForLog("Profile prod activated").WithPollInterval(time.Second / 2).
            WithStartupTimeout(time.Second * 10),
101     Env: env,
102     HostConfigModifier: func(hc *container.HostConfig) {
103       hc.NetworkMode = "host"
104     },
105   }
106
107   k, err := kafkatest.NewKafka(kafka.ConfigMap{"bootstrap.servers": "localhost:9092",
108     "acks": "all"})
109   if err != nil {
110     t.Fatal(err)
111   }
112   defer k.Close()
113   k.CreateTopics(&topic.TopicConfig{Topics: []topic.Topics{
114     {Name: entryTopic, NumPartitions: 1, Messages: []topic.Messages{}},
115     {Name: outTopic, NumPartitions: 1, Messages: []topic.Messages{}},
116     {Name: errorTopic, NumPartitions: 1, Messages: []topic.Messages{}},
117   }})
118   defer k.DeleteTopics([]string{entryTopic, outTopic, errorTopic})
119
120   // Inicia o serviço de complementação e aguarda resposta do health check para terminar
121   st.AddJob(func(dcag stage.DoneCancelArgGet) {
122     defer dcag.Done()
123     var err error
124     log.Println("Inicio job inicia docker")
125     containerOdj, err = testcontainers.GenericContainer(context.Background(), testcontainers.
            GenericContainerRequest{
126       ContainerRequest: r,
127       Started:          true,
128     })
129     if err != nil {
130       t.Fatal(err)
131     }
132     log.Println("Fim job inicia docker", containerOdj.IsRunning())
133   }, nil)
134
135   st2.AddJob(func(dcag stage.DoneCancelArgGet) {
136     produceChan := make(chan kafka.Event)
137     time.Sleep(time.Second * 10) // Health check da complementação fica 200 antes de estar pronto pra
            consumir
138
139     err = k.Produce(entryTopic, `{"idDemanda": "234"}`, produceChan)
140
141     if err != nil {
142       t.Fatal(err)
143     }
144     select {
145     case <-time.After(5 * time.Second):
146       t.Fatal("Não recebeu confirmação de mensagem")
147     case <-produceChan:
148       dcag.Done()
149     }
150
151   }, nil)
152
153   // Job para verificar que o tópico demanda-falhada recebeu mensagem
154   st3.AddJob(func(dcag stage.DoneCancelArgGet) {
155     defer dcag.Done()
156     time.Sleep(time.Second * 10) // espera a recuperação de erro
157
158     assert.Equal(t, true, containerOdj.IsRunning())
159
160     r, err := http.Get("http://localhost:" + env["ODR_HTTP_PORT"] + "/q/health/live")
161     if err != nil {
162       t.Fatal(err)
163     }
164     assert.Equal(t, 200, r.StatusCode)
165   }, nil)
166   stages.AddStages([]*stage.Stage{st, st2, st3})
167   stages.Run()
168   if containerOdj != nil {
169     containerOdj.Terminate(context.Background())
170   }
171   // DEBUG
172   // t.Log(dockertest.GetAllLog(containerOdj))
173 }
174
175 func TestEmptyMsgFormat(t *testing.T) {
176   var containerOdj testcontainers.Container
177   stages := stage.CreateStages()
178   st := stage.CreateStage("Primeiro")
179   st2 := stage.CreateStage("Segundo")
180   st3 := stage.CreateStage("Terceiro")
181
182   mockTJsvr := createFakeServer(200,200)
183   defer mockTJsvr.Close()
184
185   env := getNewContainerDefinition(mockTJsvr.URL, entryTopic)
186   r := testcontainers.ContainerRequest{
187     Image: odrImage,
188     WaitingFor: wait.ForLog("Profile prod activated").WithPollInterval(time.Second / 2).
            WithStartupTimeout(time.Second * 10),
189     Env: env,
190     HostConfigModifier: func(hc *container.HostConfig) {
191       hc.NetworkMode = "host"
192     },
193   }
194
195   k, err := kafkatest.NewKafka(kafka.ConfigMap{"bootstrap.servers": "localhost:9092",
196     "acks": "all"})
197   if err != nil {
198     t.Fatal(err)
199   }
200   defer k.Close()
201   k.CreateTopics(&topic.TopicConfig{Topics: []topic.Topics{
202     {Name: entryTopic, NumPartitions: 1, Messages: []topic.Messages{}},
203     {Name: outTopic, NumPartitions: 1, Messages: []topic.Messages{}},
204     {Name: errorTopic, NumPartitions: 1, Messages: []topic.Messages{}},
205   }})
206   defer k.DeleteTopics([]string{entryTopic, outTopic, errorTopic})
207
208   // Inicia o serviço de complementação e aguarda resposta do health check para terminar
209   st.AddJob(func(dcag stage.DoneCancelArgGet) {
210     defer dcag.Done()
```

```
211     var err error
212     log.Println("Inicio job inicia docker")
213     containerOdj, err = testcontainers.GenericContainer(context.Background(), testcontainers.
            GenericContainerRequest{
214       ContainerRequest: r,
215       Started:          true,
216     })
217     if err != nil {
218       t.Fatal(err)
219     }
220     log.Println("Fim job inicia docker", containerOdj.IsRunning())
221   }, nil)
222
223   st2.AddJob(func(dcag stage.DoneCancelArgGet) {
224     produceChan := make(chan kafka.Event)
225     time.Sleep(time.Second * 10) // Health check da complementação fica 200 antes de estar pronto pra
            consumir
226
227     err = k.Produce(entryTopic, `{}`, produceChan)
228
229     if err != nil {
230       t.Fatal(err)
231     }
232     select {
233     case <-time.After(5 * time.Second):
234       t.Fatal("Não recebeu confirmação de mensagem")
235     case <-produceChan:
236       dcag.Done()
237     }
238
239   }, nil)
240
241   // Job para verificar que o tópico demanda-falhada recebeu mensagem
242   st3.AddJob(func(dcag stage.DoneCancelArgGet) {
243     defer dcag.Done()
244     time.Sleep(time.Second * 10)
245
246     assert.Equal(t, true, containerOdj.IsRunning())
247
248     r, err := http.Get("http://localhost:" + env["ODR_HTTP_PORT"] + "/q/health/live")
249     if err != nil {
250       t.Fatal(err)
251     }
252     assert.Equal(t, 200, r.StatusCode)
253   }, nil)
254   stages.AddStages([]*stage.Stage{st, st2, st3})
255   stages.Run()
256   if containerOdj != nil {
257     containerOdj.Terminate(context.Background())
258   }
259   // t.Log(dockertest.GetAllLog(containerOdj)) //DEBUG
260 }
261
262 func TestFailHttpOdj(t *testing.T) {
263   var containerOdj testcontainers.Container
264   topicMessages := make(map[string][]string)
265   stages := stage.CreateStages()
266   st := stage.CreateStage("Primeiro")
267   st2 := stage.CreateStage("Segundo")
268   st3 := stage.CreateStage("Terceiro")
269
270   mockTJsvr := createFakeServer(200,500)
271   defer mockTJsvr.Close()
272   env := getNewContainerDefinition(mockTJsvr.URL, entryTopic)
273   r := testcontainers.ContainerRequest{
274     Image: odrImage,
275     WaitingFor: wait.ForLog("Profile prod activated").WithPollInterval(time.Second / 2).
            WithStartupTimeout(time.Second * 10),
276     Env: env,
277     HostConfigModifier: func(hc *container.HostConfig) {
278       hc.NetworkMode = "host"
279     },
280   }
281   k, err := kafkatest.NewKafka(kafka.ConfigMap{"bootstrap.servers": "localhost:9092",
282     "acks": "all"})
283   if err != nil {
284     t.Fatal(err)
285   }
286   defer k.Close()
287   k.CreateTopics(&topic.TopicConfig{Topics: []topic.Topics{
288     {Name: entryTopic, NumPartitions: 1, Messages: []topic.Messages{}},
289     {Name: outTopic, NumPartitions: 1, Messages: []topic.Messages{}},
290     {Name: errorTopic, NumPartitions: 1, Messages: []topic.Messages{}},
291   }})
292   defer k.DeleteTopics([]string{entryTopic, outTopic, errorTopic})
293
294   consumer, err := k.NewConsumer("test-group", kafkatest.OffsetEarliest)
295   if err != nil {
296     t.Fatal(err)
297   }
298
299   err = consumer.SubscribeTopics([]string{entryTopic, errorTopic}, nil)
300   if err != nil {
301     t.Fatal(err)
302   }
303
304   // Inicia o serviço de complementação e aguarda resposta do health check para terminar
305   st.AddJob(func(dcag stage.DoneCancelArgGet) {
306     defer dcag.Done()
307     var err error
308     log.Println("Inicio job inicia docker")
309     containerOdj, err = testcontainers.GenericContainer(context.Background(), testcontainers.
            GenericContainerRequest{
310       ContainerRequest: r,
311       Started:          true,
312     })
313     if err != nil {
314       t.Fatal(err)
315     }
316     log.Println("Fim job inicia docker", containerOdj.IsRunning())
317   }, nil)
318
319   // Job multi estágio que vai ficar periodicamente checando mensagens nos tópicos subscritos
320   st.AddJobMultiStage(func(dcag stage.DoneCancelArgGet) {
321     ticker := time.NewTicker(time.Millisecond * 100)
322     defer ticker.Stop()
323     checkMessages(consumer, topicMessages, t)
324     for {
325       select {
326       case <-dcag.Canceled(): // cancelado ao final do estagio st3
327         t.Log("Job de checagem de mensagens cancelado")
328         return
```

```
329         case <-ticker.C:
330           // t.Log("check message")
331           checkMessages(consumer, topicMessages, t)
332         }
333       }
334     }, st3, nil)
335
336     st2.AddJob(func(dcag stage.DoneCancelArgGet) {
337       produceChan := make(chan kafka.Event)
338       time.Sleep(time.Second * 8) // Health check da complementação fica 200 antes de estar pronto pra
              consumir
339
340       err = k.Produce(entryTopic, `{"documentoDemandada": 22211133344, "documentoDemandante":
              22211133344, "idDemanda": "234"}`, produceChan)
341
342       if err != nil {
343         t.Fatal(err)
344       }
345       select {
346       case <-time.After(5 * time.Second):
347         t.Fatal("Não recebeu confirmação de mensagem")
348       case <-produceChan:
349         dcag.Done()
350       }
351
352     }, nil)
353
354     // Job para verificar que o tópico demanda-falhada recebeu mensagem
355     st3.AddJob(func(dcag stage.DoneCancelArgGet) {
356       defer dcag.Done()
357       ticker := time.NewTicker(time.Second)
358       defer ticker.Stop()
359       to := time.NewTimer(10 * time.Second)
360       defer to.Stop()
361       for {
362         select {
363         case <-to.C: // timeout
364           t.Fatal("Não recebeu mensagem no canal de falha")
365           return
366         case <-ticker.C:
367           t.Log("Teste array demanda-falhada")
368           if len(topicMessages[errorTopic]) > 0 {
369             t.Log(topicMessages[errorTopic][0])
370             return
371           }
372         }
373       }
374     }, nil)
375     stages.AddStages([]*stage.Stage{st, st2, st3})
376     stages.Run()
377     if containerOdj != nil {
378       containerOdj.Terminate(context.Background())
379     }
380   }
381
382   func TestCorrectBehaviourOdj(t *testing.T) {
383     var containerOdj testcontainers.Container
384     topicMessages := make(map[string][]string)
385     stages := stage.CreateStages()
386     st := stage.CreateStage("Primeiro")
387     st2 := stage.CreateStage("Segundo")
388     st3 := stage.CreateStage("Terceiro")
389
390     mockTJsvr := createFakeServer(200,200)
391     defer mockTJsvr.Close()
392
393     r := testcontainers.ContainerRequest{
394       Image:        odrImage,
395       WaitingFor: wait.ForLog("Profile prod activated").WithPollInterval(time.Second / 2).
              WithStartupTimeout(time.Second * 10),
396       Env: getNewContainerDefinition(mockTJsvr.URL, entryTopic),
397       HostConfigModifier: func(hc *container.HostConfig) {
398         hc.NetworkMode = "host"
399       },
400     }
401
402     k, err := kafkatest.NewKafka(kafka.ConfigMap{"bootstrap.servers": "localhost:9092",
403       "acks": "all"})
404     if err != nil {
405       t.Fatal(err)
406     }
407     defer k.Close()
408     k.CreateTopics(&topic.TopicConfig{Topics: []topic.Topics{
409       {Name: entryTopic, NumPartitions: 1, Messages: []topic.Messages{}},
410       {Name: outTopic, NumPartitions: 1, Messages: []topic.Messages{}},
411       {Name: errorTopic, NumPartitions: 1, Messages: []topic.Messages{}},
412     }})
413     defer k.DeleteTopics([]string{entryTopic, outTopic, errorTopic})
414
415     consumer, err := k.NewConsumer("test-group", kafkatest.OffsetEarliest)
416     if err != nil {
417       t.Fatal(err)
418     }
419
420     err = consumer.SubscribeTopics([]string{entryTopic, errorTopic, outTopic}, nil)
421     if err != nil {
422       t.Fatal(err)
423     }
424
425     // Inicia o serviço de complementação e aguarda resposta do health check para terminar
426     st.AddJob(func(dcag stage.DoneCancelArgGet) {
427       defer dcag.Done()
428       var err error
429       log.Println("Inicio job inicia docker")
430       containerOdj, err = testcontainers.GenericContainer(context.Background(), testcontainers.
              GenericContainerRequest{
431         ContainerRequest: r,
432         Started:          true,
433       })
434       if err != nil {
435         t.Fatal(err)
436       }
437     }, nil)
438
439     // Job multi estágio que vai ficar periodicamente checando mensagens nos tópicos subscritos
440     st.AddJobMultiStage(func(dcag stage.DoneCancelArgGet) {
441       ticker := time.NewTicker(time.Millisecond * 100)
442       defer ticker.Stop()
443       checkMessages(consumer, topicMessages, t)
444       for {
445         select {
446         case <-dcag.Canceled(): // cancelado ao final do estagio st3
```

```
447         t.Log("Job de checagem de mensagens cancelado")
448         return
449       case <-ticker.C:
450         checkMessages(consumer, topicMessages, t)
451       }
452     }
453   }, st3, nil)
454
455   st2.AddJob(func(dcag stage.DoneCancelArgGet) {
456     produceChan := make(chan kafka.Event)
457     time.Sleep(time.Second * 8)
458     err = k.Produce(entryTopic, `{"documentoDemandada": 12211133344, "documentoDemandante":
          12211133344, "idDemanda": "134"}`, produceChan)
459     err = k.Produce(entryTopic, `{"documentoDemandada": 22222222222, "documentoDemandante":
          22211133344, "idDemanda": "234"}`, produceChan)
460     err = k.Produce(entryTopic, `{"documentoDemandada": 32233333333, "documentoDemandante":
          32211133344, "idDemanda": "334"}`, produceChan)
461     if err != nil {
462       t.Fatal(err)
463     }
464     for i := 0; i < 3; i++ {
465       select {
466       case <-time.After(5 * time.Second):
467         t.Fatal("Não recebeu confirmação de mensagem")
468       case <-produceChan:
469         if i == 2 {
470           dcag.Done()
471         }
472       }
473     }
474
475   }, nil)
476   st2.AddJob(func(dcag stage.DoneCancelArgGet) {
477     defer dcag.Done()
478     err := dockertest.WaitForLogMessage("Recebida mensagem de demanda", 3, time.Second*12,
          containerOdj)
479     if err != nil {
480       t.Fatal("ODJ não recebeu mensagens", err)
481     }
482   }, nil)
483
484   // Job para verificar que o tópico demanda-falhada recebeu uma nova mensagem
485   st3.AddJob(func(dcag stage.DoneCancelArgGet) {
486     defer dcag.Done()
487     ticker := time.NewTicker(time.Second)
488     defer ticker.Stop()
489     to := time.NewTimer(10 * time.Second)
490     defer to.Stop()
491     for {
492       select {
493       case <-to.C: // timeout
494         t.Fatal("Timeout")
495         return
496       case <-ticker.C:
497         if len(topicMessages[errorTopic]) > 0 { // recebeu msg no tópico de falha!
498           t.Fatal(topicMessages[errorTopic][0])
499           return
500         } else if len(topicMessages[outTopic]) == 3 {
501           t.Log("Sucess. All msgs correctly processed")
502           return
503         }
504       }
505     }
506   }, nil)
507
508   stages.AddStages([]*stage.Stage{st, st2, st3})
509   stages.Run()
510   if containerOdj != nil {
511     containerOdj.Terminate(context.Background())
512   }
513 }
514
515 func TestReplicaOdj(t *testing.T) {
516   var containerOdj, containerOdj2 testcontainers.Container
517   topicMessages := make(map[string][]string)
518   var tEntry = getRandTopicName(entryTopic, t.Name())
519
520   stages := stage.CreateStages()
521   st := stage.CreateStage("Primeiro")
522   st2 := stage.CreateStage("Segundo")
523   st3 := stage.CreateStage("Terceiro")
524
525   mockTJsvr := createFakeServer(200,200)
526   defer mockTJsvr.Close()
527
528   r1 := testcontainers.ContainerRequest{
529     Image:       odrImage,
530     WaitingFor: wait.ForLog("Profile prod activated").WithPollInterval(time.Second / 2).
          WithStartupTimeout(time.Second * 10),
531     Env:getNewContainerDefinition(mockTJsvr.URL, tEntry),
532     HostConfigModifier: func(hc *container.HostConfig) {
533       hc.NetworkMode = "host"
534     },
535   }
536
537   r2 := testcontainers.ContainerRequest{
538     Image:       odrImage,
539     WaitingFor: wait.ForLog("Profile prod activated").WithPollInterval(time.Second / 2).
          WithStartupTimeout(time.Second * 10),
540     Env: getNewContainerDefinition(mockTJsvr.URL, tEntry),
541     HostConfigModifier: func(hc *container.HostConfig) {
542       hc.NetworkMode = "host"
543     },
544   }
545
546   k, err := kafkatest.NewKafka(kafka.ConfigMap{"bootstrap.servers": "localhost:9092",
547     "acks": "all"})
548   if err != nil {
549     t.Fatal(err)
550   }
551   defer k.Close()
552   k.CreateTopics(&topic.TopicConfig{Topics: []topic.Topics{
553     {Name: tEntry, NumPartitions: 2, Messages: []topic.Messages{}},
554     {Name: outTopic, NumPartitions: 1, Messages: []topic.Messages{}},
555     {Name: errorTopic, NumPartitions: 1, Messages: []topic.Messages{}},
556   }})
557   defer k.DeleteTopics([]string{tEntry, outTopic, errorTopic})
558
559   consumer, err := k.NewConsumer("test-group", kafkatest.OffsetEarliest)
560   if err != nil {
561     t.Fatal(err)
562   }
```

```
563
564   err = consumer.SubscribeTopics([]string{errorTopic, outTopic}, nil)
565   if err != nil {
566     t.Fatal(err)
567   }
568
569   // Inicia o serviço de complementação e aguarda resposta do health check para terminar
570   st.AddJob(func(dcag stage.DoneCancelArgGet) {
571     defer dcag.Done()
572     var err error
573     log.Println("Inicio job inicia docker 1")
574     containerOdj, err = testcontainers.GenericContainer(context.Background(), testcontainers.
          GenericContainerRequest{
575       ContainerRequest: r1,
576       Started:          true,
577     })
578     if err != nil {
579       t.Fatal(err)
580     }
581
582   }, nil)
583
584   st.AddJob(func(dcag stage.DoneCancelArgGet) {
585     defer dcag.Done()
586     var err error
587     log.Println("Inicio job inicia docker 2")
588     containerOdj2, err = testcontainers.GenericContainer(context.Background(), testcontainers.
          GenericContainerRequest{
589       ContainerRequest: r2,
590       Started:          true,
591     })
592     if err != nil {
593       t.Fatal(err)
594     }
595   }, nil)
596
597   // Job multi estágio que vai ficar periodicamente checando mensagens nos tópicos subscritos
598   st.AddJobMultiStage(func(dcag stage.DoneCancelArgGet) {
599     ticker := time.NewTicker(time.Millisecond * 100)
600     defer ticker.Stop()
601     checkMessages(consumer, topicMessages, t)
602     for {
603       select {
604       case <-dcag.Canceled(): // cancelado ao final do estagio st3
605         t.Log("Job de checagem de mensagens cancelado")
606         return
607       case <-ticker.C:
608         checkMessages(consumer, topicMessages, t)
609       }
610     }
611   }, st3, nil)
612
613   st2.AddJob(func(dcag stage.DoneCancelArgGet) {
614     // Mesmo com health-check e/ou aguardando o log do container, precisamos esperar ou o container
          não recebe as msgs
615     // Isso é necessário no teste porque o tópico inicia vazio e sem informações de offset de
          mensagens
616     time.Sleep(time.Second * 15)
617     produceChan := make(chan kafka.Event)
618     err = k.ProduceToPartition(tEntry, `{"documentoDemandada": 12211133344, "documentoDemandante":
          12211133344, "idDemanda": "134"}`, 0, produceChan)
619     err = k.ProduceToPartition(tEntry, `{"documentoDemandada": 22222222222, "documentoDemandante":
          22211133344, "idDemanda": "234"}`, 1, produceChan)
620     if err != nil {
621       t.Fatal(err)
622     }
623     for i := 0; i < 2; i++ {
624       select {
625       case <-time.After(5 * time.Second):
626         t.Fatal("Não recebeu confirmação de mensagem")
627       case <-produceChan:
628         if i == 1 {
629           dcag.Done()
630         }
631       }
632     }
633
634   }, nil)
635   st3.AddJob(func(dcag stage.DoneCancelArgGet) {
636     defer dcag.Done()
637     err := dockertest.WaitForLogMessage("Recebida mensagem de demanda", 1, time.Second*12,
          containerOdj)
638     if err != nil {
639       t.Fatal("ODJ não recebeu mensagens", err)
640     }
641   }, nil)
642
643   st3.AddJob(func(dcag stage.DoneCancelArgGet) {
644     defer dcag.Done()
645     err := dockertest.WaitForLogMessage("Recebida mensagem de demanda", 1, time.Second*12,
          containerOdj2)
646     if err != nil {
647       t.Fatal("ODJ 2 não recebeu mensagens", err)
648     }
649   }, nil)
650
651   // Job para verificar que o tópico demanda-falhada recebeu uma nova mensagem
652   st3.AddJob(func(dcag stage.DoneCancelArgGet) {
653     defer dcag.Done()
654     ticker := time.NewTicker(time.Second)
655     defer ticker.Stop()
656     to := time.NewTimer(15 * time.Second)
657     defer to.Stop()
658     for {
659       select {
660       case <-to.C: // timeout
661         t.Fatal("Timeout")
662         return
663       case <-ticker.C:
664         if len(topicMessages[errorTopic]) > 0 { // recebeu msg no tópico de falha!
665           t.Log("Fail. Got msg on the failureTopic")
666           t.Fatal(topicMessages[errorTopic][0])
667           return
668         } else if len(topicMessages[outTopic]) == 2 {
669           t.Log("Sucess. All msgs correctly processed")
670           return
671         }
672       }
673     }
674   }, nil)
675
676   stages.AddStages([]*stage.Stage{st, st2, st3})
```

```
677   stages.Run()
678   t.Log(dockertest.GetAllLog(containerOdj))
679   if containerOdj != nil {
680     containerOdj.Terminate(context.Background())
681   }
682   if containerOdj2 != nil {
683     containerOdj2.Terminate(context.Background())
684   }
685 }
```