Processamento de dados em larga escala com Apache Kafka®

Juliano Fantozzi NUSP 9791218, Vinícius Leite Ribeiro NUSP 10388200, Pedro Fernandes Pastorello NUSP 10262502

URL do repositório: https://gitlab.com/icmc-ssc0158-2021/2021/gcloud13

Instituto de Ciências Matemáticas e de Computação – Universidade De São Paulo (USP)
Caixa Postal 668 13560-970 São Carlos - SP - Brazil

{juliano.fantozzi,viniribeiro,pedropastorf}@usp.br

Abstract. *Distributed systems have been increasingly gaining industry and researchers attention, and have proven to be the silver bullet of scaling problems. This paper describes the process of modeling and implementing a distributed, message-driven system architecture, with Kafka being the central piece of it, a worker producer, a worker consumer, and a database, although the last two couldn't be implemented due to lack of human resource/time.*

Resumo. *Sistemas distribuídos têm recebido atenção crescente da indústria e de pesquisadores, se provando ser a bala de prata para solucionar problemas de escalabilidade. Esse paper descreve o processo de modelagem e implementação de uma arquitetura de sistemas distribuída, orientada a mensagens, tendo o Apache Kafka® como peça central, um produtor, um consumidor e uma base de dados, entretanto os dois últimos componentes não foram implementados devido a falta de recurso humano/tempo.*

1. Introdução

Os Sistemas distribuídos se tornaram amplamente utilizados devido ao seu potencial poder computacional. Antes da existência dos provedores Cloud, o acesso e o gerenciamento de toda a infraestrutura necessária era tão complexa que, para uma empresa conseguir chegar próximo ao que é oferecido atualmente pela indústria, seria necessário um esforço de ordens de grandeza maior do que sua capacidade atual.

Entretanto, atualmente podemos desfrutar de forma extremamente abstraída e com custo acessível os benefícios fornecidos por estes provedores, permitindo o desenvolvimento de sistemas com arquiteturas distribuídas. Tais arquiteturas seguem premissas para garantir maior poder de processamento, maior tolerância a falhas e maior disponibilidade.

A solução abordada nesse documento conta com: o Kafka sendo seu componente central, uma aplicação que cumpre o papel o mensageiro, garantindo que todas as mensagens sejam corretamente encaminhadas para os devidos canais de comunicação, o Worker Extractor (produtor em relação ao Kafka), binário em Golang, responsável por extrair dados provenientes de tweets disponibilizados pela API do Twitter e alimentar o Kafka com mensagens contendo uma estrutura customizada que

contempla os dados necessários para o processo de transformação, o Worker Consumer, também um binário em Golang, é responsável por consumir tais mensagens no Kafka e armazená-los em um banco de dados. Por fim o Data Studio é utilizado para para para criar visualizações baseadas nas informações contidas na base de dados.

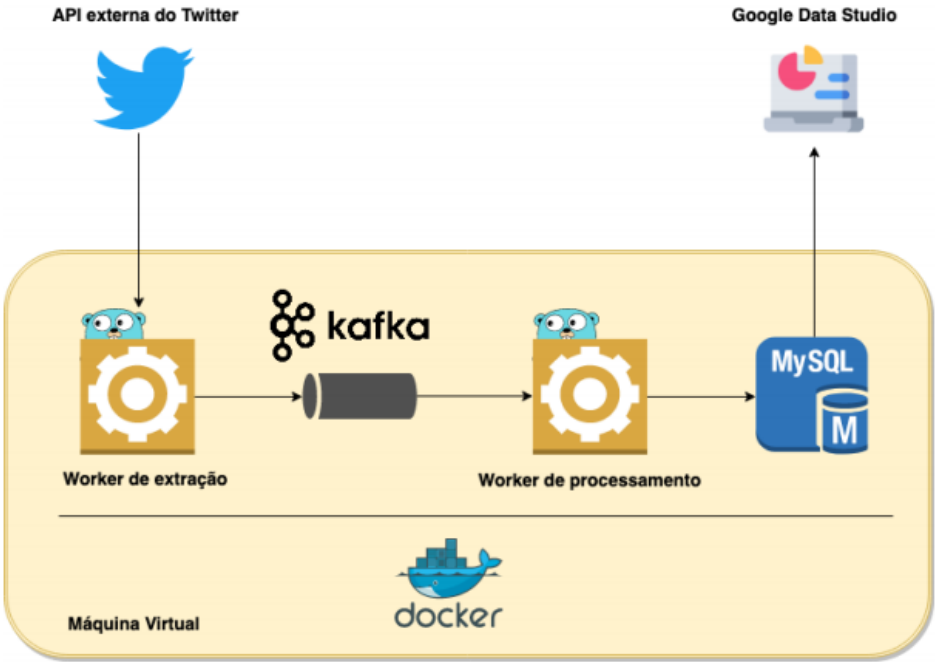

Figura 1. Modelo arquitetural

2. Componentes e tecnologias

2.1 Apache Kafka®

Camada de transporte de dados. Será o serviço de mensageria usado inicialmente para carregar os dados pré-processados vindos do serviço de extração para o serviço de transformação. Com essa abordagem será possível otimizar o processamento de forma assíncrona, garantindo integridade do processamento e oferecendo um fallback para caso a aplicação de Transformação falhe, já que quando ela se estabelecer novamente, poderá apenas voltar a consumir o conteúdo da fila.

2.2 Ansible

Camada de configuração de infraestrutura. Usado para configurar o ambiente e instalar dependências em máquinas remotas via SSH.

2.3 Terraform

Camada de configuração de infraestrutura. Será usado para provisionar infraestrutura em algum provedor de nuvem, no nosso caso AWS (IaaS). A abordagem de IaC busca versionar a infraestrutura, buscando garantir resiliência ao sistema, dando mais liberdade e segurança para realizar testes e buscar otimização da infraestrutura, além da agilidade em provisionar novos recursos de infraestrutura que essa abordagem oferece.

2.4 Docker

Docker Camada de virtualização. Utilizado para virtualizar, isolar e controlar o ciclo de vida das aplicações. Docker faz isso por meio de containers e uma API usada para interagir com os containers em execução. No geral, contêineres em execução são muito mais leves que Máquinas Virtuais e oferecem mais agilidade para provisionar novas réplicas da aplicação. Essa abordagem é essencial para abstrair a complexidade do gerenciamento dos ambientes contendo as aplicações já que cada container é uma representação ideal que contemple as necessidades para a execução da aplicação, possibilitando alterações nos ambientes que serão replicadas igualmente para todos os containers referentes a uma aplicação.

2.6 Docker Compose

Camada de orquestração. Utilizado para orquestrar os containers em execução e definir a infraestrutura usando a API do Docker. Será usado para controlar o consumo de recursos, configurar a rede e o roteamento, gerenciar volumes de armazenamento e monitorar a saúde dos serviços. Em seus arquivos de configuração são explicitados os requisitos e dependências dos serviços em relação à infraestrutura.

2.7 Golang

Linguagem usada para o desenvolvimento do worker. É uma linguagem compilada otimizada para processamento assíncrono e distribuído. Go também é muito eficiente em termos de consumo de recursos e tempo de execução. É muito utilizada para processamento de dados e computação distribuída e em alguns grandes projetos de computação em nuvem, como o próprio Docker e o Kubernetes.

2.8 Mysql

Camada de armazenamento. Servidor de banco de dados relacional de código aberto. Foi o serviço de banco de dados escolhido por ser amplamente aceito no mercado, tendo uma grande comunidade e uma boa integração com o Google Data Studio e seria usado no caso do sistema completo e funcional.

2.9 Google Data studio

Camada de visualização dos dados (SaaS). Seria usado para fazer apresentações dos dados processados, consumidos diretamente do banco de dados por queries.

3. Considerações sobre virtualização

Atualmente as duas formas de virtualização mais relevantes são Máquinas Virtuais (VMs) e Containers. Como já mencionado anteriormente, se decidiu usar containers para este projeto, devido a dois principais fatores: custo e praticidade, tendo em vista que a limitação ao acesso de root e firewalls da máquina inviabiliza a operação.

4. Implementação

4.1 Extração dos dados

O worker de processamento foi implementado em Golang, por ser uma linguagem otimizada para processamento distribuído e paralelo. Ele é composto por duas rotinas paralelas principais que rodam em loop fechado: uma que extrai as informações da API a cada intervalo de tempo e outra que recebe os dados, os codifica em Avro, otimizando a quantidade de tráfego de rede, e os produz como mensagens para o broker do Kafka.

O worker não possui estado interno, portanto pode ser indefinidamente escalado horizontalmente.O intervalo de tempo entre cada busca é um valor relevante pois a API do Twitter impõe uma mecânica de *rate limit*, que limita a quantidade de requisições que um mesmo cliente pode fazer por janela de tempo. Esse é o maior gargalo do sistema, dado que usamos a API gratuita de consulta disponibilizada pelo Twitter. Mesmo que o sistema seja projetado para escalar e suportar grandes quantidades de dados, sem pagar pelas APIs restritas, não podemos aumentar a frequência de busca dos dados acima desse limite.

Os parâmetros da busca e o intervalo de tempo entre cada uma são configuráveis pelo arquivo de configuração do extrator, em *app/worker-extractor/config/extractor.yaml*. A conexão com o broker do Kafka é configurável pelo arquivo de configuração em *app/worker-extractor/config/kafka.yaml*.

Como a extração e envio ao broker é feita em intervalos de tempo, nosso sistema possui carga de trabalho orientada à processamento em lotes (batch workload).

4.2 Infraestrutura

Toda a infraestrutura e do projeto é provisionada via código. A infraestrutura para o sistema foi desenvolvida usando Terraform para provisionar uma máquina virtual e todos os componentes de rede relacionados usando AWS EC2. A ferramenta Ansible foi usada para automatizar as configurações feitas à essa máquina e o Docker foi usado como plataforma de virtualização para as aplicações em execução na máquina.

5. Considerações finais

O sistema é capaz de extrair as informações de forma automatizada e configurável, permitindo ser escalado horizontalmente de forma indefinida (sem considerar o gargalo imposto pelo *rate limiting*), de forma que a velocidade de processamento dos dados fosse limitada apenas pela quantidade de réplicas dos workers que se deseja ter.

Apesar do worker de processamento não ter sido implementado, por falta de tempo dos integrantes, o sistema encontra-se num estado entregável pois os dados estão sendo enviados com sucesso ao Kafka, que por sua vez pode então ser plugado a qualquer sistema capaz de consumir mensagens de um tópico Kafka.

Possíveis próximos passos para o sistema envolveriam aumentar a resiliência e a observabilidade do sistema. Por exemplo, a migração da infraestrutura para uma solução mais robusta e tolerante a falhas, como um cluster Kubernetes, a instalação de sistemas de monitoramento, como Prometheus, instalação de sistemas de coleta de logs, uma API de controle para worker, armazenamento distribuído, etc.