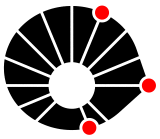

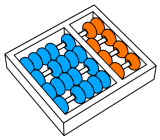

Testes de Robustez em Sistemas Baseados em Microsserviços

P. A. P. Mulotto *G. C. Pupio*

Relatório Técnico - IC-PFG-19-62
Projeto Final de Graduação
2019 - Dezembro

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

UNIVERSIDADE ESTADUAL DE CAMPINAS
FACULDADE DE ENGENHARIA ELÉTRICA E DE COMPUTAÇÃO

Trabalho de Conclusão de Curso

Testes de Robustez em Sistemas Baseados em Microsserviços

Trabalho de Conclusão de Graduação apresentado à Faculdade de Engenharia Elétrica e de Computação e ao Instituto da Computação da Universidade Estadual de Campinas como parte dos requisitos exigidos para obtenção do título de Engenheiro nos cursos de Engenharia Elétrica e Engenharia da Computação.

Autor: Paulo Mulotto
Autor: Guilherme Pupio
Orientadora: Profª. Dra. Eliane Martins

CAMPINAS
2019

Dedicatória

Dedicamos esta tese de graduação aos nossos pais Kédima Pupio, Noel Pupio, Pedro Mulotto e Márcia Mulotto que sempre acreditaram em nós e estiveram ao nosso lado em todos os momentos durante estes anos de graduação.

Agradecimentos

À nossa família, que nos apoiou incondicionalmente e sempre acreditou em nosso potencial

À nossa orientadora, Profa. Dra. Eliane Martins, por ter compreendido nossa agenda apertada e ter colaborado com os horários das reuniões no decorrer do semestre

E a todos que tenham de alguma forma contribuído para que este sonho fosse possível de ser realizado.

Resumo

Utilizando um sistema que conta com a integração de componentes de software como *Kafka e Kubernetes, MySql, Python e Java*, iremos realizar testes de estresse buscando entender o comportamento da aplicação composta por essas ferramentas em determinadas situações de estresse.

Para realizar tais testes, iremos utilizar um toolkit chamado *Chaos Toolkit*. Através desta ferramenta, iremos analisar o estado saudável do sistema e durante os cenários de estresse para poder identificar possíveis falhas.

Entendemos que mesmo utilizando componentes resilientes e bastante utilizados por toda a comunidade, o sistema em si ainda é passível de falhas e assim observamos através dos testes, que podem ocorrer problemas em um sistema que serão somente detectados em situações de falhas e que irão comprometer a experiência do usuário.

Abstract

Using a system that integrates software components such as *Kafka and Kubernetes, MySql, Python* and *Java*, we will perform stress tests to understand the behavior of the application composed by these tools in certain stress situations.

To perform such tests, we will use a toolkit called *Chaos Toolkit*. Through this tool, we will analyze the healthy state of the system and during stress scenarios in order to identify possible failures.

We understand that even using resilient components widely used throughout the community, the system itself is still prone to failure and so we have seen through testing that problems can occur in a system that will only be detected in failure situations and will compromise user experience.

Lista de Ilustrações

3.1 Exemplo para construção de um experimento de chaos 22
4.1 Exemplo para construção de um experimento de chaos 23
4.2 Fluxo com *Apache Kafka* 24
4.3 Tópicos no *Apache Kafka* 25
4.4 Cenário em evolução das plataformas de gerenciamento em nuvem 29
4.5 Diagrama de um *POD* 31
4.6 Ilustração do Replication Controller 32
4.7 Ilustração da camada "*service*" 33
4.8 infraestrutura horizontal dos Kubernetes 35
5.1 *PODs* configurados no *Kubernetes* 37
5.2 Terminal com a aplicação do *Kafka* e Zookeeper em execução 38
5.3 Exemplo de armazenamento Mysql 38
5.4 Resultado para Perda de Pacote 39
5.5 Taxa de respostas bem sucedidas para Perda de Pacote 40
5.6 Resultado para Perda de Pacote 41
5.7 Taxa de Sucesso para Perda de Pacote 41
5.8 Tabela com informações pós teste 42
5.9 Resultados Consolidados 43
A.1 Arquitetura local para testes 51
A.2 *script* para Instalação do *Docker* e *Kubernetes* 52
A.3 Configuração do *cluster Kubernetes* 53
A.4 Query Criação Banco Twitter 60

Lista de Tabelas

5.1 Tempo médio de resposta para determinada injeção de perda de pacote. . . 39
5.2 Taxa de respostas bem sucedidas para determinada injeção de perda de pacote. 40
5.3 Tempo médio de resposta para determinada injeção de latência na rede. . . 40
5.4 Taxa de respostas bem sucedidas para determinada injeção de latência. . . 41
5.5 Consolidado dos Resultados Apresentados. 42
A.1 Especificações das instâncias criadas no Google Cloud 51

Sumário

1 INTRODUÇÃO **11**

2 ORGANIZAÇÃO DAS ATIVIDADES **13**
2.1 Guilherme Pupio 13
2.2 Paulo Mulotto 13
2.3 Comum 14

3 CONTEXTUALIZAÇÃO TEÓRICA **15**
3.1 Testes 15
3.1.1 Teste de Robustez 15
3.1.2 *Chaos Engineering* 16
3.2 *Chaos Toolkit* 18

4 TECNOLOGIAS UTILIZADAS **23**
4.1 Kafka 23
4.1.1 Producers 25
4.1.2 *Consumers* 26
4.1.3 *Zookeeper* 27
4.2 Container 27
4.3 *Dockers* 28
4.4 Kubernetes 28
4.4.1 Principais Definições 29
Kubelet 29
Kubectl 29
4.4.2 *PODs* 30
4.4.3 Serviços 31
4.4.4 Nodes 32
4.4.5 Cluster 33
4.4.6 Rede de Cluster 34
4.4.7 Modelo de Rede Kubernetes 34

5 RESULTADOS **37**

6 CONCLUSÃO **45**

Referências Bibliográficas **47**

APÊNDICES **50**

Apêndices **50**

Apêndice A Configuração do Ambiente **51**
A.1 Kubernetes 52
A.2 Redis 54
A.3 Kafka 55
A.3.1 Nodes 55
A.3.2 Tópicos 57
A.3.3 Producer 57
A.4 Consumers 58
A.5 Criando um projeto genérico 58
A.6 Criando o tópico para o Twitter 59
A.7 Mysql 59

1 INTRODUÇÃO

Atualmente, o ciclo de entrega de software está ficando cada vez mais curto, enquanto, por outro lado, o tamanho e a complexidade das aplicações está ficando cada vez maior. Embora as aplicações estejam cada vez maiores, existe uma forte onda de desenvolvimento que busca cada vez mais tornar sistemas mais micros independentes uns dos outros. Dado o contexto do uso de aplicações estruturadas sobre sistemas distribuídos, como arquiteturas de micro serviços, surgiu o desafio de atuar sobre a grande quantidade de containers, máquinas virtuais e até mesmo máquinas físicas, que são utilizadas nos ambientes de desenvolvimento e de produção das aplicações atuais.

Com a grande popularidade das metodologias ágeis como scrum e metodologia Lean, as indústrias de TI passaram a utilizar o conceito de DevOps[Hüttermann, 2012], que é baseado no uso de práticas que automatizam os processos entre o desenvolvimento de uma aplicação e as equipes de TI. É sobre contexto que o *Kubernetes* vem atuar. Iremos abordar a comunicação entre peças importantes no mercado de hoje como *Kafka*, *Kubernetes* e *MySql*, assim como aplicações *Java* e *Python* executando. Conectando as peças iremos realizar casos de testes com objetivo de encontrar e entender falhas simulando um ambiente produtivo.

O Texto está organizado da seguinte forma: A seção 3, Contextualização Teórica, apresentamos uma introdução aos elementos que irão compor esse trabalho, visando esclarecer fornecer um conteúdo mais especializado dos principais pontos do que serão abordados. A seção 4 Configuração do Ambiente, irá exemplificar como o ambiente foi configurado, quais os comandos utilizados para a configuração, buscando facilitar futuros testes com o trabalho que está sendo produzido. Enquanto, na seção 5, Resultados, serão apresentados os resultados obtidos durante os testes realizados sobre a implementação do ambiente. Por fim, na seção 6, Conclusão, será realizado uma análise conclusiva a partir dos resultados obtidos e sobre os resultados esperados, bem como indicações para trabalhos futuros.

2 ORGANIZAÇÃO DAS ATIVIDADES

Como este trabalho foi realizado em um grupo de dois alunos, faz-se necessária a descrição da divisão de atividades realizadas durante o desenvolvimento do trabalho.

2.1 Guilherme Pupio

O aluno Guilherme Pupio (Ra: 168958), foi responsável pelo estudo das seguintes tecnologias:

1. Containers (seção 4.2)
2. Dockers (seção 4.3)
3. Redis (Apêndice A.2)
4. Kubernetes (seção 4.4)
5. Criação de Ambiente em *Python* para estudo de *Chaos Engineering* (Apêndice A.1)

O estudo das tecnologias citadas, abordam desde seus conceitos teóricos até às configurações de ambiente e integração das quatro tecnologias com as demais utilizadas no projeto.

2.2 Paulo Mulotto

O aluno Paulo Mulotto (Ra: 175541), foi responsável pelo estudo das seguintes tecnologias:

1. Kafka (seção 4.1)
2. Zookeeper (seção 4.1.3)
3. Chaos Toolkit (seção 3.2)
4. MySql (Apêndice A.7)
5. API Twitter (Apêndice A.6)

O estudo das tecnologias citadas, também abordam desde seus conceitos teóricos até às configurações de ambiente e integração das três tecnologias com as demais utilizadas no projeto.

2.3 Comum

Ambos os alunos tiveram de estudar sobre os conceitos de *Chaos Engineering* e participar da realização dos testes de Latência e Perda de Pacote, que serão abordados na seção 5.

Além disso, ambos os alunos tiveram de solucionar os pequenos problemas de integração entre as tecnologias, para poder montar o ambiente completo, ou seja, com a comunicação entre todas as tecnologias.

Com o objetivo de se aproximar dos processos em *Cloud*, cada uma das tecnologias foram instanciadas no *Google Cloud*, o que também levou a uma curva de aprendizado em cada um dos integrantes.

3 CONTEXTUALIZAÇÃO TEÓRICA

3.1 Testes

O "*happy path*" é um termo utilizado no desenvolvimento de software que se refere à execução de um programa em que as entradas não levam a erros ou a condições de contorno. No decorrer do projeto, ao testar o próprio código é bastante comum que os desenvolvedores acabem concentrando seus esforços pelo "*happy path*", gerando códigos que funcionam para casos comuns, mas que podem deixar a desejar no mundo real.

Sob este contexto, os testes são utilizados para obter informações da qualidade do software de forma eficiente e consistente. Através do desenvolvimento de testes é possível saber se a aplicação está respondendo corretamente em determinadas situações, de forma rápida e precisa. Isso torna a manutenção do sistema mais fácil, e reduz a chance de usuário ser impactado por erros já previstos em um determinado momento.

Existe uma série de testes que podem ser realizados para determinar se: o sistema responde corretamente a entradas (*inputs*) variadas, o desempenho do sistema é adequado, se a usabilidade do sistema é adequada, entre outros. Cada teste possui um momento e um lugar específico para ser executado para obter uma melhor resultado. Dessa forma é possível descobrir falhas, de forma que estas possam ser corrigidas e não causem impacto no usuário final. Um estudo recente indicou que cerca de 92% das falhas catastróficas em sistemas, foram resultado do tratamento incorreto de erros não fatais[Ding Yuan, 2014].

Analogamente, ao entrar no contexto de sistemas distribuídos, para executar testes em micro serviços, é necessário uma atenção especial. Além de seu funcionamento correto é preciso garantir que a forma que a aplicação interage com outros serviços continue funcionando da maneira esperada. Isso significa que para micro serviços o foco dos testes deve ser mantido na integração.

A seção seguinte traz uma breve abordagem sobre alguns tipos de testes.

3.1.1 Teste de Robustez

Robustez é um atributo que parametriza o bom funcionamento de sistema sob condições de estresse de ambiente e sob injeção de entradas diversas. O objetivo dos testes de robustez é avaliar como o sistema se comporta nessas situações adversas. Para os testes de robustez são necessários dois tipos de entradas: *workload* (carga de trabalho), que exercita as funcionalidades do sistema, e o *faultload* (carga de falhas), que introduzem dados inválidos[Regina Moraes, 2014].

Um das formas de realizar esse tipo de teste é através da interface. Por meio da

interface pública da aplicação, é realizado um bombardeio de entradas válidas e inválidas. Para realizar tarefas como essas, tem-se como exemplos algumas ferramentas como *Fuzz*[Fuz,], *Ballista*[Bal,] e *JCrasher*[JCr,].

O *Fuzz* gera entradas com strings randômicas com o intuito de sobrecarregar as aplicações. Dado que o input é randômico, não é necessário nenhum modelo do comportamento do sistema ou do tipo da aplicação, possibilitando sua utilização de maneira genérica. *Fuzz* é um tipo de teste de caixa preta (*black box testing*), pois as entradas são escolhidas levando-se em conta unicamente a interface do sistema, não necessitando de conhecimento sobre sua estrutura interna e nem do código fonte. O critério de avaliação é simples, se o sistema não apresentar falha abrupta, está aprovado.

Ballista busca analisar o quão robusta é a aplicação e o Sistema Operacional (SO) em tratamento de exceções. Um sistema robusto não depende apenas do *software* produzido, mas também do SO e de pacotes de terceiros. Assim, *Ballista* provoca falhas no SO, de forma a causar, no *software*, fins de execuções anormais, o que pode acarretar em um erro no sistema. Assim como no *Fuzz*, *Ballista* também não precisa do código fonte ou de especificações de comportamento do sistema. Os testes são criados baseados nos tipos dos parâmetros em vez de se basearem na funcionalidade dos módulos. *JCrasher*, possui um funcionamento muito semelhante ao *Fuzz*, contudo é voltado para o desenvolvimento em Java. Analogamente ao *Fuzz*, o *JCrasher* examina as informações de tipo de um conjunto de classes Java e constrói fragmentos de código que criarão instâncias de tipos diferentes para testar o comportamento de métodos públicos sob dados aleatórios. O *JCrasher* tenta detectar erros, causando o "travamento"do programa em teste - para lançar uma exceção de tempo de execução não declarada.

Testes de robustez podem ser utilizados para se criar testes padronizados para validar dependability. Nesse caso, o *workload* é formado por testes de estresse no sistema de arquivos (*file system*) ou na rede, para dessa forma observar qual *workload* está próximo da saturação do sistema. Nessas condições, além da criação do *workload*, também é criado um fault load, com falhas típicas de hardware ou *software*. Com esse método também é possível determinar o quanto de concorrência de aplicações e clientes o servidor é capaz de suportar.

3.1.2 *Chaos Engineering*

Por fim, ao trabalhar com sistemas distribuídos no ambiente de produção, naturalmente, a aplicação fica sujeita a ocorrência de eventos imprevisíveis provenientes de diversas fontes distintas, uma vez que quanto maior a quantidade de componentes no sistema, maior a fonte de falhas.

Essas falhas podem ser de caráter geral, como por exemplo, falhas em discos rígidos, falha na rede, aumento repentino de tráfego podendo sobrecarregar um componente funcional entre outras possíveis falhas. Com muita frequência, esses eventos desencadeiam interrupções nos serviços, baixo desempenho e outros comportamentos indesejáveis. Assim como em outros testes, *Chaos Engineering* também está confinado ao ato de executar experimentos, segundo *principleofchaos.org* [pri,] "*Chaos Engineering* é a disciplina de realizar experimentos sobre sistemas distribuídos com o intuito de construir confiança com relação a capacidade de um sistema distribuído suportar condições adversas em produção"[pri,].

A ideia fundamental que guia o *Chaos Engineering* é a premissa que os desenvolvedores analisam cada um dos serviços executados em produção como um único e complexo sistema, onde é possível compreender melhor o comportamento do sistema ao injetar, ou simular, *inputs*, que provocam pequenas falhas no sistema e, então, observar o que ocorre com os limites do sistema.

Os testes baseados em *Chaos Engineering* são focados em primeiramente caracterizar o estado saudável que é visível nas interfaces do sistema, onde é analisado diretamente uma interação entre os usuários e o sistema.

Dessa maneira, os engenheiros observam o sistema como uma única entidade, notando o comportamento geral a partir de métricas definidas de acordo com o contexto da aplicação e, então, focando na dinâmica do sistema, ou seja, como o sistema como um todo se comporta para o usuário. O foco principal é o comportamento típico das métricas ao longo do tempo, ou seja, a métrica fundamental é o comportamento saudável do sistema.

É interessante observar que o comportamento do sistema pode ser alterado tanto a partir de interações dos usuários, como por exemplo, a taxa de requisições, tipos de requisições etc, quanto por falhas em serviços terceiros que fornecem informações importantes para o sistema em estudo. O *Chaos Engineering* possui quatro princípios básicos[Ali Basiri, 2016], comentados a seguir:

1. Construção de hipóteses a partir do comportamento saudável (esperado) do sistema.
2. Buscar variáveis que possam ser estimuladas ou simuladas a partir dos diversos eventos do mundo real.
3. Executar experimentos no ambiente de produção.
4. Gerar experimentos automáticos para serem executados continuamente.

Embora o termo *"Chaos"* transmita uma sensação de imprevisibilidade, uma das

principais suposições de *Chaos Engineering* é que um sistema complexo demonstra comportamentos que são regulares a ponto de poderem ser previstos. Dessa forma, é possível gerar hipóteses sobre quais comportamentos devem ser considerados métricas que caracterizam o comportamento esperado do sistema.

No segundo passo deve-se determinar as variáveis que serão estimuladas durante os testes. Para isso pode-se utilizar entradas que potencialmente podem afetar a "saúde" (bom funcionamento) do sistema. Para determinar estas entradas pode-se analisar logs de falhas para determinar as entradas que estavam envolvidas em falhas anteriores do sistema, assim há garantias que causa de problemas passados não serão causa de problemas futuros.

O próximo passo consiste em executar os experimentos, ou seja, aplicar os testes. ma vez que *Chaos Engineering* realiza testes em ambiente de produção, faz se necessário o uso de estratégias para reduzir o risco, como por exemplo reduzir o escopo do experimento para um pequeno subconjunto controlado de usuários, pois assim é possível reproduzir todos os aspectos do sistema em um contexto de teste.

Assim, com o intuito de manter a confiança dos resultados ao longo do tempo, o último princípio do *Chaos Engineering* alerta que os testes devem ser automatizados, uma vez que é necessário executá-los com uma certa frequência, de acordo com o contexto da aplicação.

Por fim, tendo estes princípios em mente, é possível separar este tipo de teste em quatro principais passos, onde primeiramente é necessário definir um comportamento estável, a partir de alguma saída que possa medir o comportamento esperado do sistema. Em seguida, criar dois grupos, onde um contemplará os testes enquanto o outro utiliza o sistema normalmente, partindo da hipótese de que ambos os grupos continuaram com o sistema saudável. O terceiro passo requer a introdução ou simulação de variáveis que refletem eventos do mundo real, como por exemplo queda de servidores, mal funcionamento em discos rígido, entre outras falhas, e por último analisar a métrica definida em ambos os grupos, e então, buscar por diferenças, tentando quebrar a hipótese gerada no segundo passo.

3.2 *Chaos Toolkit*

Com o intuito de realizar *Chaos Engineering*, foi estudado a plataforma *Chaos Toolkit*, que é uma plataforma gratuita, *open source*, e projetada para permitir que desenvolvedores possam criar e aplicar experimentos "caóticos", para descobrir e eventualmente corrigir defeitos encontrados tanto nos níveis de aplicação, quanto na plataforma e infraestrutura do sistema.

Chaos Toolkit CLI, é um programa *Python* que é executado no terminal do **desen-**

volvedor do teste, sendo instalado por meio do *Python Package Installer* (*PIP*) [pip,] e executado pelo comando (**chaos**). Uma vez instalado pode receber vários sub-comandos para realizar diversas tarefas. Para o uso do *Chaos Toolkit* é necessário o interpretador *Python* 3 para que a ferramenta possa ser executada, contudo esta condição é um requisito de praticamente todo sistema *UNIX*.

A ferramenta *Chaos Toolkit* se apropria de várias extensões para conduzir experimentos sobre as diversas tecnologias e camadas em um sistema típico. As integrações são instaladas no mesmo ambiente *Python* que o próprio comando caos, por *default*.

Todo experimento do *Chaos Toolkit* precisa de um método, que é a lista de ações que você deseja executar em seu sistema antes de verificar novamente se a Hipótese do Estado Estável está dentro de suas tolerâncias.

Além de executar ações em relação ao seu sistema, também é possível especificar análises para capturar pontos de dados adicionais durante o experimento para auxiliar a explorar ainda mais as condições presentes no sistema durante a execução.

Esta ferramenta possui também um comando que pode ser usado para executar um experimento definido em **JSON**, conforme ilustrado no exemplo a seguir:

- "*version*": Representa a versão do arquivo de teste
- "*title*": Título do experimento
- "*description*": Uma breve descrição geral da atuação do teste
- "*steady-state-hypothesis*": chave primária para o *setup* da hipótese do teste
  - "*title*": Título da hipótese
  - "*probe*": Lista de testes a serem executados, a fim de verificar a hipótese.
    - "*type*": Define teste como "*probe*"
    - "*name*": Nome do teste executado
    - "*tolerance*": Faixa de resultado saudável ao executar o teste
    - "*provider*": Define a ação tomada pelo teste de hipótese
      - "*type*": Define ação como um processo
      - "*path*": comando a ser executado no bash
      - "*arguments*": argumentos do comando anterior
    - "*pauses*": Define tempo de espera após a ação tomada
  - "method": Método que injeta corrupções no sistema
    - "*type*": Define método como uma ação
    - "*name*": Nome da ação de injeção
    - "*provider*": Define a ação tomada pelo método injetor . Análogo ao *provider* da chave "*probe*"
    - "*pauses*": análogo ao da chave "*probe*"
  - "rollbacks": Aplicado ao fim do teste, caso ocorra sucesso ou falha durante o teste.
    - "*type*": Define rollback como uma ação
    - "*name*": Define nome do rollback
    - "*provider*": análogo à chave "*provider*".

Vale notar que o fluxo do teste é primeiramente determinar se as hipóteses sobre o comportamento esperado do sistema foram satisfeitas em execução sem falhas. Caso sejam, o experimento aplica o método que causa a corrupção definida e, então, volta a verificar a hipótese definida. Caso ocorra alguma falha, ou seja, a hipótese é violada, o

usuário é notificado da falha. Terminada a execução de um experimento é aplicado um rollback no sistema, para remover as falhas injetadas no ambiente da aplicação.

A figura 3.1 mostra um exemplo de experimento que parte da hipótese de que, se houver uma falha em um serviço fornecedor, o serviço consumidor continuará em funcionamento.

Figura 3.1: Exemplo para construção de um experimento de chaos

4 TECNOLOGIAS UTILIZADAS

4.1 Kafka

Atualmente, devido a complexidade de processos de TI onde um sistema origina informações que são necessárias para vários outros sistemas como BI [BI,], monitoração, auditoria entre outros, surge um cenário em que vários sistemas precisam comunicar-se entre si.

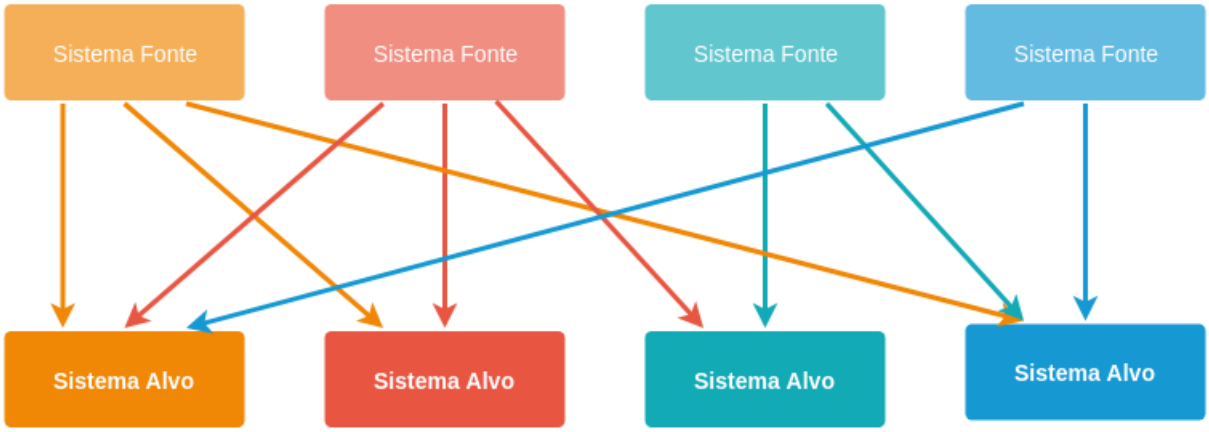

Figura 4.1: Exemplo para construção de um experimento de chaos

Dessa forma, surgem diversas dificuldades para a manutenção do ambiente. No caso da figura 4.1, existem 10 integrações. Cada uma das integrações possui seus próprios obstáculos como, por exemplo, o protocolo de comunicação (**TCP**, **HTTP**, **REST**, **FTP**), a forma na qual o dado obtido é analisado (***binary***, **CSV**, **JSON**); evolução dos esquemas e modelos das informações, entre outros desafios.

Com o intuito de enfrentar essas dificuldades, o Apache Kafka[8] permite desacoplar o compartilhamento de mensagens entre os sistemas de origem e de destino. O apache Kafka também possui a capacidade de realizar streaming processing, ou seja, enriquecer e manipular os dados em tempo real, guardar registros em caso de falhas e integrações com outras ferramentas de big data como Spark, Storm e Hadoop[21].

Com o intuito de enfrentar essas dificuldades, o *Apache Kafka*[THEIN, 2016] permite desacoplar o compartilhamento de mensagens entre os sistemas de origem e de destino. O apache Kafka também possui a capacidade de realizar *streaming processing*, ou seja, enriquecer e manipular os dados em tempo real, guardar registros em caso de falhas e integrações com outras ferramentas de *big data* como *Spark*, *Storm*, *Hadoop*[had,].

A plataforma *Apache Kafka* é *open-source* e foi desenvolvida pela *Apache Software Foundation*, escrita em *Scala* e *Java*. É uma plataforma distribuída, resiliente e com tolerância a falhas. Além disso, é escalável e possui uma alta performance. O *Apache Kafka* é utilizado como uma ponte de dados real-time permitindo o fluxo de dados entre

vários sistemas. Para realizar tal função, é executado em um *cluster* de um ou mais servidores, separando os registros em categorias chamadas **tópicos** e cada registro possui uma **chave**, um **valor** e um **timestamp**.

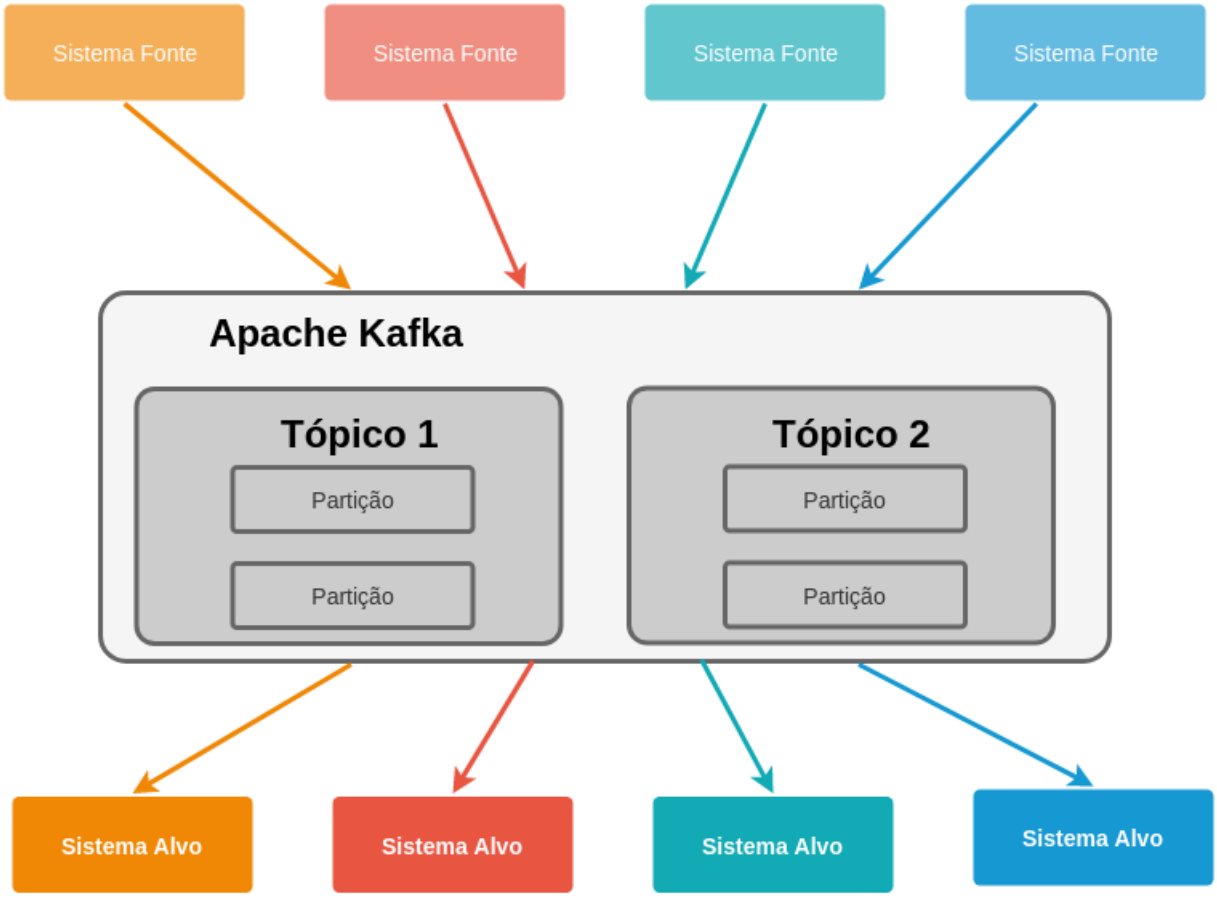

Figura 4.2: Fluxo com *Apache Kafka*

Um tópico é um fluxo particular de dados. Os tópicos são identificados por um nome e podem ser criados em grande volume caso necessário. Eles são divididos em partições, conforme ilustra a figura 4.2, sempre começando pela partição 0. Cada partição é ordenada e recebe mensagens identificadas com mensagens identificadas por um **ID** incremental, chamado *offset*. Além disso, as partições são independentes e podem receber um número diferente de mensagens.

A ordem das mensagens recebidas pelo *Apache Kafka* é garantida apenas em uma partição. Podemos garantir que mensagens, de uma mesma partição sejam escritas de acordo com seus *offsets*, respeitando uma ordem crescente de prioridade, entretanto essa relação não é garantia entre mensagens de diferentes partições. Além disso, as informações são mantidas por um determinando tempo e uma vez que o dado é escrito na partição, ele passa a ser imutável, ou seja, não pode ser alterado. A figura 4.3 ilustra um pequeno esquemático dos tópicos do *Apache Kafka*.

Um *cluster Apache Kafka* é composto por *acknowledgment*. Cada *broker* é identifi-

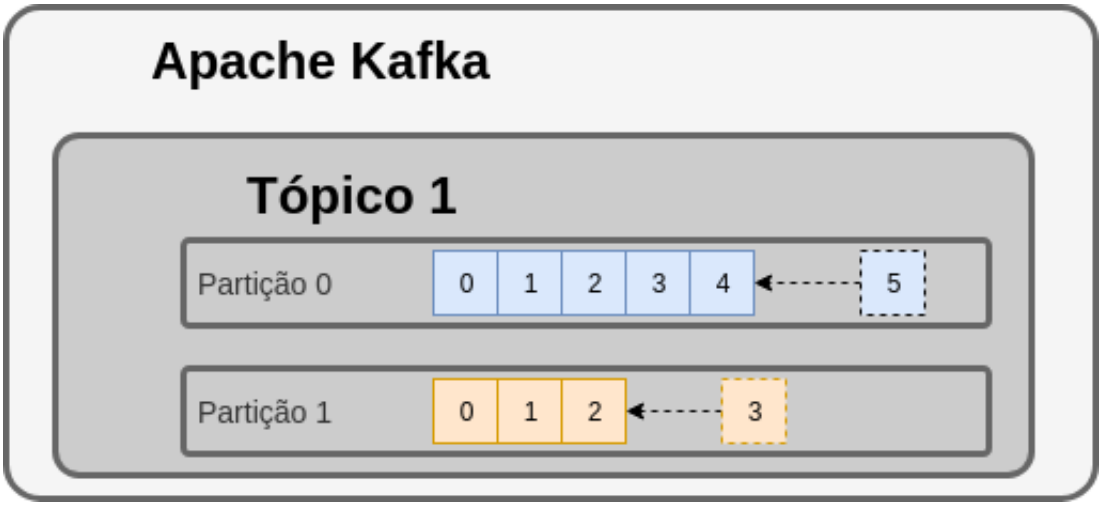

Figura 4.3: Tópicos no *Apache Kafka* ordenados e com *offsets* incrementais

cado com seu **ID** e contém um certo número de partições de tópicos. Um *broker* é a porta de entrada para o *cluster*, ou seja, ao se conectar a qualquer *broker*, você é conectado a todo o *cluster Kafka*.

As partições dos tópicos são distribuídas através do *acknowledgment* automaticamente. Além disso, há o fator de replicação de *Nodes*. Dessa forma, caso ocorra alguma falha, por motivos quaisquer, em um *Node* do *cluster*, outro *Node* pode continuar com o envio de informações.

Dentro de uma partição, há um mecanismo de hierarquia, onde apenas um *broker* pode ser o líder de uma partição durante um determinado momento, assim, apenas um *broker* pode receber e enviar dados para a partição. Enquanto isso, os outros *acknowledgment* irão apenas sincronizar a informação e, caso ocorra alguma falha no líder, o *Node* sincronizado assume este papel, evitando indisponibilidade.

4.1.1 Producers

Producers são utilizados para os dados nos tópicos. Para enviar dados aos tópicos, não é necessário especificar em qual *broker* ou partição um *producer* deve escrever, pois o destino do dado escrito é reconhecido automaticamente. Em caso de falhas do *broker*, os producers se recuperam automaticamente, de forma transparente para o desenvolvedor.

É possível especificar o *acknowledgment* de escrita de dados, que evita a perda de dados caso a mensagem se perca. Tal se dá por meio da variável **acks**, permitindo verificar se a mensagem não se perdeu durante a comunicação, garantindo sua entrega. Caso acks = 0, Producer não espera nenhum reconhecimento de recebimento de dado e, assim, é possível que ocorra perda de dados. Caso acks = 1 Producer irá aguardar o reconhecimento do líder, limitando as perdas. Já no caso de acks=all, o Producer irá

esperar o reconhecimento do líder e das réplicas, evitando perda de informações.

Producers podem enviar uma chave, que pode ser uma *string* ou um número, com a mensagem. Caso a chave seja nula, é realizado um sorteio round robin para determinar para qual *broker* o dado será enviado. Se a chave é enviada, então todas as mensagens para aquela chave serão enviadas para a mesma partição. Essa *feature* é importante para garantir que as informações de um mesmo elemento estejam em um mesma partição, e dessa forma, ordenadas de acordo com a ordem de chegada das mensagens.

4.1.2 *Consumers*

Consumers são responsáveis pela leitura das informações de um tópico. Através de uma identificação via nome os *Consumers* conseguem identificar automaticamente qual dos tópicos é o seu fornecedor e, então, consumir as informações dos *acknowledgment*. No caso de falhas, o Consumer se recupera automaticamente, sem a necessidade de intervenções de um programador. Além disso, em cada partição, os dados são lidos em ordem de chegada.

Um único Consumer pode consumir múltiplos *acknowledgment* em paralelo. Nesse caso, as mensagens em cada partição continuam sendo lidas em ordem, conforme foram recebidas, contudo não é garantido a sequência das mensagens entre as partições diferentes. Caso existam mais *Consumers* do que partições, alguns deles ficarão inativos. *Consumers* inativos são importantes para garantir que caso um dos consumers ativos falhem, os inativos assumam o papel dos que falharam, fazendo com que a aplicação possa continuar desempenhando seu objetivo.

O *Kafka* armazena os *offsets*, que são registros do último **ID** recebido, de cada grupo de consumidores em operação. Quando um consumidor, em um grupo, processou o dado recebido do *Kafka*, ele deve *commitar* o *offset*. Isto é importante para controle em caso de falhas do sistema, pois dessa maneira fica registrada a última informação recebida pelo consumidor e, então, é possível dar continuidade de leitura a partir do último *offset* registrado.

Consumidores podem escolher 3 formas de ***commitar*** o *offset*. O *commit* pode ser realizado, por exemplo, assim que a mensagem é recebida. Nesse caso, se o processo falha, a mensagem é perdida e não pode ser lida novamente. No segundo cenário, é realizado o *commit* somente após a mensagem ser processada, assim, se algo der errado no processo, a mensagem pode ser lida novamente. Por último, a terceira forma de *commit* é indicado apenas em casos particulares como, por exemplo, no *Kafka Streams API* e é realizado apenas uma única vez.

Apesar da segunda forma ser a mais indicada, pois não há perda de dados, a men-

sagem pode ser processada múltiplas vezes, o que pode gerar impacto no sistema. Para um uso adequado desta forma, é necessário garantir que o processo seja idempotente, ou seja, a mensagem ser processada várias vezes não irá impactar o sistema.

4.1.3 *Zookeeper*

Zookeeper é um gerenciador de *acknowledgment*, mantendo uma lista com cada um deles. Este mecanismo ajuda a eleger o líder na eleição das partições. O *Zookeeper* envia notificações para o *Kafka* caso ocorram mudanças, tais como: o surgimento de um novo tópico, uma falha ou nova inserção de algum *broker*, ou ainda, supressão de tópicos. O *Zookeeper* foi projetado para operar com um número ímpar de *acknowledgment*, onde um é o líder, que trata das escritas, enquanto os outros são seguidores e tratam das leituras.

4.2 Container

Os *containers* são semelhantes às VMs (*Virtual-Machines*), mas possuem propriedades de isolamento relaxadas para compartilhar o Sistema Operacional **SO** entre os aplicativos. Portanto, os *containers* são considerados como VMs leves. Semelhantes a uma VM, um contêiner possui seu próprio sistema de arquivos, CPU, memória, espaço de processo e muito mais. À medida que são dissociados da infraestrutura subjacente, eles são portáveis em nuvens e distribuições de **SO**.

Os *containers* estão se tornando populares por conta dos seus muitos benefícios. Alguns dos benefícios de um *container* são:

1. Criação e implantação de aplicativos ágeis: maior facilidade e eficiência na criação de imagens de contêiner em comparação com o uso de imagens de VM.
2. Desenvolvimento, integração e implantação contínuos: fornece criação e implantação confiável e frequente de imagens de *containers* com reversões rápidas e fáceis (devido à imutabilidade da imagem).
3. Portabilidade de distribuição na nuvem e no SO. É possível executar tanto em *Linux*, *macOSX*, *Windows* e em qualquer outro Sistema Operacional.
4. Ambiente Compacto. Um *container* possui apenas os pacotes e instalações essenciais para sua aplicação.

Com o uso de *containers* é possível construir micro serviços fracamente acoplados, distribuídos, elásticos e liberados, ou seja, os aplicativos são divididos em partes menores e independentes e podem ser implantados e gerenciados dinamicamente - não uma pilha

monolítica em execução em uma grande máquina de uso único.

4.3 *Dockers*

O *Docker* é uma plataforma para que desenvolvedores e administradores de sistemas possam construir, compartilhar e executar aplicativos em contêineres. O uso de contêineres para implantar aplicativos é chamado de conteinerização.

Como analisado na seção 4.2, basicamente, um *container* não passa de um processo em execução, com alguns recursos adicionais de encapsulamento aplicados a ele para mantê-lo isolado do host e dos outros *containers*. Um dos aspectos mais importantes do isolamento de um *container* é que cada contêiner interage com seu próprio sistema de arquivos privado; esse sistema de arquivos é fornecido por uma imagem do *Docker*.

4.4 Kubernetes

O *Kubernetes* é uma ferramenta *open-source* de orquestração desenvolvida pelo *Google* e lançada para a comunidade em 2014. O principal objetivo do *Kubernetes*, também conhecido como **K8s**, é realizar a orquestração de *clusters* de *containers*. O nome *Kubernetes* é originário do grego, significando timoneiro ou piloto, o que faz jus ao seu ícone. Segundo sua própria documentação[kub, a], "O *Kubernetes* se baseia em uma década e meia de experiência que o *Google* tem em executar cargas de trabalho de produção em grande escala, combinadas com as melhores idéias e práticas da comunidade"[kub, d].

Como citado na seção 4.2, os *containers* estão sendo utilizados em diversos estágios no desenvolvimento de aplicações; dessa forma é justificado o fato da grande maioria das empresas continuarem a aumentar o uso do *Kubernetes* para desenvolvimento e testes, pois com o K8s, é possível realizar o gerenciamento de diversos *containers* de maneira simples e eficiente.

Com base em uma pesquisa realizada pela Cloud *Native Computing Foundation* (**CNCF**) [cnc,], é inegável que o *Kubernetes* atingiu um elevado grau de confiança no mercado. A figura 4.4 mostra a preferência do kubernetes frente a outras ferramentas de orquestração de *containers*.

As seções a seguir visam uma abordagem mais profunda sobre alguns termos já mencionados até então, além da apresentação de outros segmentos do *Kubernetes*.

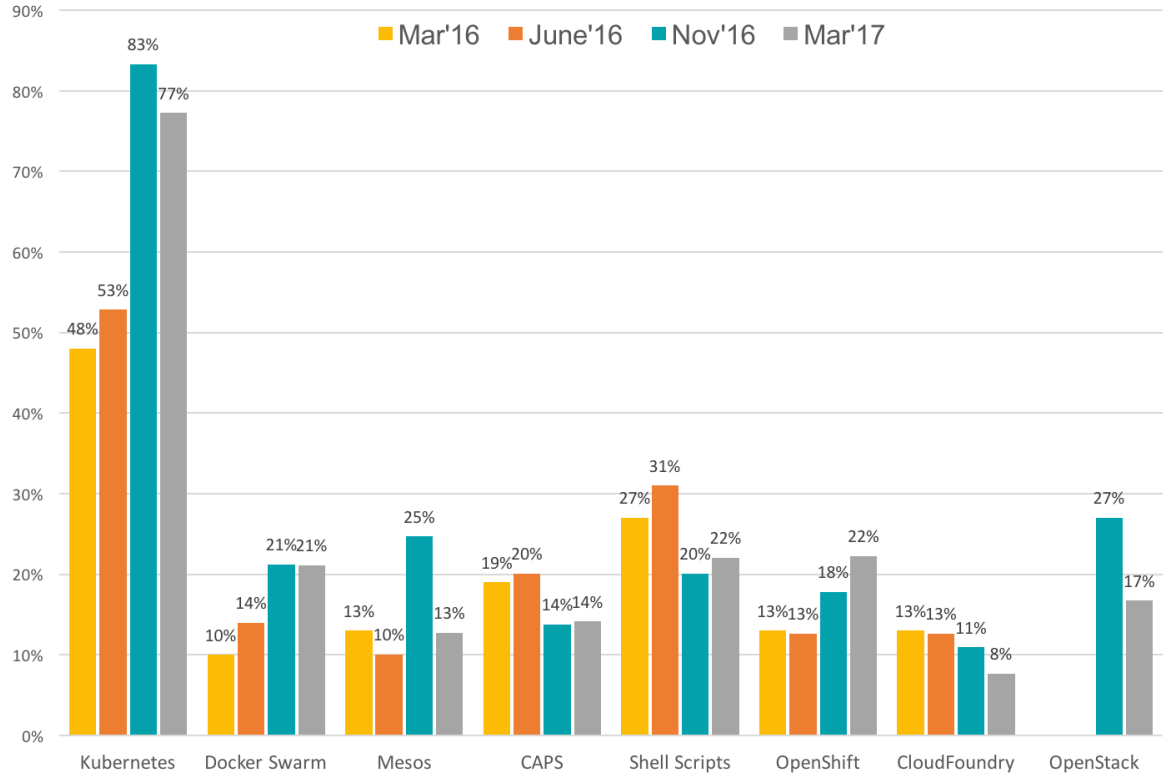

Figura 4.4: O cenário em evolução das plataformas de gerenciamento em nuvem. Preferências das plataformas de gerenciamento de *containers*, referência da imagem em https://www.cncf.io/blog/2017/01/17/container-management-trends-kubernetes-moves-testing-production/

4.4.1 Principais Definições

Kubelet

O *Kubelet* monitora o estado de um *POD*, garantindo que todos os *containers* no *Node* estejam em execução e funcionando em seu estado saudável, i.e., sem falhas. Caso este não esteja no estado desejado ou o próprio *Node* falhar, o *Replication Controller* entra em ação para ativar novos *PODs* íntegros, substituindo aqueles onde foram observadas as falhas.

Kubectl

Um agente que é executado em cada nó no *cluster*. Ele garante que os *containers* estejam sendo executados em um *POD*.

O *Kubelet* utiliza um conjunto de *PodSpecs* que são fornecidos por vários mecanismos e garante que os *containers* descritos nestes *PodSpecs* estejam funcionando corretamente. Uma importante observação é que o *Kubelet* não gerencia *containers* que não foram criados pelo *Kubernetes*.

4.4.2 *PODs*

Cada unidade básica de execução do *Kubernetes* é chamada de um pod. Um pod é a menor e mais simples unidade de computação que pode ser criada e gerenciada no *Kubernetes*. Um *POD* é um bloco contendo um ou mais *containers*, que executam as tarefas implementadas na aplicação.

Para o uso de vários *containers*, um *POD* pode possuir armazenamento compartilhado entre estes *containers* (volumes[vol,]), como discutido na seção de *containers* 4.2. Além disso, um fato importante é o compartilhamento de um mesmo endereço de IP e espaço de portas, fazendo com que os *containers* possam se encontrar via localhost. Os *containers* em diferentes *PODs* têm endereços IP distintos e não podem se comunicar pelo IPC sem serem previamente configurados.

Analogamente a *containers* individuais, os *PODs* são considerados entidades relativamente efêmeras. Ao criar um *POD*, lhe é atribuído um ID exclusivo (UID) e este é alocado para um *Node*, onde permanecem até o seu término ou exclusão, de acordo com a política de reinicialização ou exclusão. Se um *Node* é desativado, as propriedades de container são utilizadas em essência pois, após um período limite, os *PODs* alocados nesse *Node* são agendados para exclusão ao invés de serem realocados para um novo *Node*, uma vez que estes podem ser substituídos por outros *PODs* idênticos, com os respectivos nomes, se desejado, mas com novos UIDs [rep,].

A figura 4.5 apresenta uma pequena ilustração do diagrama de um *POD*.

Para gerenciar estes eventos de substituição *PODs*, que são parte do ciclo de vida dos *PODs*[cic,], o *Kubernetes* usufrui de um "Controlador de Replicação"(*Replication Controller* - ***RC***). Em um ambiente com diversos *PODs* em execução, é o RC quem garante que uma certa quantidade de *PODs* esteja em execução. Se ocorre algum problema com um *POD*, o RC gera um novo para o substituir, buscando garantir que o estado definido seja mantido o mais rápido possível. Caso seja necessário aumentar ou até mesmo diminuir o número de *PODs*, o RC fica responsável por criar ou destruir os

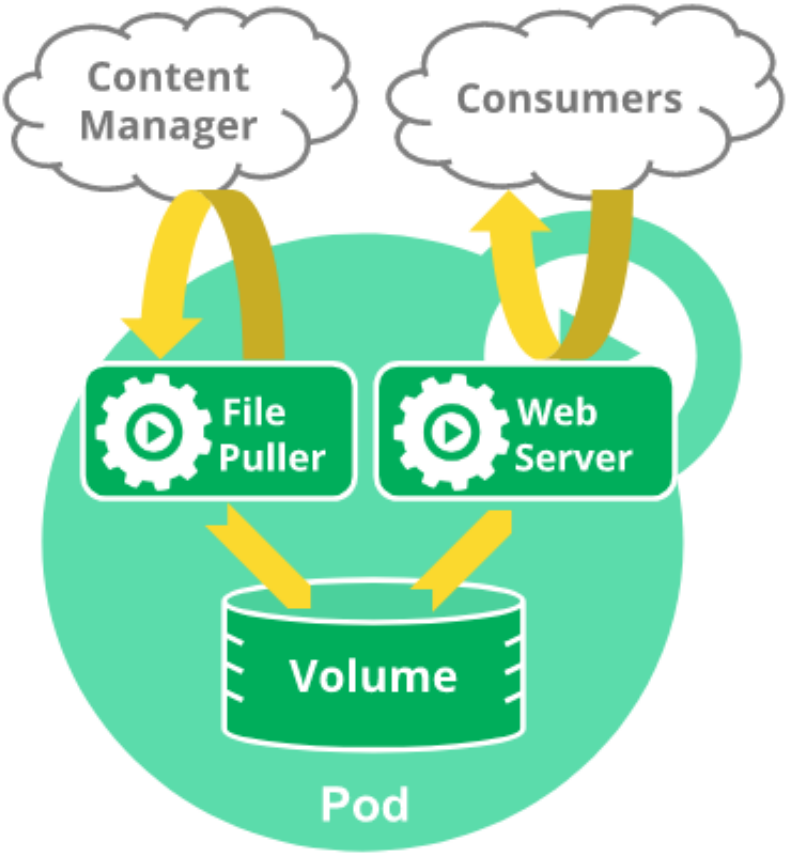

Figura 4.5: Diagrama de um *POD* com vários *containers*, contendo um extrator de arquivos e um servidor da *Web* que usa um volume persistente para armazenamento compartilhado entre os *containers*. Referência da imagem em: https://kubernetes.io/docs/concepts/workloads/pods/pod/

PODs.

Atualmente, existem três tipos de controladores, controlador de replicação , conjunto de réplicas e implantação . Um conjunto de réplicas permite a seleção baseada em conjuntos e a implantação é basicamente um controlador de replicação com recursos para atualizações contínuas. A figura 4.6 ilustra o gerenciamento do Replication Controller.

4.4.3 Serviços

Como mencionado até agora, como os *PODs* são "mortais", ou seja, morrem e não retornam aos seus Nodes. Dessa forma, surge um problema no que diz respeito à comunicação dos *PODs*. Tomando o cenário onde um *POD* com funções *back-end* fornece dados para outro *POD* com atribuições de *front-end*, uma vez que o primeiro *POD* é removido no decorrer da aplicação, quando um novo *POD* de *back-end* ocupar seu lugar,

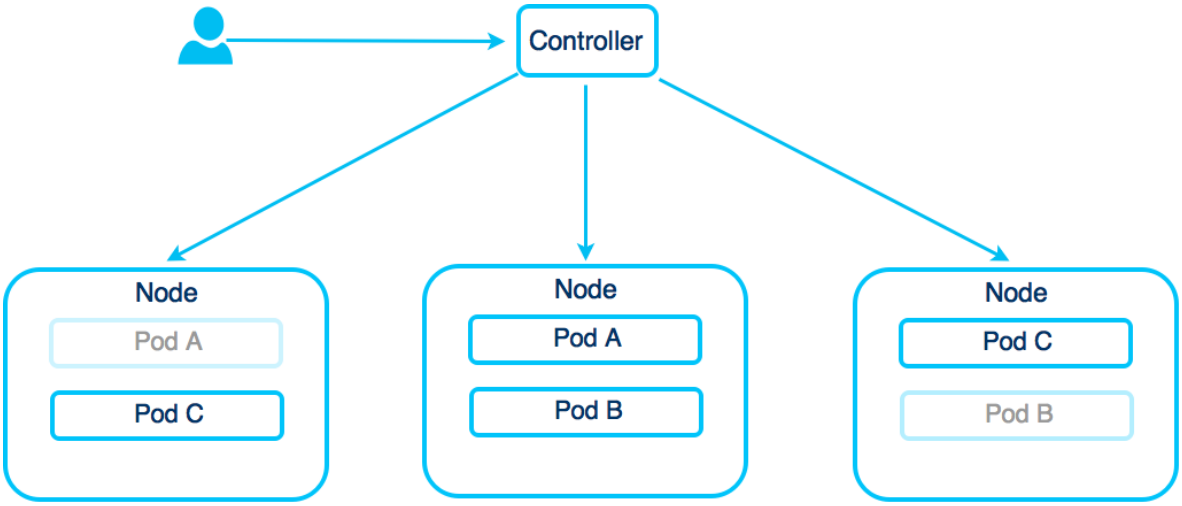

Figura 4.6: Ilustração do Replication Controller. Note que o primeiro e último node, possuem *PODs* mais claros, isso representa a remoção dos *PODs* dos respectivos nods. Abstraindo para um contexto real, um desenvolvedor, pode ter notado que o pod C requer muito mais recursos que os *PODs* A e B, e então, decidiu remover as réplicas dos *PODs* A e B, dessa forma, o Replication Controller atua, garantindo que os demais *PODs* estejam saudáveis. Fonte da imagem: *https://dzone.com/articles/event-driven-microservices-with-vertx-and-kubernet-1*

o *POD* de *front-end* não acompanhará a mudança no endereço IP, e perderá a referência de onde deve se conectar.

No *Kubernetes*, um serviço (*service*) é um agrupamento de *PODs*, que permite o acesso de maneira consistente aos processos servidos por eles. Em outras palavras, um serviço é uma abstração que define um conjunto lógico de *PODs* e uma política pela qual será possível acendê-los. O conjunto de *PODs* segmentados por um Serviço geralmente é determinado por um seletor, que tem um endereço IP definido, e outras partes da aplicação se conectam através dele para chegar nos *PODs*, sem saberem quantos *PODs* existem ou onde eles estão alocados[ser,]. A figura 4.7 ilustra um pequeno exemplo da atuação do *service*, onde fica claro que a dinâmica de réplicas dos *PODs* é transparente para o solicitante de tal serviço.

4.4.4 Nodes

Um *Node*, também conhecido como ***minion***, pode ser tanto uma máquina virtual quanto uma física, dependendo apenas do *cluster* que está envolvido. Cada *Node* contém os serviços necessários para executar *PODs* e é gerenciado por alguns componentes principais, como o *kubelet* tratado na seção . Além do *Kubelet*, existem outros dois serviços em um *Node*, são eles: *runtime* do contêiner, que é responsável pela execução em si do container[con,] e *kube-proxy*, que é um *proxy* de rede com regras que permitem a

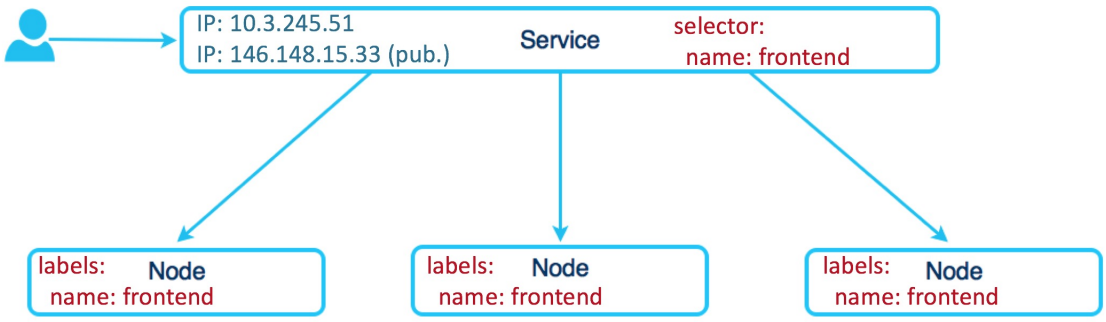

Figura 4.7: Ilustração da camada "*service*". É possível observar, que para o *client* é transparente a quantidade de *PODs* e seus respectivos IPs. Todos são acessados por meio do *service*. Fonte da imagem: *https://dzone.com/articles/event-driven-microservices-with-vertx-and-kubernet-1*

comunicação em rede entre os *PODs* de um *Node*.

O *Kubernetes* possui dois tipos de *Nodes*, os Masters e os "*workers*". Existe uma grande diferença nas funções de ambos. O primeiro é responsável por decidir o que é executado em todos os nodes do *cluster*. Isso pode incluir a programação de cargas de trabalho, como aplicativos em *containers*, e o gerenciamento do ciclo de vida, da escala e dos upgrades das cargas de trabalho. A máquina master também gerencia recursos de rede e de armazenamento para essas cargas de trabalho.

A criação de *PODs* é realizada no *Node* master, contudo ao subir estes *PODs* o *Kubernetes* proporciona um balanceamento automático entre os Nodes, por exemplo, no caso de três nodes, um *master* e dois *slaves* ao realizar o *deploy* de 10 réplicas de um *POD*, automaticamente, haverá 5 réplicas em cada um dos dois *Nodes Slaves*.

4.4.5 Cluster

Um *cluster* consiste em pelo menos um *Node master* e várias máquinas de *worker*, que nada mais são que os Nodes abordados na seção anterior 4.4.4.

Um *cluster* possui o *Node master* como o ponto de extremidade unificado. Todas as interações com o *cluster* são feitas por meio de chamadas da API do *Kubernetes*, e o node master executa o processo do servidor da API do *Kubernetes* para lidar com essas solicitações.

4.4.6 Rede de Cluster

O *Kubernetes* está completamente envolvido no compartilhamento de máquinas por meio de aplicativos. Entretanto garantir o bom funcionamento destas máquinas compartilhadas requer a coordenação de portas entre vários desenvolvedores, que, em escala, possui uma grande dificuldade de ser realizada, pois por exemplo, toda aplicação terá de abordar portas como *flags*, em que os serviços têm de saber como encontrar uns aos outros[red,].

Para evitar expor os usuários a problemas no cluster, o *Kubernetes* usufrui de uma abordagem diferente, denominada de modelo de rede *Kubernetes*, que será abordado na seção seguinte.

4.4.7 Modelo de Rede Kubernetes

A maneira que o *Kubernetes* trata o compartilhamento das máquinas é definindo um endereço de IP para cada *POD*, logo, praticamente não é necessário mapear as portas do container com as portas *host*. Dessa forma, tem-se um modelo limpo, onde cada *POD*, pode ser tratado como uma Máquina Virtual, ou até como mesmo um *host* físico a partir das perspectivas de alocação de portas, nomeação, balanceamento de carga e configuração de aplicativos, entre outras tratativas.

Segundo a Documentação do modelo de rede *Kubernetes*[mod,], são necessários os seguintes requisitos fundamentais a qualquer implementação de rede (exceto as políticas intencionais de segmentação de rede):

- Pods em um *Node* podem se comunicar com todos os *PODs* em todos os *Nodes* sem a necessidade de configurar regras de firewall, como as *NAT* (*Network Address Translation*)[N. Steinleitner, 2006].
- Agentes em um *Node* (por exemplo, daemons do sistema, *Kubelet*) podem se comunicar com todos os *PODs* desse *Node*.
- Pods na rede *host* de um *Node* podem se comunicar com todos os *PODs* em todos os *Nodes* sem *NAT*.

Com esse modelo, além de uma maior simplicidade em relação aos métodos de alocação dinâmica de portas, o *Kubernetes* recebe maior adaptabilidade a projetos, pois possui o mesmo modelo básico de, por exemplo, um projeto executado em Máquinas Virtual com um IP e que podia trocar informações com outras diversas VMs.

Neste trabalho foi utilizado o *Pod NetWork - wave*[net,], responsável por configurar todos os requerimentos de rede para o *Kubernetes*, estabelecendo uma comunicação entre todos os *PODs*, independentemente de seu *Node*.

De maneira geral, o *Kubernetes* trabalha com um conceito, onde não há uma hierarquia de *Nodes*, ou seja, cada um é uma instância capaz de executar suas tarefas através de seus *PODs*. Os *Nodes* são controlados pelo Controlador de *Nodes* (*Node Controller*), o qual é responsável por atribuir tarefas, manter a lista de *Nodes* atualizadas de acordo com a disponibilidade das máquinas, e por fim, monitorar a saúde dos *Nodes*.

O *Kubernetes* possui um escalonamento de infraestrutura horizontal, ou seja, as operações são realizadas no nível do servidor individual. Novos servidores podem ser adicionados ou removidos de maneira simples e ágil. A figura 4.8 ilustra a arquitetura horizontal dos *PODs*.

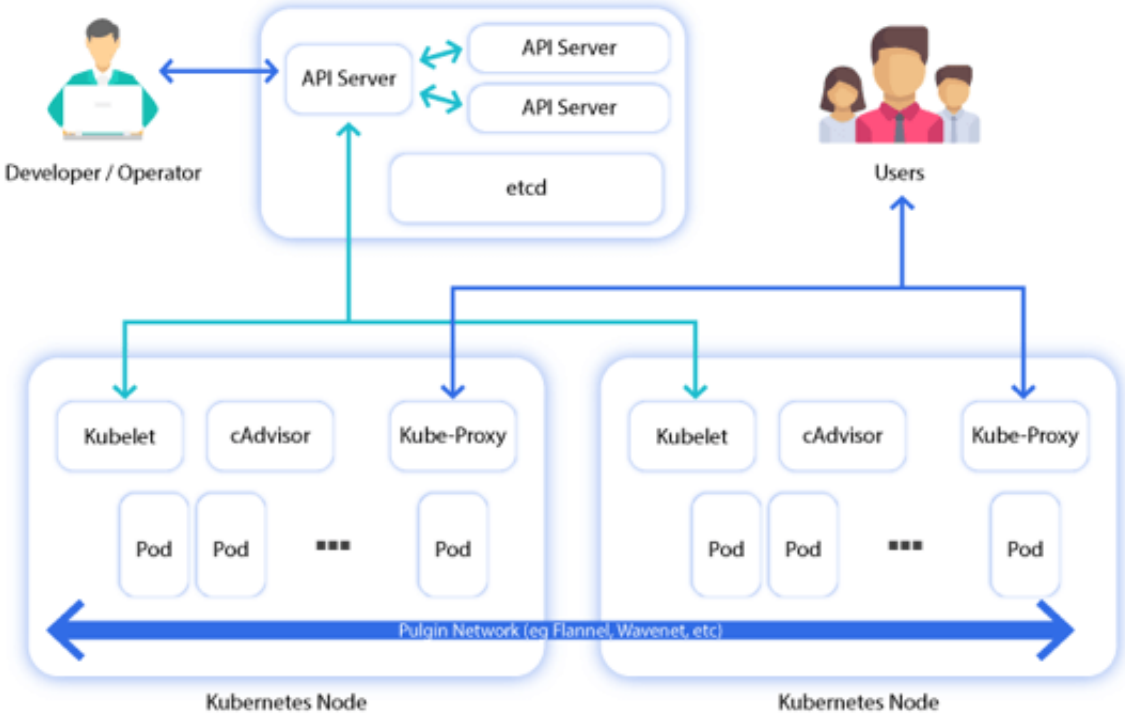

Figura 4.8: Exemplo de infraestrutura horizontal do Kubernetes. Referência da imagem: *https://hackernoon.com/kubernetes-in-10-minutes-a-complete-guide-to-look-for-ad0be0f8a9b8*

5 RESULTADOS

Com o ambiente devidamente configurado, como pode-se observar no apêndice A, tivemos a seguinte configuração dos *PODs*, conforme ilustra a figura 5.2.

```
root@instance-1:~# kubectl get pods -o wide
NAME                                                READY   STATUS    RESTARTS   AGE    IP          NODE
chaos-consumidor-twitches-deployment-7fc5569c87-78v5t   1/1     Running   1          6d2h   10.44.0.6   instance-4
chaos-consumidor-twitches-deployment-7fc5569c87-d7xsz   1/1     Running   1          6d2h   10.40.0.5   instance-3
chaos-consumidor-twitches-deployment-7fc5569c87-nclgb   1/1     Running   1          6d2h   10.44.0.3   instance-4
chaos-consumidor-twitches-deployment-7fc5569c87-r4cc6   1/1     Running   1          6d2h   10.40.0.6   instance-3
consumidor-twitches-deployment-5c89667dbf-g7ws2         1/1     Running   1          6d2h   10.40.0.3   instance-3
consumidor-twitches-deployment-5c89667dbf-kbxfv         1/1     Running   1          6d2h   10.40.0.7   instance-3
consumidor-twitches-deployment-5c89667dbf-l2rh7         1/1     Running   1          6d2h   10.40.0.2   instance-3
consumidor-twitches-deployment-5c89667dbf-ztcrh         1/1     Running   1          6d2h   10.44.0.5   instance-4
redis-deployment-85b48ffb6f-gj574                       1/1     Running   1          8d     10.40.0.4   instance-3
twitter-api-kafka-deployment-6cc84c8dc9-7cg84           1/1     Running   1          6d3h   10.44.0.2   instance-4
```

Figura 5.1: Ilustração do número de *PODs* configurados no *Kubernetes*

A partir da configuração da figura 5.2, pode-se notar que os *PODs* foram distribuídos equilibradamente entre as instâncias 3 e 4, conforme explicado na seção 4.4.

Os quatro primeiros *PODs* são reservados para aplicação dos testes de *Chaos Engineering*, enquanto dos quatro seguintes atuam em produção no sistema. Por fim, os dois últimos contêm, respectivamente, o banco *Redis* e a aplicação integrada com o *Twitter*, responsável por enviar dados para um dos tópicos do *Kafka*.

Note que cada um dos *PODs* representam um serviço, entretanto existem apenas 3 serviços diferentes sendo executados, pois os 8 primeiros *PODs* possuem a mesma imagem *Docker*, ou seja, são *PODs* idênticos. Cada uma das imagens utilizadas para a construção destes *containers* podem ser encontradas no *docker hub* do projeto, ou, podem ser construídas a partir dos *Dockerfiles* presentes no *github do projeto*.

Após o início dos serviços no *Kubernetes* o *Kafka* passou a receber mensagens do *POD* que realiza a comunicação com o *Twitter* e popular o tópico Tweets. Visto que a aplicação que estava extraindo os tweets do *Twitter* monitorava apenas posts com palavras específicas como "*Unicamp*", "*Lula Livre*", "*Eleições*" o fluxo de dados entre os servidores do *Kubernetes* e o *Kafka* era variado.

Em sequência é consumido do tópico do *Kafka* e inserido no *MySql*. Com o transposição das informações do *Kafka* para o *MySql*, é montado um schema para estruturar o dado, conforme a figura A.4 do apendice A. Nesse processo não ocorreram perdas durante a execução dos testes deste trabalho. A partir da figura 5.3 é possível observar como os dados foram armazenados no banco de dados.

Ainda com a figura 5.3, é possível perceber pela diferença entre o horário de criação do *tweet* e o momento em que o dado foi salvo no banco uma pequena variação. Por exemplo, no caso de id = 14, o *tweet* foi postado às 20:57:48 e foi inserido no banco às 20:57:55. Isso mostra que há uma latência no tempo de comunicação entre o *Twitter* e

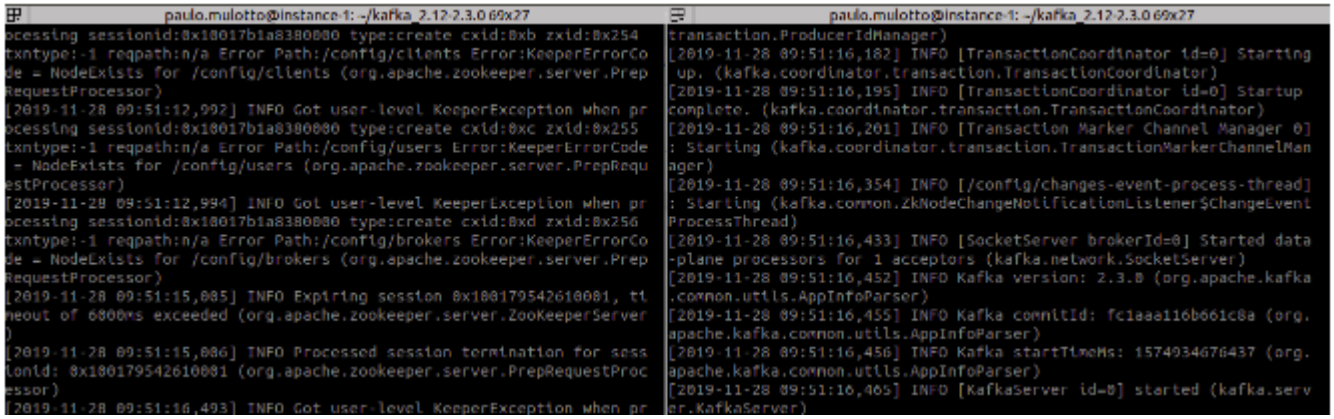

Figura 5.2: Terminal com a aplicação do *Kafka* e Zookeeper em execução

ID	local	created_at	text	reg_date
8	Boa Viagem, Recife	2019-11-10 20:25:42	RT @***: O choro é livre, e o Lula também ?🏴	2019-11-10 20:40:33
9	Brasil	2019-11-10 20:44:07	RT @***: Ordem de soltura de Lula já foi dada O sonho dos petistas de "Lula Livre" virou um pesadelo real para sociedade brasileira e p…	2019-11-10 20:44:15
10	São Paulo, Brazil	2019-11-10 20:44:13	RT @***: Estou em Sorocaba, uma das cidades mais reacionárias e estúpidas do Sudeste brasileiro. Faz dois dias que não durmo. É Bet…	2019-11-10 20:44:20
11	Jardim Gramacho, DC	2019-11-10 20:44:14	RT @***: Gente perdão ainda estar no clima LULA LIVRE mas eu fui uma criança petista e se tem uma coisa pior que petista é o filho…	2019-11-10 20:44:23
12	São Paulo, Brasil	2019-11-10 20:44:17	RT @***: Elio Gaspari: “Moro, o Justiceiro de Curitiba, tornou-se um ministro subserviente e inócuo. Os procuradores da Lava Jato en…	2019-11-10 20:44:25
13	Rio de Janeiro	2019-11-10 20:45:49	RT @***: Atenção: não aceite participar de um site para ganhar camiseta da Campanha Lula Livre. É virus. Querem pegar dados de esque…	2019-11-10 20:45:56
14	Barra Mansa, Brasil	2019-11-10 20:57:48	RT @***: POR FAVOR LULA ! Venha para rua. Você vai ver a popularidade que tem. Lula não anda cercado de povo, só de militantes e…	2019-11-10 20:57:55

Figura 5.3: Tabela com exemplos do formato do dado gravado no *MySql*. *local, created_at, text*, representa o local, o horário e o conteúdo do *tweet* que foi postado. Os valores em *reg_date* representam o momento em que o dado foi inserido no banco de dados.

nossa aplicação.

Em relação aos testes de estresse do ambiente, iniciamos com testes muito agressivos, perdendo a comunicação com as instâncias da Google Cloud. Devido a isso, foi necessário realizar o *tunning* dos testes para não perdermos o acesso e consequentemente

os resultados.

Através dos gráficos representados nas figuras 5.4 e 5.5 é possível analisar que quanto maior a perda de pacotes, mais prejudicada fica a experiência do usuário que está utilizando a aplicação. Entre o melhor e o pior cenário, ocorreu uma piora de 17,87 vezes no tempo de resposta.

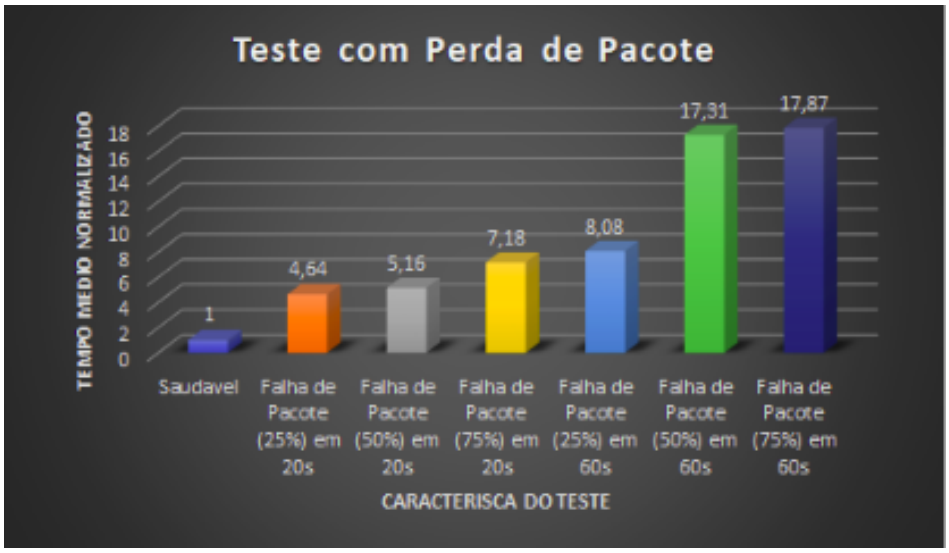

Figura 5.4: Gráfico do tempo médio de resposta para determinada injeção de perda de pacote. Gráfico formado a partir da tabela 5.1

Tabela 5.1: Tempo médio de resposta para determinada injeção de perda de pacote.

Tipo de Injeção	**Tempo Médio Normalizado**
Nenhuma (Saudável)	1
Falha de Pacote (25%) em 20s	4,64
Falha de Pacote (50%) em 20s	5,16
Falha de Pacote (75%) em 20s	7,18
Falha de Pacote (25%) em 60s	8,08
Falha de Pacote (50%) em 60s	17,31
Falha de Pacote (75%) em 60s	17,87

Porém, o que é mais notório é que ocorreu falha de comunicação em que o sistema não foi capaz de se recuperar. Quando inserido regra de perda de 75% de perda de pacotes, 22% das requisições foram perdidas, como ilustrado na figura 5.5, ou seja, retornaram erro 404. Dessa forma, é possível que esse tipo de falha é crítica para o usuário.

Já no caso da inserção de latência na comunicação, também ocorreu um aumento no tempo de resposta, como era esperado. Analogamente ao teste de perda de pacote,

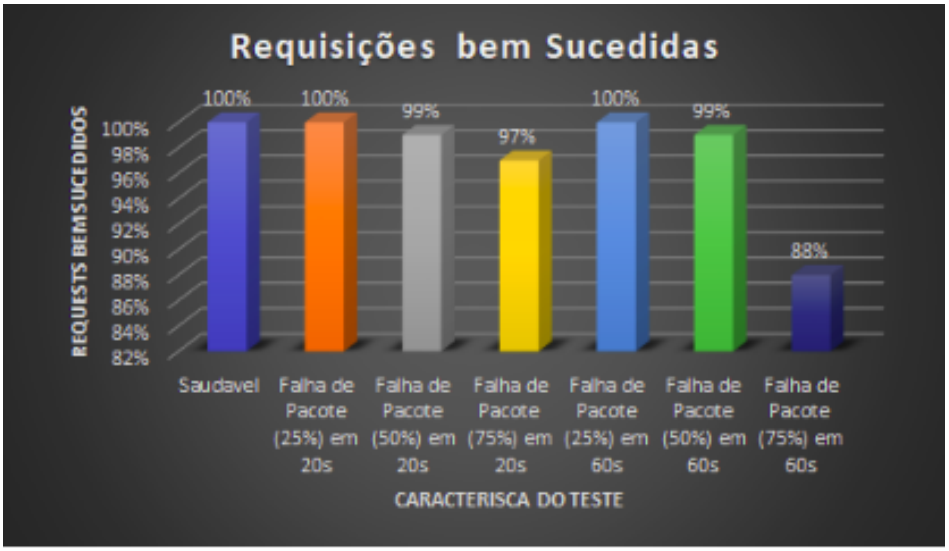

Figura 5.5: Gráfico da taxa de respostas bem sucedidas por com a injeção de perda de pacote. Gráfico formado a partir da tabela 5.2

Tabela 5.2: Taxa de respostas bem sucedidas para determinada injeção de perda de pacote.

Tipo de Injeção	**Taxa Sucesso**
Nenhuma (Saudável)	100%
Falha de Pacote (25%) em 20s	100%
Falha de Pacote (50%) em 20s	99%
Falha de Pacote (75%) em 20s	97%
Falha de Pacote (25%) em 60s	100%
Falha de Pacote (50%) em 60s	99%
Falha de Pacote (75%) em 60s	88%

o tempo de resposta foi 17,47 vezes mais lento que o ambiente saudável, como se pode observar a partir das figuras 5.6 e 5.7

Tabela 5.3: Tempo médio de resposta para determinada injeção de latência na rede.

Tipo de Injeção	**Tempo Médio Normalizado**
Nenhuma (Saudável)	1
Latência (700ms) em 20s	5,52
Latência (1000ms) em 20s	5,53
Latência (5000ms) em 20s	5,68
Latência (700ms) em 60s	15,5
Latência (1000ms) em 60s	15,47
Latência (5000ms) em 60s	17,47

É interessante notar, que apesar do tempo de resposta ao se injetar diferentes latências na rede ser alto o número de falhas foi zero durante os testes. Isso demonstrou que a utilização do usuário fica prejudicada nesse cenário, porém ainda há resposta do sistema.

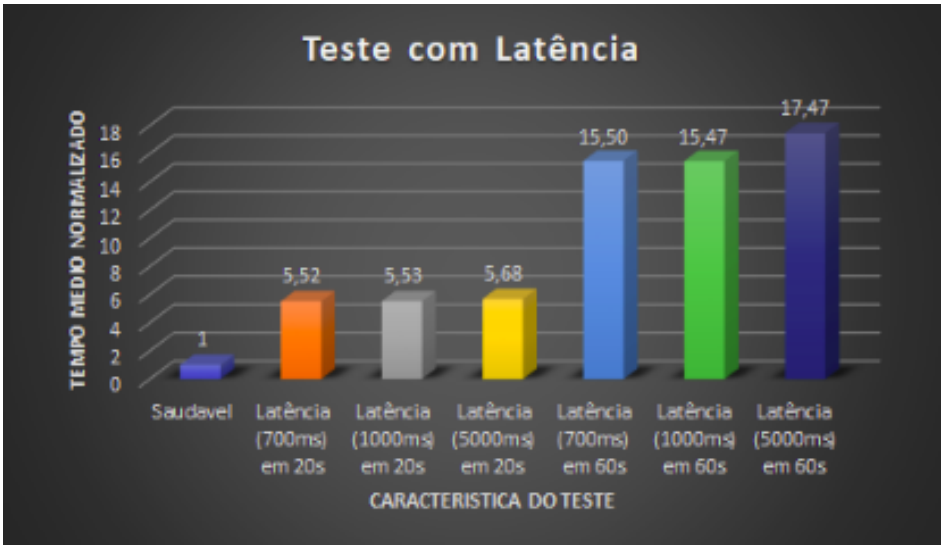

Figura 5.6: Gráfico do tempo médio de resposta para determinada injeção de latência na rede. Gráfico formado a partir da tabela 5.3

Figura 5.7: Gráfico do número de respostas bem sucedidas por injeção de latência na rede. Gráfico formado a partir da tabela 5.4

Tabela 5.4: Taxa de respostas bem sucedidas para determinada injeção de latência.

Tipo de Injeção	**Taxa Sucesso**
Nenhuma (Saudável)	100%
Latência (700ms) em 20s	100%
Latência (1000ms) em 20s	100%
Latência (5000ms) em 20s	100%
Latência (700ms) em 60s	100%
Latência (1000ms) em 60s	100%
Latência (5000ms) em 60s	100%

Ao injetar falhas no *MySql* verificamos que essa diferença aumentava conforme a intensidade da falha, porém não ocorreram perda de informações. Como é possível analisar

ID	local	created_at	text	reg_date	reg_date-created_at
365	ur heart	2019-11-10 21:34:08	RT @***: Boomers Queer Farmer Leftists	2019-11-10 21:36:50	162
364	Rio de Janeiro	2019-11-10 21:34:04	RT @***: POR FAVOR LULA ! Venha para rua. Você vai ver a popularidade que tem. Lula não anda cercado de povo, só de militantes e...	2019-11-10 21:36:47	163
363	N/A	2019-11-10 21:34:04	@*** @*** @*** Não querer ver é diferente de não se importar! Desenha pra menina ali q ela...	2019-11-10 21:36:45	161
362	N/A	2019-11-10 21:34:02	RT @***: Pra mim só Jesus Cristo, Papai Noel e sandy que não xingam	2019-11-10 21:36:43	161
361	N/A	2019-11-10 21:33:58	RT @***: Nuovo meme per l'Inter Twitter	2019-11-10 21:36:41	163
360	کراچی, پاکستان	2019-11-10 21:33:56	RT @***: not @ non shawols and some shawols acting like kibum's song becoming a trend on tiktok is the most disgusting thing ever.....	2019-11-10 21:36:38	162

Figura 5.8: Tabela com exemplos do formato do dado gravado no *MySql*. *local, created_at, text*, representa o local, o horário e o conteúdo do *tweet* que foi postado em um momento de injeção de falha. Os valores em *reg_date* representam o momento em que o dado foi inserido no banco de dados.

na figura 5.8, a latência entre a informação ser gerada pelo usuário do *Twitter* e ela ser refletida em nossa aplicação foi maior do que 160 segundos. Contudo, ainda com a latência alta, não ocorreram perda de informações.

Tabela 5.5: Consolidado dos Resultados Apresentados.

Intensidade de Injeção	**Latência**	**Perda de Pacote**
Nenhuma (Saudável)	1	1
Baixa em 20s	4,6	5,5
Media em 20s	5,2	5,5
Alta em 20s	7,2	5,7
Baixa em 60s	8,1	15,5
Media em 60s	17,3	15,5
Alta em 60s	17,9	17,5

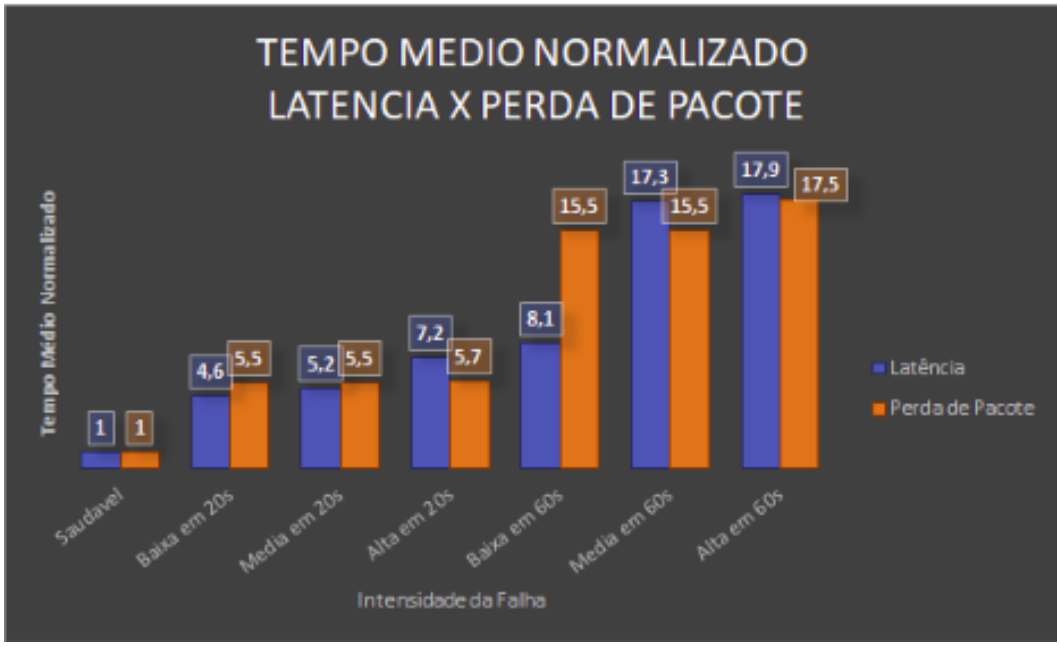

Figura 5.9: Gráfico de comparação entre o tempo de resposta de injeção de latência e perda de pacote. Gráfico formado a partir da tabela 5.5

Com base na figura 5.9, podemos comparar o resultados de injeção de falhas, é possível notar que eles acompanham o impacto gerado em nível de intensidade, com certa variação. A maior diferença entre as falhas injetadas é a quantidade falhas na requisição, mostrando que em um mesmo sistema, falhas diferentes podem gerar impactos.

6 CONCLUSÃO

Os testes realizados visam entender como as falhas e testes de estresse podem influenciar em *softwares* principalmente open sources vastamente utilizados na atualidade. Visou-se aproximar a arquitetura criada o máximo possível de ambientes produtivos e dessa forma trazer os testes para a realidade de diversos *softwares* utilizados pela comunidade.

Dessa forma, é possível perceber a relevância do tema, dado que tal tipo de teste pode ser aplicado de forma genérica podendo gerar *insights*. A utilização desse tipo de teste tem o potencial de prevenir falhas que poderiam ter sido evitadas durante o desenvolvimento de *software* ou com ajustes de arquiteturas, aumentando a disponibilidade e a confiabilidade dos sistemas.

Os resultados encontrados demonstram que falhas diferentes geram impactos semelhantes na experiência de um usuário final. O critério utilizado de comparação, o tempo de resposta, teve uma variação semelhante para a mesma categoria de impacto. Entretanto, quando se analisa o retorno de resposta 404, em casos mais graves, como perda de 75% ou mais de pacotes, a falha passa a ser crítica.

O objetivo de detectar falhas e entender o comportamento de um sistema composto por peças bastante utilizadas na atualidade foi cumprido. Mesmo se tratando de componentes resilientes individualmente, concluímos isso não garante uma alta resiliência do ambiente necessitando de outros cuidados que vão além do próprio desenvolvimento.

Como sugestão de estudos futuros, estudar remediações automáticas para os problemas encontrados de forma a contornar problemas que ocorrem no mundo real, há grande potencial. Dessa forma, ao entender as falhas que podem ocorrer, também seria construído uma solução para remediar o problema e evitar impactos graves para o usuário final.

Referências Bibliográficas

[Bal,] The ballista project. http://users.ece.cmu.edu/~koopman/ballista/. Acessado em: 29-11-2019.

[had,] Big data framework. https://dzone.com/articles/hadoop-vs-spark-choosing-the-right-big-data-framew. Acessado em: 29-11-2019.

[cic,] Ciclo de vida de um pod. https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/. Acessado em: 25-11-2019.

[con,] Container runtimes. https://kubernetes.io/docs/setup/production-environment/container-runtimes/. Acessado em: 30-11-2019.

[doc,] Docker deamon. https://docs.docker.com/config/daemon/. Acessado em: 22-11-2019.

[kub, a] Documentacao kubernetes. https://kubernetes.io/docs/home/. Acessado em: 29-11-2019.

[vol,] Documentação docker. https://docs.docker.com/storage/volumes/. Acessado em: 29-11-2019.

[kub, b] Documentação kubernetes - instalação. https://kubernetes.io/docs/tasks/tools/install-kubectl/. Acessado em: 21-11-2019.

[kub, c] Documentação kubernetes - instalação. https://kubernetes.io/docs/tasks/administer-cluster/cluster-management/#creating-and-configuring-a-cluster. Acessado em: 21-11-2019.

[red,] Documentação kubernetes. modelo de redes. https://kubernetes.io/docs/concepts/cluster-administration/networking/#the-kubernetes-network-modell. Acessado em: 29-11-2019.

[Fuz,] Fuzz testing of application reliability. http://pages.cs.wisc.edu/~bart/fuzz/fuzz.html. Acessado em: 29-11-2019.

[kub, d] Kubernetes. what is kubernetes? 2016. http://kubernetes.io/docs/whatisk8s/. Acessado em: 29-11-2019.

[mod,] Modelo de redes - documentação kubernetes. https://kubernetes.io/docs/concepts/cluster-administration/networking/#the-kubernetes-network-modell. Acessado em: 26-11-2019.

[pip,] pip - the python package installe. https://pip.pypa.io/en/stable/. Acessado em: 30-11-2019.

[pri,] Principles of chaos engineering. https://principlesofchaos.org/?lang=ENcontent. Acessado em: 29-11-2019.

[BI,] Referencia do bi. https://referenciaBI.com. Acessado em: 29-11-2019.

[rep,] Replication controler. https://kubernetes.io/docs/concepts/workloads/controllers/replicationcontroller/. Acessado em: 27-11-2019.

[ins,] Script de inserção de dados kafka- mysql. https://github.com/Guipupio/TCC. Acessado em: 21-11-2019.

[ser,] Serviços kubernetes. https://kubernetes.io/docs/concepts/services-networking/service/. Acessado em: 25-11-2019.

[cnc,] Survey shows kubernetes leading as orchestration platform. https://www.cncf.io/blog/2017/06/28/survey-shows-kubernetes-leading-orchestration-platform/. Acessado em: 29-11-2019.

[twi,] Twitter getting started. https://help.twitter.com/en/twitter-guide. Acessado em: 22-11-2019.

[net,] Weave net with kubernetes in just one line. https://www.weave.works/blog/weave-net-kubernetes-integration/. Acessado em: 29-11-2019.

[JCr,] Welcome to jcrasher - an automatic robustness tester for java. http://ranger.uta.edu/~csallner/jcrasher/index.html. Acessado em: 29-11-2019.

[Ali Basiri, 2016] Ali Basiri, Niosha Behnam1, R. d. R. L. H. L. K. J. R. C. R. (2016). Chaos engineering. *IEEE*, 33:35–41.

[Ding Yuan, 2014] Ding Yuan, Yu Luo, X. Z. G. R. R. X. Z. Y. Z. P. U. J. M. S. (2014). Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems*. USENIX.

[Hüttermann, 2012] Hüttermann, M. (2012). *DevOps for Developers*. Springer Science + Business Media.

[N. Steinleitner, 2006] N. Steinleitner, H. Peters, X. F. (2006). Implementation and performance study of a new nat/firewall signaling protocol. *26th IEEE International Conference on Distributed Computing Systems Workshops (ICDCSW'06)*.

[Regina Moraes, 2014] Regina Moraes, Hélène Waeselynck, J. G. (2014). Uml-based modeling of robustness testing. *IEEE International Symposium on High Assurance Systems Engineering*, pages 168–175.

[THEIN, 2016] THEIN, K. M. M. (2016). Apache kafka: Next generation distributed messaging system. *International Journal of Scientific Engineering and Technology Research*, 03:9478–9483.

Apêndices

A Configuração do Ambiente

A figura A.1 ilustra a arquitetura do ambiente utilizado neste trabalho.

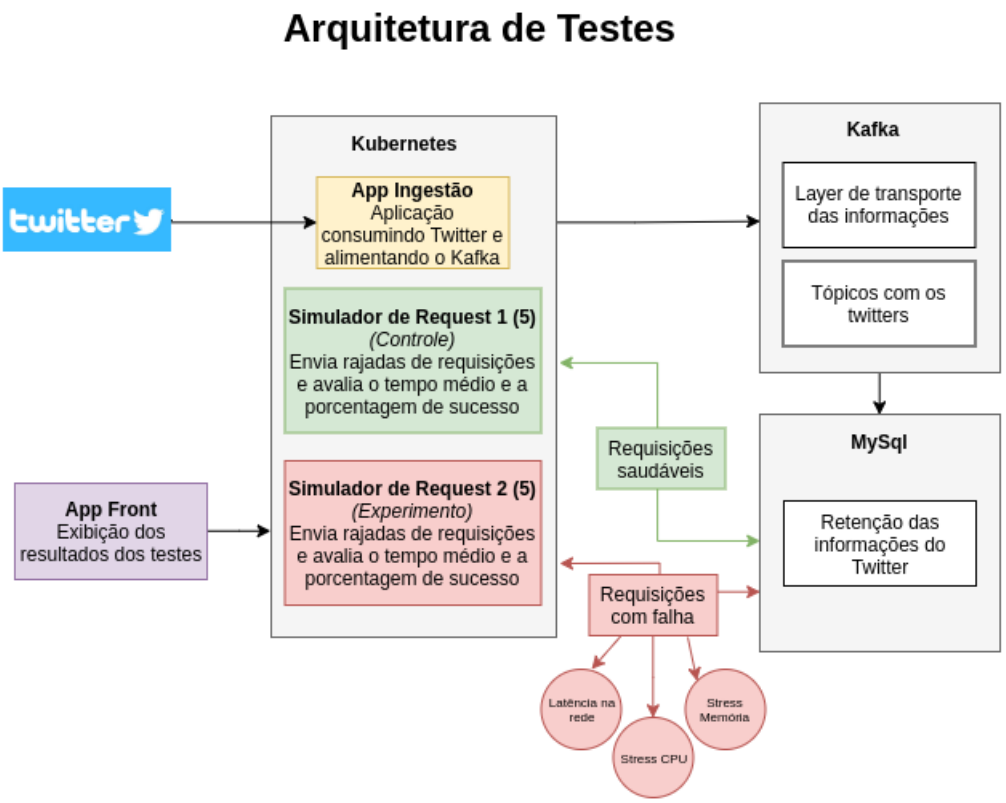

Figura A.1: Arquitetura do sistema criado para aplicação de Chaos Engineering

O projeto foi instanciado na plataforma *Google Cloud*, e utilizou 5 instâncias para executar uma simples aplicação. Três instâncias eram responsáveis por conter os *PODs* e *containers* associados ao *Kubernetes*, outra responsável em manter um banco *Mysql* utilizado para receber os *twitts* provenientes de um serviço integrado ao *Twitter*[twi,], além de armazenar informações dos testes de *Chaos Engineering*. Por fim, a última instância continha a aplicação *Kafka*.

Todas as instâncias possuiam sistema operacional **Ubuntu 18.04 LTS**. A Tabela 1 exibe a relação de número de CPUs e Memória Ram das máquinas utilizadas no sistema.

Tabela A.1: Especificações das instâncias criadas no Google Cloud

Função	**CPUs**	**RAM**
Master - Kubernetes	4	3.6GB
Slave 1 - Kubernetes	2	3GB
Slave 2 - Kubernetes	2	3GB
Kafka	1	2GB
Mysql	1	2GB

Inicialmente configuramos o projeto utilizando um *docker* principal para construir uma imagem com toda a arquitetura do projeto. Configuramos um ambiente onde os *dockers* não possuiam um *docker daemon*[doc,] independente, entretanto estavam conectados ao *docker daemon* do *host* do sistema.

A.1 Kubernetes

Para instalação do *Kubernetes*, existem diversos tutoriais pela *internet*, entretanto foram seguidas as instruções disponíveis na própria documentação do *Kubernetes*[kub, b]. A Figura A.2 a seguir é mostrado o *script*, que realiza os passos sugeridos na documentação e, que foi utilizado para setar o ambiente de teste. Este *script* pode ser encontrado no **github do projeto**.

```
# Atualizamos os repositorios das maquinas do Google Cloud
apt-get update

#Desligamos o swap das maquinas
swapoff -a

# Intalando Docker nas VMs do Google Cloud
curl -fsSL https://docs.docker.com | bash

# Chave do repositorio com comandos do kubernetes
curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg \
 | apt-key add -

# ADicionamos o repositorio
echo "deb http://apt.kubernetes.io/ kubernetes-xenial main" \
 > /etc/apt/sources.list.d/kubernetes.list

# Baixamos a lista de pacotes disponiveis no novo repositorio
apt-get update

# Instalamos os comandos necessarios do kubernetes
# kubeadm - responsavel por montar o cluster
# kubectl - resposavel por operar/iteragir com o cluster
# kubelet - como se fosse um agente operando em todos os nos,
# conversando com o nó master
apt-get install -y kubelet kubeadm kubectl
```

Figura A.2: *script* para Instalação do *Docker* e *Kubernetes*

Ao executar o *script* ilustrado na figura A.2, é esperado que o *Kubernetes* seja

instalado na máquina. Esse *script* foi executado nas três instâncias *Kubernetes* do sistema, para todas as três máquinas, possuírem as tecnologias *Docker* e *Kubernetes*.

Para configurar o *cluster Kubernetes* no sistema, é necessário executar a sequência de comandos definidos no *script* ilustrado na figura A.3, novamente, este *script* pode ser encontrado no **github do projeto**.

```
# Para setar o cluster master
# OBS: Desde Comando sera informado o comando para
# conectar as maquinas slaves neste cluster
kubeadm init --apiserver-advertise-address $(hostname -i)

# Criacao de de estrutura de diretorios
mkdir -p $HOME/.kube

# Copia das configuracaoes do kubernetes para
# comunicacao do kubectl
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config

# Altera as permissoes das configuracoes copiadas
sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

((a)) Comandos Executados no *Node* Master do *Kubernetes*

```
## ----------------- NAS MAQUINAS SLAVES -----------------
# Nota este comando pode ser diferente.. pois ele eh informado a
# partir do primeiro comando deste arquivo
kubeadm join 10.128.0.10:6443 --token xws0a5.bpzv3ojm5pohql9k \
--discovery-token-ca-cert-hash \
sha256:130a4d8dbde047efad28fe446214d5fd71e74862e73b942918f907c9815d646e
```

((b)) Comandos Executados nos *Nodes Slaves* do *Kubernetes*

```
## ----------------- NA MASTER -----------------
## Realizar deploy do POD Network, responsavel por estabelecer
## uma comunicacao entre os PODS
kubectl apply -f \ "https://cloud.weave.works/k8s/net?k8s-version=\
$(kubectl version | base64 | tr -d '\n')"
```

((c)) *Deploy* do POD *Weave-network*

Figura A.3: Comandos para configurar o *cluster* de *Kubernetes* utilizado no trabalho.

Os comandos ilustrados na figura A.3(a) devem ser executado na máquina *master* e, a partir deles, é informado um *token*, que será utilizado nas máquinas *slaves*, para a formação do *cluster*, conforme ilustra a figura A.3(b). Note que o *token* da figura A.3(b) será **diferente** para cada configuração. A figura A.3 foi construida com base na documentação do *Kubernetes*, para configuração de *cluster*[kub, c]. Por fim, no *Node master*, foi

executado o comando presente na figura A.3(c), que realiza o *deploy* dos *PODs Weave-network*, com o objetivo de estabelecer uma comunicação entre os *PODs* do *cluster* (seção 4.4.7).

A.2 Redis

O principal objetivo deste componente é armazenar o **IP** dos serviços utilizados pelos *PODs* do *Kubernetes*, evitando configurações mais complexas no próprio *Kubernetes*.

Na instância *master*, foi instalado o redis-cli (*Redis client*), o qual fornece uma interface, em linha de comando para facilitar a comunicação com o banco Redis. Para instalar o *Redis client*, foi necessário executar os comandos:

```
# sudo apt-get update
# sudo apt-get install redis-cli
```

Ao concluir a instalação, é possível executar uma série de comandos amigáveis[26] para a utilização do Redis. A aplicação Redis utilizada neste trabalho foi a oferecida pelo *docker*. Dessa forma com o *docker* instalado, para iniciar uma instância do Redis, foi necessário executar o seguinte comando:

```
# docker run -d -p 80:80 --name redis redis
```

Este comando faz com que um *container* seja levantado com a imagem oficial do Redis 5.0.7 [27]. A *flag* -d indica que a execução deste *container* será executado em *background*, a *flag* -p 80:80, indica que a porta 80 do *container* está exposta pela porta 80 do *host*. A *flag* name, nomeia o *container* e, por fim, tem-se o nome da imagem *docker*, que define o conteúdo do *container*.

Para registrar informações no Redis, assim como em qualquer outra estrutura de armazenamento, é necessário saber seu IP. Como a aplicação Redis está armazenada em um Docker *container*, para obter o IP do *container*. É necessário utilizar o comando abaixo:

```
# docker inspect --format '{{ .NetworkSettings.IPAddress }}' redis
```

A.3 Kafka

A primeira coisa a se fazer para configurar o *Kafka* é instalar o java. Estamos considerando a utilização de um ambiente *Ubuntu*. É importante notar que os diretórios utilizados podem divergir assim como a versão dos pacotes.

Para verificar se o java já está instalado, execute o comando:

```
# java -version
```

Caso não esteja instalado, execute seguinte comando para instalá-lo:

```
# apt install openjdk-8-jdk
```

Para verificar se a instalação foi concluída com sucesso, escreva, novamento, no terminal:

```
# java -version
```

e confira o output.

A.3.1 Nodes

Após instalar o java, iniciamos a instalação do Kafka. Para isso, realizamos o download do pacote do Apache Kafka. Para baixar diretamente no Ubuntu server e extrair o conteúdo, execute os comandos:

```
curl https://www-us.apache.org/dist/kafka/2.3.0/kafka_2.12-2.3.0.tgz \ --
tar -xvf kafka.tgz
```

Por fins de facilidade, export o PATH do Kafka, para o *.bashrc*:

```
# export PATH=/home/kafka/kafka_2.12-2.3.0/bin:$PATH
```

Dentro da pasta do *kafka_2.12-2.3.0* crie uma pasta *"data"* e outra *"data/zookeeper"*:

```
# mkdir data
# mkdir data/zookeeper
```

Abra o arquivo *config/zookeeper.properties* e altere a linha do diretório para a pasta criada:

```
# dataDir=/home/kafka/kafka_2.12-2.3.0/data/zookeeper
```

Agora, para iniciar o serviço, execute o comando a seguir e confira que o zookeeper está executando o bind na porta 2181.

```
# zookeeper-server-start.sh   \
    config/zookeeper.properties
```

O output esperado é:

"[...] INFO binding to port 0.0.0.0/0.0.0.0:2181 (org.apache.zookeeper.server.NIOServerCnxnFactory) "

Agora iremos criar um diretório onde serão armazenados os dados do Kafka em *data/kafka*. Para isso, dentro do diretório do *kafka/data*, crie uma pasta chamada kafka. Em sequência, altere o arquivo **config/server.properties**. Altere a linha:

```
log.dirs=[...]
```

Para:

```
log.dirs=/home/kafka/kafka_2.12-2.3.0/data/kafka
```

Finalmente, para executar o Kafka execute

```
# kafka-server-start.sh config/server.properties
```

Em resumo, após as configurações efetuadas, para iniciar o Kafka, é necessário executar dois comandos, na seguinte ordem, dentro da pasta **kafka_2.12-2.3.0**:

```
# zookeeper-server-start.sh  config/zookeeper.properties
# kafka-server-start.sh config/server.properties
```

A.3.2 Tópicos

Para criar tópicos no *Kafka* é necessário se conectar ao *Zookeeper*. Para isso, inserir o seguinte comando:

```
# kafka-topics.sh --zookeeper 127.0.0.1:2181 \
    --topic first_topic --create --partitions 3 --replication-factor 1
```

Para listar todos os tópicos existentes

```
# kafka-topics.sh --zookeeper 127.0.0.1:2181 --list
```

Para obter informações dos tópicos:

```
# kafka-topics.sh --zookeeper 127.0.0.1:2181 --topic \
    first_topic --describe
```

Para remover um tópico:

```
# kafka-topics.sh --zookeeper 127.0.0.1:2181 --topic \
    second_topic --delete
```

A.3.3 Producer

Para "criar" uma *Producer* deve-se executar o comando abaixo:

```
# kafka-console-producer.sh --broker-list 127.0.0.1:9092 \
    --topic fisrt_topic
```

Para alterar a forma de confirmação da *Producer*:

```
# kafka-console-producer.sh --broker-list 127.0.0.1:9092 \
    --topic fisrt_topic --producer-property acks=all
```

Caso o tópico referenciado ainda não exista, o *Producer* irá criar o novo tópico. Porém como o tópico será criado com valores *defaults*, isto não é uma boa prática, pois algumas ações importantes podem não ser aplicadas, como o fator de replicação. Para alterar os valores *defaults*, é necessário configurar o arquivo *config/server.properties*

A.4 Consumers

Para realizar um streaming de mensagens no *Kafka* com o consumer, execute o comando:

```
# kafka-console.consumer.sh --botstrap-server 127.0.0.1:9092 \
--topic first_topic
```

Para ler todas as mensagens do tópico execute o comando a seguir:

```
# kafka-console.consumer.sh --botstrap-server 127.0.0.1:9092 \
--topic first_topic --from-beginning
```

É importante notar nessa situação, que a ordem das mensagens no consumer não é mantida, pois o tópico possui 3 partições e a ordem é garantida apenas para o nível de partição.

Podemos definir um grupo de consumidores, para balancear a carga:

```
# kafka-console-consumer.sh --bootstrap-server 127.0.0.1:9092 \
--topic first_topic --group my-first-application
```

Para listar todos os grupos consumidores, execute:

```
# kafka-consumer-groups.sh --bootstrap-server localhost:9092 \
    --list
```

Para obter mais informações do grupo consumidor execute:

```
# kafka-consumer-groups.sh --bootstrap-server localhost:9092 \
    --describe --group my-first-application
```

A.5 Criando um projeto genérico

É necessário instalar o **Java 8**. *Kafka* irá converter tudo que enviarmos para ele em *bytes*. Por isso é importante utilizarmos o "*key.serializer*" e "*value.serializer*" para ajudar a producer saber que tipo de valor estamos enviando para o *Kafka*.

Para o *Kafka* começar a receber informações externas é preciso alterar o arquivo *server.properties*, adicionando a linha:

```
listeners=PLAINTEXT://a.b.c.d:9092.
```

Dessa forma o *Kafka* passa a receber informações de um solicitante externo.

A.6 Criando o tópico para o Twitter

Na máquina do *Kafka*, será realizar algumas configurações para a efetivação da comunicação com o *Kafka* e a aplicação do *Twitter*. Inicialmente iremos criar um novo tópico.

```
# kafka-topics.sh --zookeeper 127.0.0.1:2181 --create \
    --topic twitter_tweets --partitions 6 --replication-factor 1
```

consumer:

```
# kafka-console-consumer.sh --bootstrap-server \
    <ip_publico>:9092 --topic twitter_tweet
```

A.7 Mysql

Para salvar os registros obtidos do Twitter, iremos criar uma instância do *MySql* versão 5.7.O, cujo sistema operacional utilizado para criar o ambiente é Ubuntu Server 18.04. Foi utilizado uma instância na Google Cloud para o ambiente ficar disponível. Para habilitar a comunicação com o serviço do *MySql* é necessário liberar a porta 3306. Caso contrário, não é possível acessar o *MySql* externamente.

Inicialmente no terminal é necessário realizar um update para atualizar os repositórios e em sequência executar o comando de instalação do mysql-server.

```
# sudo apt-get update
# sudo apt-get install mysql-server
# sudo service mysql start
```

Para verificar se o serviço está corretamente instalado, execute o seguinte comando:

```
# sudo service mysql status
```

Agora que o *MySql* está instalado, realizamos a criação do database e das tabelas que irão comportar os dados. Para acessar o banco de dados utilize o comando:

```
# mysql -h<ip_address> -u<usuario> -p<senha>
```

```
create database twitter;

CREATE TABLE `twitters` (
  `id` int(6) unsigned NOT NULL AUTO_INCREMENT,
  `local` varchar(255) NOT NULL,
  `created_at` datetime NOT NULL,
  `text` varchar(300) NOT NULL,
  `reg_date` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP \
    ON UPDATE CURRENT_TIMESTAMP,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=57119 DEFAULT CHARSET=utf8
```

Figura A.4: Query para criação de um banco Mysql, com o objetivo de armazenar as informações do Twitter

A partir da figura A.4, conseguimos registrar o local de onde o *tweet* foi enviado, a data em que o *tweet* foi publicado **(created_at)**, o momento em que ele foi inserido no banco de dados **(reg_date)**, que é atualizada no momento da inserção automaticamente, e o texto do *tweet* (**text**).

Agora que já temos a estrutura do *MySql* pronta para receber os dados, é necessário construir uma integração para consumir as mensagens enviadas para o *Kafka* e popular a base. Para isso foi criado um *script* em *Python*[ins,] que consome as informações do *Kafka* e insere no *MySql*.