INFRAESTRUTURA PARA DESENVOLVIMENTO WEB

UNIDADE IV

APACHE KAFKA

Elaboração

Miriã Falcão Freitas

Produção

Equipe Técnica de Avaliação, Revisão Linguística e Editoração

SUMÁRIO

APACHE KAFKA

UNIDADE IV

Um número crescente de empresas está implementando soluções de *software* que utilizam *Big Data*, *Machine Learning* e aplicativos nativos da nuvem. Essa parte avançada e difícil do desenvolvimento de *software* requer especialistas experientes e versáteis e o uso de ferramentas escaláveis e de resiliência.

Uma dessas ferramentas é o *Apache Kafka*. Ele permite que você aceite o desafio proposto pelo *Big Data*, quando as tecnologias do *broker* baseadas em outros padrões falharam.

O que é *Kafka*? Como as empresas se beneficiam com sua implementação? O que pensar antes de introduzir o *Kafka* em sua organização? Nesta unidade elaboramos um guia simples do *Apache Kafka* para não técnicos e desenvolvedores que desejam se familiarizar com ele.

CAPÍTULO 1

ASPECTOS GERAIS

Kafka se originou por volta de 2008 e seus autores foram: Jay Kreps, Neha Narkhede e Jun Rao, que, na época, trabalhavam no *LinkedIn*. Era um projeto feroz de código aberto, agora comercializado pela Confluent, e usado como infraestrutura fundamental por milhares de empresas, variando de AirBNB a Netflix.

Jay Kreps, Co-fundador da Confluent, definiu *Kafka* da seguinte forma:

> Nossa ideia era que, em vez de nos concentrarmos em armazenar pilhas de dados, como bancos de dados relacionais, armazenamentos de valores-chave, *índices* de pesquisa ou caches, focaríamos em tratar os dados como um fluxo em constante evolução e crescente e criar um sistema de dados – e de fato, uma arquitetura de dados – orientada em torno dessa ideia. (NARKNEDE *et al.*, 2017, xi).

Apache Kafka no GitHub. Disponível em: https://github.com/apache/kafka. (GITHUB, 2020).

A ascensão em tempo real de Apache Kafka. Disponível em: https://www.datanami.com/2016/04/06/real-time-rise-apache-kafka/. (WOODIE, 2016).

Por assim dizer, Kafka ganhou vida **para superar o problema com fluxos contínuos de dados**, pois não havia solução naquele momento para lidar com esse fluxo de dados.

Por definição, o *Apache Kafka* é uma plataforma de *streaming* distribuída para a construção de *pipelines* de dados em tempo real e aplicativos de *streaming* em tempo real. Como funciona? Como um sistema de publicação/assinatura que pode entregar mensagens persistentes e em ordem de forma escalável.

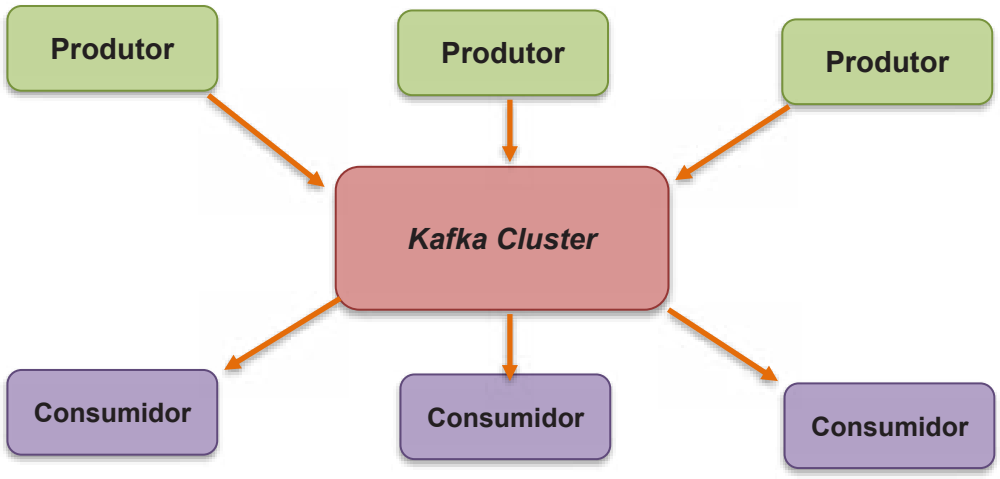
Kafka permite enviar mensagens entre aplicativos em sistemas distribuídos. O remetente pode enviar mensagens para Kafka, enquanto o destinatário recebe mensagens do fluxo publicado por Kafka.

As mensagens são agrupadas em tópicos – uma abstração primária do Kafka. O remetente (produtor) envia mensagens sobre um tópico específico. O destinatário (consumidor) recebe todas as mensagens sobre um tópico específico de muitos remetentes. Qualquer mensagem de determinado tópico enviada por qualquer remetente será enviada para todos os destinatários que estiverem ouvindo esse tópico.

O *Kafka* é executado como um *cluster* composto por um ou mais servidores, cada um dos quais é chamado de *broker* e a comunicação entre os clientes e os servidores é feita

com um protocolo TCP independente da linguagem, simples e de alto desempenho. A Figura 21 ilustra como o *Kafka* é executado.

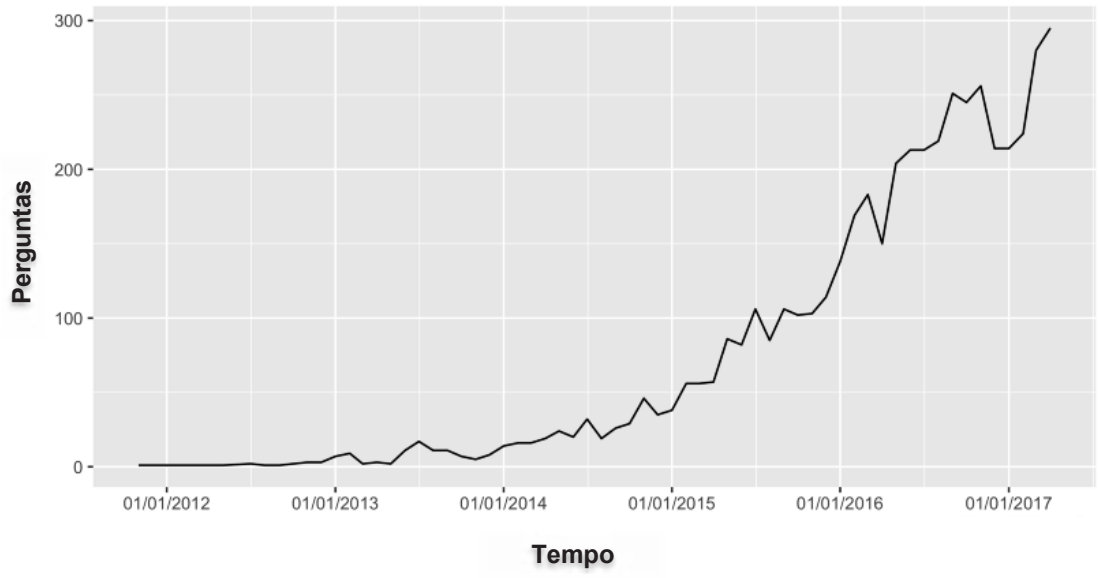
Figura 21. *Kafka* como *cluster*



Fonte: Elaborado pela autora.

O uso do Kafka continua a crescer em vários segmentos da indústria. A tecnologia se encaixa perfeitamente nos mercados orientados a dados, em mudança dinâmica, onde enormes volumes de registros gerados precisam ser processados à medida que ocorrem. A Figura 22 apresenta um gráfico do interesse crescente por *Kafka no Stack Overflow*, com volumes crescentes para ambas as métricas (perguntas e tempo).

Figura 22. Perguntas sobre *Apache Kafka no Stack Overflow* até abril de 2017



Fonte: RedMonk (2017).

Um imenso volume de informações que chega a alta velocidade e altos custos de armazenamento de dados leva os executivos a procurar soluções de processamento de dados em tempo real, econômicas, resilientes e escaláveis. Por quê? **É** tudo sobre dados ... e o cliente.

Os dados são a força vital da empresa moderna. As empresas querem descobrir tendências, aplicar *insights*, reagir às necessidades dos clientes em tempo real. A automação e a análise preditiva não são mais reservadas para unicórnios e aqueles que respondem rapidamente **às** possibilidades do mercado obtêm a vantagem sobre os concorrentes menos orientados a dados.

No momento, Kafka é o segundo projeto Apache mais ativo e visitado, usado por mais de 100.000 organizações em todo o mundo.

Como outros sistemas de mensagens, o *Kafka* facilita a troca de dados assíncrona entre processos, aplicativos e servidores e é capaz de processar até trilhões de eventos por dia. Toda solução de *big data* em tempo real pode se beneficiar de seu sistema especializado para atingir o desempenho desejado. O *Kafka* ajuda você a ingerir e mover rapidamente grandes quantidades de dados de maneira confiável e é uma ferramenta muito flexível para comunicação entre elementos pouco conectados dos sistemas de TI.

1.1. Reaja aos clientes em tempo real

O *Apache Kafka* usa o *Kafka Streams*, uma biblioteca cliente para criar aplicativos e microsserviços. É uma tecnologia de *big data* que permite processar dados em movimento e determinar rapidamente o que está funcionando e o que não está. Agora, a fonte típica de dados – dados transacionais, como pedidos, inventário e carrinhos de compras – é enriquecida com outras fontes de dados: recomendações, interações nas mídias sociais, consultas de pesquisa. A adoção do processamento de fluxo permite uma redução significativa do tempo entre o momento em que um evento é gravado e o momento em que o aplicativo de sistema e dados reage a ele, para que mais e mais empresas possam avançar para um processamento em tempo real como este. Todos esses dados podem alimentar os mecanismos de análise e ajudar as empresas a conquistar clientes.

1.2. Use os dados certos

A remoção de sistemas legados e silos de dados permite que os tomadores de decisão transformem seus dados em insights acionáveis, alcançando e informando todas as funções dos negócios. Com o processamento de fluxo, o estado de fluxos de dados em

constante mudança de diferentes fontes pode ser facilmente detectado, mantido e agir em tempo oportuno.

1.3. Dimensionar e automatizar

O principal ponto de venda do uso das soluções *Apache Kafka* é a sua escalabilidade. Os sistemas distribuídos são mais fáceis de expandir e operar como um serviço. A mudança para arquiteturas de microsserviço orientadas a eventos permite a agilidade dos aplicativos, não apenas do ponto de vista do desenvolvimento e dos *devops*, mas principalmente da perspectiva dos negócios.

O setor de TI já está lá e pronto para o mercado de massa. Não são mais apenas as bolsas de valores, que podem e devem pagar por esses sistemas, onde as transações e devem ser executadas com latência de milissegundos. Os provedores de nuvem oferecem toda a pilha, cobrindo ferramentas maduras para orquestrar, dimensionar, monitorar e rastrear – tudo de maneira automatizada.

Leia mais sobre **Transformação digital com processamento de fluxo**. Disponível em: https://blog.softwaremill.com/digital-transformation-with-streaming-application-development-100fb4189149. (WACHAL, 2019).

Os desenvolvedores e arquitetos de *software* amam o *Apache Kafka* porque ele vem com vários *softwares* que o tornam uma opção altamente atraente para integração de dados. Está se tornando a escolha clara para lidar com o processamento de distribuição, pois oferece todo o ecossistema de ferramentas para lidar com o fluxo de dados de maneira rápida e oportuna.

O *Kafka* foi projetado como um sistema distribuído e pode armazenar um grande volume de dados em *hardware* comum. Por ser também um sistema com várias assinaturas, ele permite que o conjunto de dados publicado seja consumido várias vezes. Não surpreende que seja uma das ferramentas mais populares para arquitetura de microsserviços. Ele serve como uma camada de comunicação intermediária e desacopla os módulos um do outro. A escalabilidade é alcançada dividindo um canal de comunicação, chamado tópico, em várias partições. Isso permite iniciar novas instâncias de um módulo específico até o número de partições.

Kafka tem uma comunidade muito ativa (Confluent e colaboradores). As versões são atualizadas regularmente, as APIs mudam, novos recursos são frequentemente adicionados. Os cofundadores da Kafka eram todos funcionários do *LinkedIn* quando iniciaram o projeto e Kafka se originou como um projeto de código aberto da Apache Software Foundation em 2011. Naquela época, Kafka estava ingerindo mais de 1 bilhão

de eventos por dia. Recentemente, o *LinkedIn* registrou taxas de ingestão de 7 trilhões de mensagens por dia.

A expansão dos conjuntos de dados com o *Kafka* é mais fácil, pois é melhor do que em outros sistemas de mensagens. Uma das razões é que o *Kafka* só garante pedidos por partição. A maneira como você escala o *Kafka* é adicionando mais partições; as mensagens de cada partição podem ser processadas em paralelo.

Finalmente, os desenvolvedores de *software* operam em ecossistemas de serviços que, juntos, trabalham em direção a algum objetivo comercial de nível superior. É benéfico tornar esses sistemas orientados a eventos e o *Kafka* é uma ótima ferramenta para isso.

Tem certeza de que está usando microsserviços? Disponível em: https://blog.softwaremill.com/are-you-sure-youre-using-microservices-f8d4e912d014. (MATTOKA, 2019).

Avaliando filas de mensagens persistentes e replicadas. Disponível em: https://softwaremill.com/mqperf/. (WARSKI, 2017).

***Sourcing* de eventos usando o Kafka**. Disponível em: https://blog.softwaremill.com/event-sourcing-using-kafka-53dfd72ad45d. (WARSKI, 2018).

CAPÍTULO 2

SEMÂNTICA

O uso do *Apache Kafka* está se tornando cada vez mais difundido. À medida que a quantidade de dados com que as empresas lidam explode, e à medida que as demandas por dados continuam aumentando, *Kafka* serve a um propósito valioso. Isso inclui seu uso como um barramento de mensagens padronizado devido a vários atributos principais.

Um dos atributos mais importantes do *Kafka* é sua capacidade de oferecer suporte à semântica exatamente uma vez. Com a semântica exatamente uma vez, você evita a perda de dados em trânsito, mas, também, evita o recebimento dos mesmos dados várias vezes. Isso evita problemas como o reenvio de uma atualização antiga do banco de dados, substituindo uma atualização mais recente que foi processada com êxito na primeira vez.

Em um sistema de mensagens de publicação/assinatura distribuída, os computadores que compõem o sistema sempre podem falhar independentemente um do outro. No caso de *Kafka*, um *broker* individual pode falhar ou uma falha de rede pode ocorrer enquanto o produtor está enviando uma mensagem para um tópico. Dependendo da ação que o produtor executa para lidar com essa falha, você pode obter semânticas diferentes:

- **Semântica pelo menos uma vez**: se o produtor recebe uma confirmação (*ack*) do *broker Kafka* e *acks = all*, significa que a mensagem foi gravada exatamente uma vez no tópico Kafka. No entanto, se um *ack* do produtor atingir o tempo-limite ou receber um erro, ele poderá tentar enviar a mensagem novamente, assumindo que a mensagem não foi gravada no tópico *Kafka*. Se o *broker* falhou pouco antes de enviar a confirmação, mas após a mensagem ter sido gravada com sucesso no tópico *Kafka*, essa nova tentativa levará a mensagem a ser gravada duas vezes e, portanto, entregue mais de uma vez ao consumidor final. E todo mundo adora um doador alegre, mas essa abordagem pode levar a trabalho duplicado e resultados incorretos.
- **Semântica no máximo uma vez**: se o produtor não tentar novamente quando uma confirmação exceder o tempo-limite ou retornar um erro, a mensagem poderá acabar não sendo gravada no tópico *Kafka* e, portanto, não entregue ao consumidor. Na maioria dos casos, será, mas, a fim de evitar a possibilidade de duplicação, aceitamos que algumas vezes as mensagens não sejam enviadas.
- **Semântica exatamente uma vez**: mesmo que um produtor tente enviar uma mensagem novamente, isso leva a mensagem a ser entregue exatamente uma vez ao consumidor final. A semântica exata uma vez é a garantia mais desejável, mas,

também, mal compreendida. Isso ocorre porque requer uma cooperação entre o próprio sistema de mensagens e o aplicativo que produz e consome as mensagens. Por exemplo, se, após consumir uma mensagem com êxito, você retroceder seu consumidor *Kafka* para um deslocamento anterior, receberá todas as mensagens desse deslocamento para a mais recente, novamente. Isso mostra por que o sistema de mensagens e o aplicativo cliente devem cooperar para que a semântica exata exista.

2.1. Falhas que devem ser tratadas

Para descrever os desafios envolvidos no suporte à semântica de entrega exatamente uma vez, vamos começar com um exemplo simples.

Suponha que exista um aplicativo de *software* produtor de processo único que envie a mensagem "*Hello Kafka*" para um tópico do *Kafka* de partição única chamado "EoS". Além disso, suponha que um aplicativo consumidor de instância única, do outro lado da linha, extraia dados do tópico e imprima a mensagem. No caminho feliz em que não há falhas, isso funciona bem, e a mensagem "*Hello Kafka*" é gravada na partição de tópicos da EoS apenas uma vez. O consumidor puxa a mensagem, a processa e confirma o deslocamento da mensagem para indicar que concluiu seu processamento. Ele não será recebido novamente, mesmo se o aplicativo consumidor falhar e reiniciar.

No entanto, todos sabemos que não podemos contar com o caminho feliz. Em escala, mesmo cenários improváveis de falha são coisas que acabam acontecendo o tempo todo.

- **Um *broker* pode falhar**: o *Kafka* é um sistema altamente disponível, persistente e durável, em que todas as mensagens gravadas em uma partição são persistidas e replicadas várias vezes (chamaremos *n*). Como resultado, o *Kafka* pode tolerar falhas do intermediário *n-1*, o que significa que uma partição estará disponível enquanto houver pelo menos um intermediário disponível. O protocolo de replicação do *Kafka* garante que, uma vez que uma mensagem tenha sido gravada com sucesso na réplica líder, ela será replicada para todas as réplicas disponíveis.
- **O RPC produtor-para-intermediário pode falhar**: a durabilidade em *Kafka* depende do recebimento do produtor do corretor. A falha em receber essa confirmação não significa necessariamente que a solicitação em si falhou. O *broker* pode falhar após escrever uma mensagem, mas antes de enviar uma confirmação ao produtor. Também pode falhar antes mesmo de escrever a mensagem no tópico. Como não há como o produtor conhecer a natureza da falha, ele é forçado a assumir que a mensagem não foi gravada com sucesso e a tentar novamente. Em alguns

casos, isso fará com que a mesma mensagem seja duplicada no *log* da partição *Kafka*, fazendo com que o consumidor final a receba mais de uma vez.

» **O cliente pode falhar**: a entrega exata uma vez também deve levar em conta as falhas do cliente. Mas como podemos saber que um cliente realmente falhou e não está apenas temporariamente particionado dos intermediários ou está passando por uma pausa no aplicativo? Ser capaz de dizer a diferença entre uma falha permanente e uma falha suave é importante; para correção, o corretor deve descartar as mensagens enviadas por um produtor de zumbis. O mesmo vale para o consumidor. Depois que uma nova instância do cliente é iniciada, ela deve poder se recuperar de qualquer estado que a instância com falha tenha deixado para trás e começar a processar a partir de um ponto seguro. Isso significa que as compensações consumidas devem sempre ser mantidas sincronizadas com a saída produzida.

2.2. Semântica exata no *Apache Kafka*

Antes da 0.11.x, o *Apache Kafka* suportava pelo menos uma vez a semântica de entrega e a entrega em ordem por partição. Como você pode ver no exemplo acima, isso significa que novas tentativas do produtor podem causar mensagens duplicadas. No novo recurso de semântica exatamente uma vez, reforçamos a semântica de processamento de *software* da *Kafka* de três maneiras diferentes e inter-relacionadas.

2.3. Idempotência: exatamente uma vez em ordem semântica por partição

Uma operação idempotente é aquela que pode ser executada várias vezes sem causar um efeito diferente do que apenas uma vez. A operação de envio do produtor agora é idempotente. No caso de um erro que cause uma nova tentativa do produtor, a mesma mensagem – que ainda é enviada pelo produtor várias vezes – será gravada apenas no *log Kafka* no *broker* uma vez. Para uma única partição, o produtor idempotente envia *remove* a possibilidade de mensagens duplicadas devido a erros do produtor ou do intermediário. Para ativar esse recurso e obter semântica exata uma vez por partição – ou seja, sem duplicatas, sem perda de dados e semântica em ordem – configure seu produtor para definir "*enable.idempotence = true*".

Como esse recurso funciona? Nos bastidores, funciona de maneira semelhante ao TCP; cada lote de mensagens enviadas para *Kafka* conterá um número de sequência que o *broker* usará para deduzir qualquer envio duplicado. Ao contrário do TCP, no entanto

– que fornece garantias apenas em uma conexão transitória na memória – esse número de sequência é mantido no *log* replicado; portanto, mesmo que o líder falhe, qualquer intermediário que assumir o controle também saberá se um reenvio é uma duplicata. A sobrecarga desse mecanismo é bastante baixa: são apenas alguns campos numéricos extras com cada lote de mensagens.

2.4. Transações: gravações atômicas em várias partições

O *Kafka* agora suporta gravações atômicas em várias partições por meio da nova API de transações. Isso permite que um produtor envie um lote de mensagens para várias partições, de modo que todas as mensagens no lote sejam eventualmente visíveis para qualquer consumidor ou nenhuma seja visível para os consumidores. Esse recurso também permite comprometer as compensações do consumidor na mesma transação, juntamente com os dados que você processou, permitindo, assim, a semântica de ponta a ponta exatamente uma vez. Aqui está um exemplo de *snippet* de código para demonstrar o uso da API de transações:

```
producer.initTransactions();
try {
  producer.beginTransaction();
  producer.send(record1);
  producer.send(record2);
  producer.commitTransaction();
} catch(ProducerFencedException e) {
  producer.close();
} catch(KafkaException e) {
  producer.abortTransaction();
}
```

O *snippet* de código acima descreve como você pode usar as novas APIs do Producer para enviar mensagens atomicamente para um conjunto de partições de tópicos. Vale ressaltar que uma partição de tópico *Kafka* pode ter algumas mensagens que fazem parte de uma transação, enquanto outras não.

Portanto, no lado do Consumidor, você tem duas opções para ler mensagens transacionais, expressas através da configuração do consumidor "*isolated.level*":

1. *read_committed*: além de ler as mensagens que não fazem parte de uma transação, também é possível ler as que são após a confirmação da transação.

2. *read_uncommitted*: leia todas as mensagens em ordem de deslocamento sem esperar que as transações sejam confirmadas. Essa opção é semelhante à semântica atual de um consumidor *Kafka*.

Para usar transações, você precisa configurar o Consumidor para usar o "*isolation.level*" correto, usar as novas APIs do Produtor e definir uma configuração do produtor "*transactional.id*" para um ID exclusivo. Esse ID exclusivo é necessário para fornecer continuidade do estado transacional entre as reinicializações do aplicativo.

2.5. Processamento de fluxo exatamente uma vez no *Apache Kafka*

Com base na idempotência e na atomicidade, o processamento de fluxo exatamente uma vez agora é possível por meio da API *Streams* no *Apache Kafka*. Tudo o que você precisa para fazer seu aplicativo *Streams* empregar a semântica exatamente uma vez é definir essa configuração como "*processing.guarantee = exact_once*". Isso faz com que todo o processamento ocorra exatamente uma vez; isso inclui fazer o processamento e também todo o estado materializado criado pelo trabalho de processamento que é gravado de volta no *Kafka*, exatamente uma vez.

É por isso que as garantias exatas fornecidas pela API *Streams* da Kafka são as garantias mais fortes oferecidas por qualquer sistema de processamento de fluxo até agora. Ele oferece garantias ponta a ponta, exatamente uma vez, para um aplicativo de processamento de fluxo que se estende dos dados lidos de *Kafka*, qualquer estado materializado para *Kafka* pelo aplicativo *Streams*, até a saída final gravada em *Kafka*. Sistemas de processamento de fluxo que dependem apenas de sistemas de dados externos para materializar o suporte do estado, garantias mais fracas para processamento de fluxo exatamente uma vez. Mesmo quando eles usam o *Kafka* como fonte para processamento de fluxo e precisam se recuperar de uma falha, eles só podem retroceder seu deslocamento *Kafka* para reconsumar e reprocessar mensagens, mas não podem reverter o estado associado em um sistema externo, levando a resultados incorretos quando o estado atualização não *é* idempotente.

A pergunta crítica para um sistema de processamento de fluxo é "*meu aplicativo de processamento de fluxo obtém a resposta certa, mesmo se uma das instâncias travar no meio do processamento?*" A chave, ao recuperar uma instância com falha, é retomar o processamento exatamente no mesmo estado de antes da falha.

Agora, o processamento do fluxo nada mais é do que uma operação de leitura-processo-gravação em um tópico do *Kafka*; um consumidor lê mensagens de um tópico *Kafka*,

alguma lógica de processamento transforma essas mensagens ou modifica o estado mantido pelo processador e um produtor grava as mensagens resultantes em outro tópico *Kafka*. O processamento de fluxo exatamente uma vez é simplesmente a capacidade de executar uma operação de leitura-processo-gravação exatamente uma vez. Nesse caso, "obter a resposta certa" significa não perder nenhuma mensagem de entrada ou produzir qualquer saída duplicada. Esse é o comportamento que os usuários esperam de um processador de fluxo exatamente uma vez.

Existem muitos outros cenários de falha a serem considerados além do simples que discutimos até agora:

- O processador de fluxo pode receber informações de vários tópicos de origem e a ordenação entre esses tópicos de origem não é determinística em várias execuções. Portanto, se você executar novamente o processador de fluxo que recebe informações de vários tópicos de origem, ele poderá produzir resultados diferentes.
- Da mesma forma, o processador de fluxo pode produzir saída para vários tópicos de destino. Se o produtor não puder executar uma gravação atômica em vários tópicos, a saída do produtor poderá estar incorreta se a gravação em algumas partições (mas não em todas) falhar.
- O processador de fluxo pode agregar ou unir dados em várias entradas usando os recursos de estado gerenciado fornecidos pela API do *Streams*. Se uma das instâncias do processador de fluxo falhar, será necessário reverter o estado materializado por essa instância do processador de fluxo. Ao reiniciar a instância, você também deve poder retomar o processamento e recriar seu estado.
- O processador de fluxo pode procurar informações enriquecedoras em um banco de dados externo ou chamando um serviço atualizado fora da banda. Dependendo de um serviço externo, o processador de fluxo é fundamentalmente não determinístico; se o serviço externo alterar seu estado interno entre duas execuções do processador de fluxo, isso resultará em resultados incorretos a jusante. No entanto, se tratado corretamente, isso não deve levar a resultados totalmente incorretos. Deve apenas levar à saída do processador de fluxo pertencente a um conjunto de saídas legais.

Falha e reinicialização, especialmente quando combinadas com operações não determinísticas e alterações no estado persistente calculado pelo aplicativo, podem resultar não apenas em duplicatas, mas em resultados incorretos. Por exemplo, se um estágio do processamento estiver computando uma contagem do número de eventos vistos, uma duplicata em um estágio de processamento *upstream* pode levar a uma contagem incorreta no *downstream*. Portanto, devemos qualificar a frase "exatamente

uma vez processamento de fluxo". Refere-se ao consumo de um tópico, materializando o estado intermediário em um tópico *Kafka* e produzindo para um, nem todos os cálculos possíveis feitos em uma mensagem usando a API do *Streams*. Alguns cálculos (por exemplo, dependendo de um serviço externo ou consumindo de vários tópicos de origem) são fundamentalmente não determinísticos.

A maneira correta de pensar em garantias de processamento de fluxo exatamente uma vez para operações determinísticas *é* garantir que a saída de uma operação de leitura-processo-gravação seja a mesma que seria se o processador de fluxo visse cada mensagem exatamente uma vez – como ocorreria no caso em que não ocorresse falha.

2.6. Espere, mas o que é exatamente uma vez para operações não determinísticas?

Isso faz sentido para operações determinísticas, mas o que significa exatamente o processamento de fluxo uma vez quando a própria lógica de processamento é não determinística? Digamos que o mesmo processador de fluxo que mantém uma contagem contínua de eventos recebidos seja modificado para contar apenas os eventos que satisfazem uma condição ditada por um serviço externo. Fundamentalmente, essa operação é de natureza não determinística, uma vez que a condição externa pode mudar entre duas execuções distintas do processador de fluxo, levando potencialmente a diferentes resultados a jusante. Então, qual é a maneira correta de pensar em garantias exatas para operações não determinísticas como essa?

A maneira correta de pensar em garantias *únicas* para operações não determinísticas *é* garantir que a saída de uma operação de processamento de fluxo de leitura-processo-gravação pertença ao subconjunto de saídas legais que seriam produzidas pelas combinações de valores legais de a entrada não determinística.

CAPÍTULO 3

ESTUDOS DE CASO

Com a evolução digital, o volume de transferência de dados **é** bastante grande e as empresas precisam de plataforma robusta, escalável e flexível que possa lidar com o *Big Data*. O *Apache Kafka* **é** uma plataforma de serviço de mensagens pub-sub que permite enviar mensagens de um extremo ao outro enquanto lida com grande parte dos dados. Também funciona bem para os tipos de serviços de mensagens *on-line* e *off-line*. A compatibilidade do *Apache Kafka* com as ferramentas de análise de dados em tempo real, como o *Apache Spark* e o *Storm*, oferece uma vantagem competitiva sobre as plataformas. Além disso, como o *Apache Kafka* **é** uma plataforma de código aberto, as empresas podem modificar o conjunto de operações conforme a conveniência.

Ele nos daria uma ideia melhor sobre as vantagens, a utilidade e a necessidade dessa plataforma de serviço se analisarmos detalhadamente o uso de casos do Apache Kafka. Então, vamos dar uma olhada em como o *Apache Kafka* beneficiou as empresas por meio de exemplos da vida real.

» Netflix

A *Netflix* não precisa de introdução. Uma das plataformas OTT mais inovadoras e robustas do mundo usa o *Apache Kafka* em seu projeto de *pipeline* de pedra angular para enviar e receber notificações. Existem dois tipos de *Kafka* usados pela Netflix: o *Fronting Kafka*, usado para coleta e armazenamento em *buffer* pelos produtores e o *Consumer Kafka*, usado para o roteamento de conteúdo para os consumidores.

Todos nós sabemos que a quantidade de dados processados na *Netflix* **é** enorme e a Netflix usa 36 *clusters Kafka* (dos quais 24 são *Fronting Kafka* e 12 são *Consumer Kafka*) para trabalhar em quase 700 bilhões de instâncias por dia.

A *Netflix* alcançou uma taxa de perda de dados de 0,01% por meio desse projeto de *pipeline* de *keystone* e o *Apache Kafka* **é** um fator essencial para reduzir essa perda de dados para uma quantidade tão significativa. A Netflix planeja usar a versão 0.9.0.1 para melhorar a utilização e disponibilidade de recursos.

» Spotify

O *Spotify*, que **é** a maior plataforma de música do mundo, possui um enorme banco de dados para manter 200 milhões de usuários e 40 milhões de faixas pagas.

Para lidar com uma quantidade tão grande de dados, o *Spotify* usou várias ferramentas de análise de *big data*. O *Apache Kafka* foi usado para notificar os usuários que recomendam

as listas de reprodução, colocando anúncios direcionados entre muitos outros recursos importantes. Essa iniciativa ajudou o *Spotify* a aumentar sua base de usuários e a se tornar um dos líderes de mercado na indústria da música.

No entanto, recentemente o *Spotify* decidiu que não queria manter e processar todos esses dados, então mudou para a plataforma pub-sub hospedada pelo Google para gerenciar os dados em crescimento.

» Uber

Existem muitos parâmetros em que um gigante da indústria de viagens como a *Uber* precisa ter um sistema que não comprometa os erros e seja tolerante a falhas. A *Uber* usa o *Apache Kafka* para executar seu programa de proteção de feridos em mais de 200 cidades. Os motoristas registrados na *Uber* pagam um prêmio a cada viagem e esse programa tem funcionado com sucesso devido **à** escalabilidade e robustez do *Apache Kafka*.

Ela alcançou esse sucesso em grande parte através do método de processamento em lote desbloqueado, que permite **à** equipe de engenharia da *Uber* obter um rendimento constante. As várias tentativas permitiram **à** equipe da *Uber* trabalhar na segmentação de mensagens para obter atualizações e flexibilidade de processos em tempo real.

A Uber planeja introduzir uma estrutura, na qual eles possam melhorar o tempo de atividade, crescer, escalar e facilitar o programa sem precisar diminuir o tempo do desenvolvedor com o Apache Kafka.

» Lyft

A *Lyft*, uma das empresas em crescimento no setor de transporte, **é** conhecida por seu foco avançado no uso da tecnologia. Ele está usando a versão corporativa do *Apache Kafka*, fornecida pelo *Confluent* como uma plataforma de *streaming* de dados. Antes, a *Lyft* usava o *Kinesis* para esse fim, mas **à** medida que o volume de dados aumentava, eles migraram para o Apache Kafka devido **à** sua escalabilidade e estabilidade para processar grande quantidade de dados.

Atualmente, a *Lyft* está planejando mudar para o *Flink*, para esses serviços, pois planeja mudar para mais modelos baseados em geografia.

» LinkedIn

O *LinkedIn*, uma das plataformas de mídia social B2B mais proeminentes do mundo, processa bem mais de um trilhão de mensagens por dia. E achamos que o número de

mensagens manipuladas pela Netflix era enorme. Esse número **é** surpreendente e o LinkedIn registrou um aumento de mais de 1200x nos **últimos** anos.

O *LinkedIn* usa *clusters* diferentes para aplicativos diferentes, para evitar o colapso da falha de um aplicativo, o que levaria a prejudicar os outros aplicativos no *cluster*. Os *clusters* do *Broker Kafka* no *LinkedIn* os ajudam a diferenciar e listar certos usuários para permitir uma maior largura de banda e garantir a experiência perfeita do usuário.

O *LinkedIn* planeja alcançar um número menor de taxa de perda de dados através do *Mirror Maker*. Como o *Mirror Maker* **é** usado como intermediário entre os *clusters Kafka* e os tópicos *Kafka*. No momento, há um limite para o tamanho da mensagem de 1 MB. Mas, através do ecossistema *Kafka*, o *LinkedIn* planeja permitir que os editores e os usuários enviem muito além desse limite no futuro próximo.

» Twitter

O *Twitter*, uma plataforma de mídia social conhecida por suas notícias em tempo real, as atualizações de histórias agora estão usando o *Apache Kafka* para processar a enorme quantidade de dados. Anteriormente, o *Twitter* costumava ter seu próprio sistema pub-sub *EventBus* para fazer essa análise e o processamento de dados, mas, olhando para os benefícios e as capacidades do *Apache Kafka*, eles fizeram a troca.

Como a quantidade de dados no *Twitter* aumenta a cada dia que passa, era mais lógico usar o *Apache Kafka* em vez de aderir ao *EventBus*. A migração para o *Apache Kafka* permitiu facilitar as operações de entrada e saída, aumentar a alocação de largura de banda, facilitar a replicação de dados e reduzir a quantidade de custos.

» Rabobank

O *Rabobank*, um banco multinacional holandês, conhecido por sua iniciativa digital, usa o *Apache Kafka* para um de seus serviços essenciais chamados *Rabo Alerts*. O objetivo deste serviço **é** notificar os clientes sobre os vários eventos financeiros, desde eventos simples, como transações de valor de ou em sua conta, até eventos mais complexos, como sugestões de investimentos futuros com base em sua pontuação de crédito, etc.

Essas notificações são notificações por *push* e, embora o *Rabobank* pudesse executar tarefas simples sem o *Apache Kafka*, eles precisavam de uma ferramenta robusta para realizar uma análise detalhada de uma enorme quantidade de dados.

» Goldman Sachs

A *Goldman Sachs*, uma gigante do setor de serviços financeiros, desenvolveu uma Plataforma Principal para lidar com dados que são quase em torno de 1,5 TB por semana.

Essa plataforma usa o *Apache Kafka* como uma plataforma de mensagens pub-sub. Mesmo assim, o número de dados manipulados pelo *Goldman Sachs* **é** relativamente menor que o do *Netflix* ou *LinkedIn*, ainda **é** uma quantidade considerável de dados.

Os principais fatores da *Goldman Sachs* foram o desenvolvimento de um sistema *Core Platform* que poderia atingir uma maior taxa de prevenção de perda de dados, recuperação mais fácil de desastres e minimizar o tempo de interrupção. Os outros objetivos significativos incluíam a melhoria da disponibilidade e a transparência, pois esses fatores são essenciais em qualquer empresa de serviços financeiros.

A *Goldman Sachs* alcançou esses objetivos com a implementação bem-sucedida do sistema *Core Platform* e o *Apache Kafka* foi um dos principais impulsionadores deste projeto.

» New York Times

O *New York Times*, um dos mais antigos meios de comunicação, transformou-se para prosperar nesta era de transformação digital. O uso de tecnologias como análise de *big data* não **é** novidade para esta casa de mídia. Vamos dar uma olhada em como o *Apache Kafka* transformou o processamento de dados do *New York Times*.

Sempre que um artigo **é** publicado no NYT, ele precisa ser disponibilizado em todos os tipos de plataformas e entregue aos seus assinantes em pouco tempo. Anteriormente, o NYT costumava distribuir os artigos e permitir o acesso aos assinantes, mas havia alguns problemas, como o que impedia os usuários de acessar os artigos publicados anteriormente. Ou aqueles em que era necessário um nível mais alto de coordenação entre equipes para manter e segmentar os artigos, pois equipes diferentes usavam APIs diferentes.

Para resolver esse problema, o NYT desenvolveu um projeto chamado de *Pipeline* de publicação, no qual o *Apache Kafka* **é** usado para remover problemas baseados em API por meio de sua arquitetura baseada em *log*. Como **é** um sistema de mensagens pub-sub, ele não pode apenas cobrir a integração de dados, mas, também, a parte de análise de dados, ao contrário de outros serviços de arquitetura baseados em *log*.

Eles implementaram o *Kafka* em 2015-16 e foi um sucesso para eles, pois o *Apache Kafka* simplificou as implantações de *back-end* e *front-end*. Também diminuiu a carga de trabalho dos desenvolvedores e ajudou o NYT a melhorar a acessibilidade do conteúdo.

» Shopify

O *Shopify*, uma renomada plataforma CMS de comércio eletrônico, usa o *Apache Kafka* para enviar notificações que variam de atualizações, ofertas a muitos outros cenários

relacionados a serviços. O *Shopify* também implantou o *Apache Kafka* para atender **à** agregação de *logs* e aos propósitos de gerenciamento de eventos.

O *Shopify* usa diferentes *clusters* para diferentes tipos de eventos para segmentar a grande quantidade de dados com a ajuda do *mirror maker*.

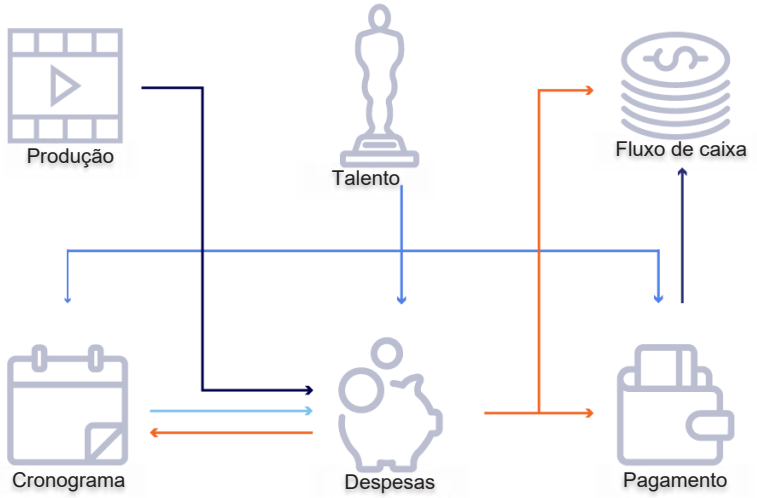
Atualmente, o *Shopify* implantou os dados de seus clientes na nuvem e usa o Kubernetes para garantir que a disponibilidade do *Apache Kafka* não seja prejudicada.

3.1. *Apache Kafka* no *Netflix Studio*

Estudo de caso retirado de (CONFLUENT, 2020).

Do ponto de vista da engenharia, todo aplicativo financeiro **é** modelado e implementado como um microsserviço. A Netflix adota a governança distribuída e incentiva uma abordagem de aplicativos orientada por microsserviços, o que ajuda a alcançar o equilíbrio certo entre a abstração e a velocidade dos dados conforme a empresa cresce. Em um mundo simples, os serviços podem interagir muito bem via HTTP, mas **à** medida que expandimos, eles evoluem para um gráfico complexo de interações síncronas e baseadas em solicitações que podem potencialmente levar a um cérebro/estado dividido e interromper a disponibilidade. A Figura 23 ilustra o modelo orientado por microsserviços.

Figura 23. Modelo orientado por microsserviços da *Netflix*



Fonte: Confluent (2020).

Considere na figura acima as entidades relacionadas, uma alteração na data de produção de um programa. Isso afeta a nossa programação, que por sua vez influencia projetos de fluxo de caixa, pagamentos de talentos, orçamentos para o ano, etc. Geralmente, em uma arquitetura de microsserviço, uma porcentagem de falha **é** aceitável. No entanto, uma

falha em qualquer uma das chamadas de microsserviço para Engenharia de Finanças de Conteúdo levaria a uma infinidade de cálculos fora de sincronia e poderia resultar em dados perdidos em milhões de dólares. Isso também levaria a problemas de disponibilidade, pois o gráfico de chamadas se expandia e causaria pontos cegos ao tentar rastrear e responder efetivamente a perguntas comerciais, como: por que as projeções de fluxo de caixa se desviam do cronograma de lançamento? Por que a previsão para o ano atual não leva em consideração os programas em desenvolvimento ativo? Quando podemos esperar que nossos relatórios de custos reflitam com precisão as mudanças *upstream*?

Repensar as interações de serviço como fluxos de trocas de eventos – em oposição a uma sequência de solicitações síncronas – permite criar uma infraestrutura inerentemente assíncrona. Ele promove a dissociação e fornece rastreabilidade como cidadão de primeira classe em uma rede de transações distribuídas. Eventos são muito mais do que gatilhos e atualizações. Eles se tornam o fluxo imutável a partir do qual podemos reconstruir todo o estado do sistema.

A mudança para um modelo de publicação/assinatura permite que todos os serviços publiquem suas alterações como eventos em um barramento de mensagens, que pode ser consumido por outro serviço de interesse que precisa ajustar seu estado do mundo. Esse modelo nos permite rastrear se os serviços estão sincronizados com relação **às** alterações de estado e, se não, quanto tempo antes eles podem estar sincronizados. Essas informações são extremamente poderosas ao operar um grande gráfico de serviços dependentes. A comunicação baseada em eventos e o consumo descentralizado nos ajudam a superar problemas que normalmente vemos em grandes gráficos de chamadas síncronas (como mencionado acima).

A *Netflix* adota o *Apache Kafka*® como o padrão de fato para suas necessidades de eventos, mensagens e processamento de fluxo. Kafka atua como uma ponte para todas as comunicações ponto a ponto e no *Netflix Studio*. Ele nos fornece a alta durabilidade e a arquitetura multilocação escalonável e linearmente necessária para os sistemas operacionais da *Netflix*. A oferta interna de serviço *Kafka* oferece tolerância a falhas, observabilidade, implantações em várias regiões e autoatendimento. Isso facilita para todo o ecossistema de microsserviços produzir e consumir facilmente eventos significativos e liberar o poder da comunicação assíncrona.

Uma troca típica de mensagens no ecossistema do *Netflix Studio* **é** apresentada na Figura 24.

Figura 24. Troca de mensagens do *Netflix Studio*

Fonte: Confluent (2020).

Podemos dividi-los em três principais subcomponentes.

3.1.1. Produtores

Um produtor pode ser qualquer sistema que queira publicar todo o seu estado ou sugerir que uma parte crítica do seu estado interno foi alterada para uma entidade específica. Além da carga, um evento precisa aderir a um formato normalizado, o que facilita o rastreamento e a compreensão. Este formato inclui:

- **UUID:** identificador universalmente **único**.
- **Tipo:** um dos tipos de criação, leitura, atualização ou exclusão (CRUD).
- **Ts:** registro de data e hora do evento.

As ferramentas de captura de dados de alteração (CDC) são outra categoria de produtores de eventos que derivam eventos de alterações no banco de dados. Isso pode ser **útil** quando você deseja disponibilizar alterações no banco de dados para vários consumidores. O benefício do uso do CDC **é** que ele não impõe carga adicional no aplicativo de origem.

Para eventos do CDC, o *TYPEcampo* no formato de evento facilita a adaptação e a transformação de eventos, conforme exigido pelos respectivos coletores.

3.1.2. Enriquecedores

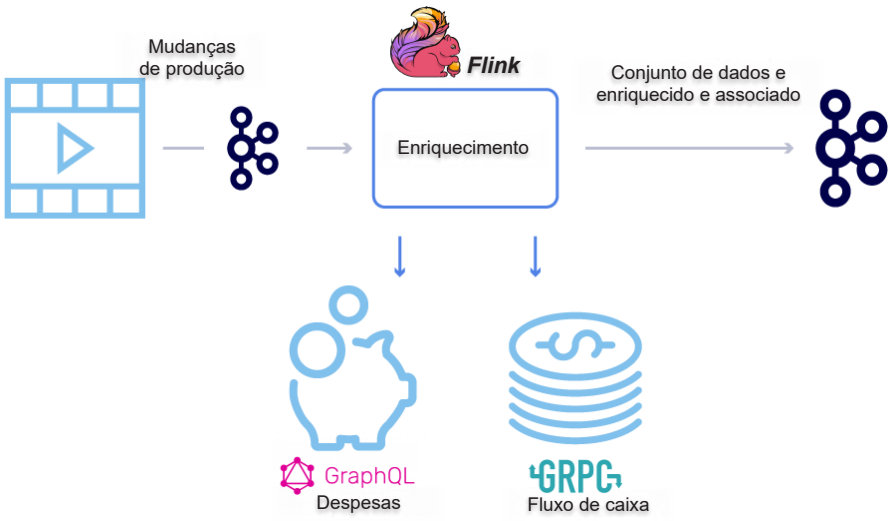
Quando os dados existem no *Kafka*, vários padrões de consumo podem ser aplicados a eles. Os eventos são usados de várias maneiras, inclusive como gatilhos para cálculos do sistema, transferência de carga **útil** para comunicação em tempo quase real e dicas para enriquecer e materializar visualizações de dados na memória.

O enriquecimento de dados está se tornando cada vez mais comum, onde os microsserviços precisam da visualização completa de um conjunto de dados, mas parte dos dados **é** proveniente do conjunto de dados de outro serviço. Um conjunto de dados associado pode ser **útil** para melhorar o desempenho da consulta ou fornecer uma visualização quase em tempo real dos dados agregados. Para enriquecer os dados do evento, os

consumidores leem os dados do *Kafka* e chamam outros serviços (usando métodos que incluem gRPC e GraphQL) para construir o conjunto de dados associado, que depois **é** alimentado com outros tópicos do *Kafka*. A Figura 25 ilustra o enriquecimento de dados da *Netflix*.

O enriquecimento pode ser executado como um microsserviço separado, que **é** responsável por fazer a *fanout* e por materializar conjuntos de dados. Há casos em que se quer fazer um processamento mais complexo, como janelas, sessões e gerenciamento de estado. Para esses casos, **é** recomendável usar um mecanismo de processamento de fluxo maduro no *Kafka* para criar lógica de negócios. Na *Netflix*, **é** o *Apache Flink*® e o RocksDB para processar o fluxo. Também podemos considerar o ksqlDB para propósitos semelhantes.

Figura 25. Enriquecimento de dados *Netflix Studio*



Fonte: Confluent (2020).

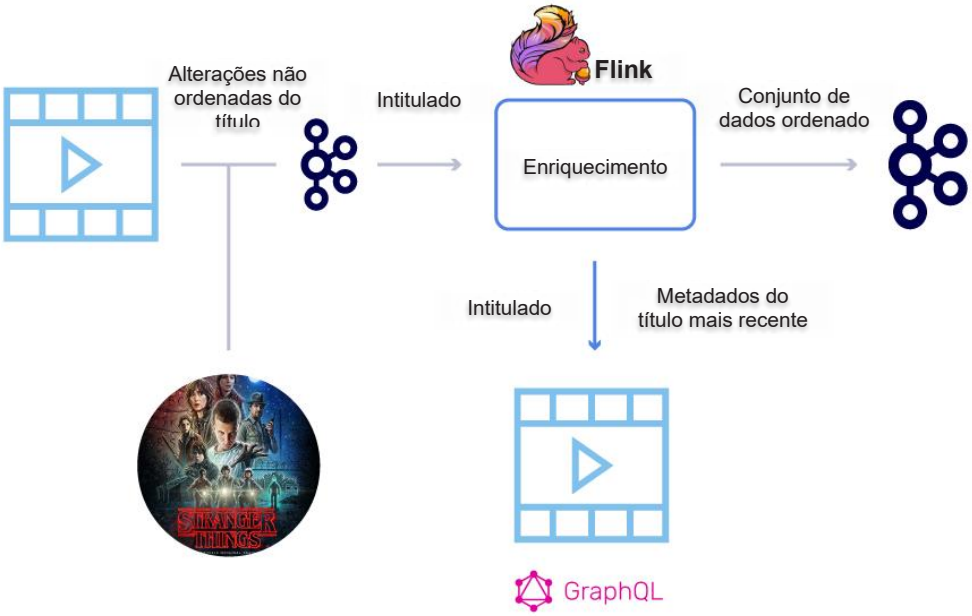
3.1.3. Ordenação de eventos

Um dos principais requisitos em um conjunto de dados financeiros **é** a ordem estrita de eventos. *Kafka* nos ajuda a conseguir isso enviando mensagens com chave. Qualquer evento ou mensagem enviada com a mesma chave terá ordem garantida, pois será enviada para a mesma partição. No entanto, os produtores ainda podem atrapalhar a ordem dos eventos.

Por exemplo, a data de lançamento de *Stranger Things* foi originalmente movida de julho para junho, mas depois de junho para julho. Por vários motivos, esses eventos podem ser gravados na ordem errada para o *Kafka* (tempo-limite da rede quando o produtor tentou acessar o *Kafka*, um *bug* de simultaneidade no código do produtor, etc.).

Para contornar esse cenário, os produtores são incentivados a enviar apenas o ID principal da entidade que foi alterada e não a carga **útil** completa na mensagem *Kafka*. O processo de enriquecimento (descrito na seção acima) consulta o serviço de origem com o ID da entidade para obter o estado/carga **útil** mais atualizado, fornecendo, assim, uma maneira elegante de contornar o problema fora de ordem. Nós nos referimos a isso como **materialização atrasada** e garante conjuntos de dados ordenados. A Figura 26 ilustra a ordenação de eventos para o lançamento de *Stranger Things*.

Figura 26. Ordenação de eventos para *Stranger Things*



Fonte: Confluent (2020).

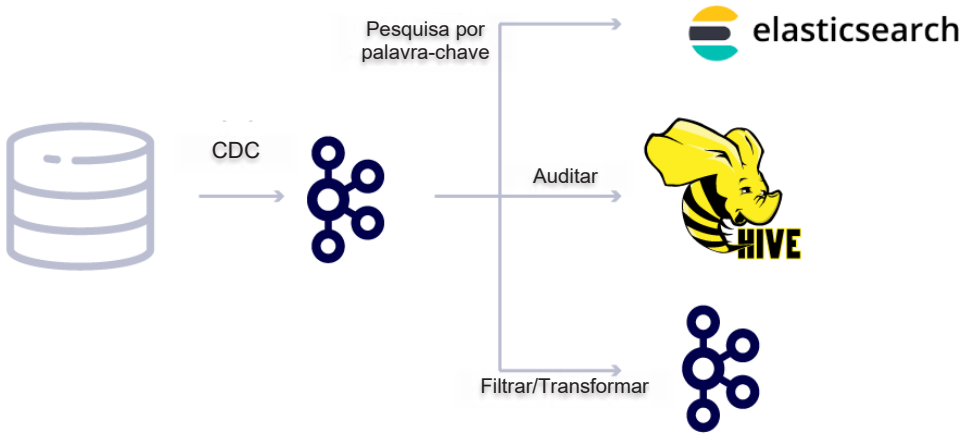
3.1.4. Consumidores

Usamos o *Spring Boot* para implementar muitos dos microsserviços consumidos que são lidos nos tópicos do Kafka. O *Spring Boot* oferece **ótimos** consumidores *Kafka* integrados, chamados *Spring Kafka Connectors*, que tornam o consumo contínuo, oferecendo maneiras fáceis de conectar anotações para consumo e desserialização de dados.

Um aspecto dos dados que ainda não discutimos são os contratos. **À** medida que se expande o uso de fluxos de eventos, acabe-se com um grupo variado de conjuntos de dados, alguns dos quais são consumidos por um grande número de aplicativos. Nesses casos, definir um esquema na saída **é** ideal e ajuda a garantir compatibilidade com versões anteriores. Para fazer isso, pode-se utilizar o *Confluent Schema Registry* e o Apache Avro™ para criar fluxos esquematizados para versionar fluxos de dados.

Além dos consumidores dedicados de microsserviço, também pode ser usado coletores de CDC que indexam os dados em várias lojas para análise posterior. Isso inclui o *Elasticsearch* para pesquisa de palavras-chave, o Apache Hive™ para auditoria e o próprio *Kafka* para processamento posterior. A carga **útil** para esses sumidouros **é** derivada diretamente da mensagem *Kafka*, usando o campo ID como a chave primária e TYPEpara identificar operações CRUD. A Figura 27 ilustra os coletores de CDC.

Figura 27. Coletores de CDC



Fonte: Confluent (2020).

3.1.5. Garantias de entrega de mensagens

Garantir exatamente uma vez que a entrega em um sistema distribuído não **é** trivial devido **às** complexidades envolvidas e **à** infinidade de peças móveis. Os consumidores devem ter um comportamento idempotente para dar conta de qualquer potencial infraestrutura e contratempos do produtor.

Apesar de os aplicativos serem idempotentes, eles não devem repetir operações pesadas de computação para mensagens já processadas. Uma maneira popular de garantir isso **é** acompanhar o UUID das mensagens consumidas por um serviço em um cache distribuído com vencimento razoável (definido com base no SLA). Sempre que o mesmo UUID for encontrado dentro do intervalo de vencimento, o processamento **é** pulado.

O processamento no *Flink* fornece essa garantia usando seu gerenciamento de estado interno baseado no *RocksDB*, com a chave sendo o UUID da mensagem. Se você quiser fazer isso puramente usando o *Kafka*, o *Kafka Streams* também oferece uma maneira de fazer isso. Os aplicativos de consumo baseados no *Spring Boot* usam o *EVCache* para conseguir isso. A Figura 28 ilustra o processo de idempotência usando *Flink* e *EVCache*.

A equipe de produção e finanças do *Netflix Studio* adota a governança distribuída como a maneira de arquitetar sistemas. Eles usam o *Kafka* como plataforma de escolha para trabalhar com eventos, que são uma maneira imutável de registrar e derivar o estado do sistema. O *Kafka* os ajudou a alcançar maiores níveis de visibilidade e dissociação em nossa infraestrutura, além de ajudar a dimensionar organicamente as operações. Está no centro da revolução da infraestrutura do *Netflix Studio* e, com ela, da indústria cinematográfica.

Figura 28. Processo de idempotência usando *Flink* e *EVCache*

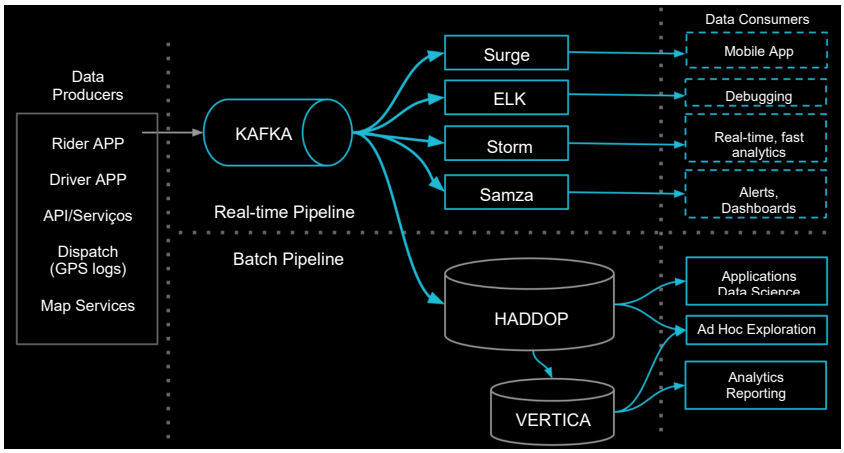
Fonte: Confluent (2020).

3.2. Apache Kafka no Uber

Estudo de caso retirado de (SOMAN; et al., 2016).

No *Uber*, o *Apache Kafka* **é** usado como um barramento de mensagens para conectar diferentes partes do ecossistema. São coletados *logs* do sistema e de aplicativos, bem como dados de eventos dos aplicativos de piloto e motorista. Em seguida, são disponibilizado esses dados para uma variedade de consumidores a jusante via *Kafka*. A Figura 29 apresenta o ecossistema *Kafka* do *Uber*.

Figura 29. Ecossistema *Kafka Uber*



Fonte: Soman *et al.* (2016).

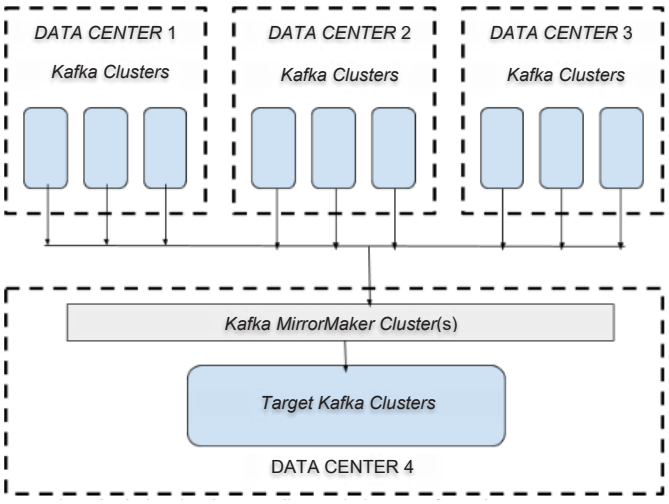
Os dados no *Kafka* alimentam os *pipelines* em tempo real e os *pipelines* em lote. Os dados anteriores são para atividades como computação de métricas de negócios, depuração, alerta e painel. Os dados do *pipeline* em lote são mais exploratórios, como ETL no *Apache Hadoop* e *HP Vertica*.

Nesta seção, descreveremos o **uReplicator, a solução de código aberto do *Uber*** para replicar dados do *Apache Kafka* de maneira robusta e confiável. Esse sistema estende o *design* original do *MirrorMaker* da *Kafka* para se concentrar em confiabilidade extremamente alta, garantia de perda de dados zero e facilidade de operação. Em produção desde novembro de 2015, o *uReplicator* **é** uma peça essencial da infraestrutura de vários data centers da *Uber*.

3.2.1. O que é um *mirror maker* e por que precisamos de um?

Dado o uso em larga escala do *Kafka no Uber*, acaba-se usando vários *clusters* em diferentes *data centers*. Para uma variedade de casos de uso, **é** preciso examinar a visão global desses dados. Por exemplo, para calcular métricas de negócios relacionadas a viagens, **é** preciso coletar informações de todos os *data centers* e analisá-las em um **único** local. Para conseguir isso, historicamente **é** usada a ferramenta *MirrorMaker* de código aberto fornecida com o pacote *Kafka* para replicar dados nos *data centers*, como mostrado na Figura 30.

Figura 30. *Pipeline* de dados do *Uber*



Fonte: Soman *et al.* (2016).

O *MirrorMaker* **é** bastante simples. Ele usa um consumidor *Kafka* de alto nível para buscar os dados do *cluster* de origem e, em seguida, alimenta esses dados em um produtor Kafka para despejá-los no cluster de destino.

3.2.2. Limitações do *MirrorMaker* de *Kafka* no *Uber*

Embora a configuração original do *MirrorMaker* tenha começado o suficiente, logo foi encontrado problemas de escalabilidade. **À** medida que o número de tópicos e a taxa de dados (bytes/segundo) aumentavam, começamos a ver a entrega de dados atrasada ou a perda completa de dados entrando no cluster agregado, resultando em problemas de produção e reduzindo a qualidade dos dados. Alguns dos principais problemas com a ferramenta *MirrorMaker* existente para casos de uso específicos da *Uber* estão listados abaixo:

» **Rebalanceamento caro.** Como mencionado anteriormente, cada trabalhador do *MirrorMaker* usa um consumidor de alto nível. Esses consumidores geralmente passam por um processo de reequilíbrio. Eles negociam entre si para decidir quem será o proprietário de qual partição de tópico (feita via *Apache Zookeeper*). Esse processo pode demorar muito; observamos de 5 a 10 minutos de inatividade em determinadas situações. Isso **é** um problema, pois viola nossa garantia de latência de ponta a ponta. Além disso, os consumidores podem desistir após 32 tentativas de reequilíbrio e ficar presos para sempre. Infelizmente, vimos isso acontecer em primeira mão algumas vezes. Após cada tentativa de reequilíbrio, vimos um padrão de tráfego semelhante ao que ilustra a Figura 31.

Figura 31. Problemas de inatividade com o *Kafka MirrorMaker*

Fonte: Soman *et al.* (2016).

Após a inatividade durante o reequilíbrio, o *MirrorMaker* teve um enorme estoque de dados com os quais teve que acompanhar. Isso resultou em um pico de tráfego no *cluster* de destino e, posteriormente, em todos os consumidores *downstream*, resultando em interrupções na produção e no aumento da latência de ponta a ponta.

» **Dificuldade em adicionar tópicos.** No *Uber*, **é** preciso especificar uma lista de permissões de tópicos na ferramenta de espelhamento para controlar a quantidade de dados que flui pelo *link* da WAN. Com o *Kafka MirrorMaker*, essa lista de permissões era completamente estática e era preciso reiniciar o *cluster MirrorMaker* para adicionar novos tópicos. A reinicialização **é** cara, pois força os consumidores de alto nível a se reequilibrarem. Isso se tornou um pesadelo operacional!

» **Possível perda de dados.** O antigo *MirrorMaker* tinha um problema - parece estar corrigido na versão mais recente – com a confirmação automática de deslocamento que poderia resultar em perda de dados. O consumidor de alto nível comprometeu automaticamente as compensações das mensagens buscadas. Se ocorrer uma falha antes que o *MirrorMaker* possa verificar se realmente gravou as mensagens no *cluster* de destino, essas mensagens serão perdidas.

» **Problemas de sincronização de metadados.** Também encontramos um problema operacional com a atualização da configuração. Para adicionar ou excluir tópicos da lista de permissões, listamos todos os nomes finais de tópicos em um arquivo de configuração, que foi lido durante a inicialização do *MirrorMaker*. **Às** vezes, a configuração falhou ao atualizar em um dos nós. Isso derrubou todo o *cluster*, pois os vários funcionários do *MirrorMaker* não concordaram com a lista de tópicos a serem replicados.

3.2.3. Por que o *uReplicator* foi desenvolvido?

Consideramos as seguintes alternativas para solucionar os problemas mencionados:

» **Divida em vários clusters MirrorMaker.** A maioria dos problemas listados acima resultou do processo de reequilíbrio do consumidor de alto nível. Uma maneira de reduzir seu impacto **é** restringir o número de partições de tópicos replicadas por um *cluster MirrorMaker*. Assim, acabaríamos com vários *clusters* do *MirrorMaker*, cada um replicando um subconjunto dos tópicos a serem agregados.

Prós:

› Adicionar novos tópicos **é** fácil. Basta criar um novo *cluster*.

› A reinicialização do *cluster MirrorMaker* ocorre rapidamente.

Contras:

› **É** outro pesadelo operacional: temos que implantar e manter vários *clusters*.

» **Use o *Apache Samza* para replicação.** Como o problema está no consumidor de alto nível, uma solução **é** usar o *Kafka SimpleConsumer* e adicionar as peças que faltam na eleição do líder e na atribuiç**ão** de partição. O *Apache Samza* **é** uma estrutura de processamento de fluxo, já atribui estaticamente partições aos trabalhadores. Podemos então simplesmente usar um trabalho do Samza para replicar e agregar dados ao destino.

Prós:

› **É** altamente estável e confiável.

› **É** fácil de manter. Podemos replicar muitos tópicos usando um trabalho.

› A reinicialização do trabalho tem um impacto mínimo no tráfego de replicação.

Contras:

› Ainda está muito estático. Precisamos reiniciar o trabalho para adicionar e/ou excluir tópicos.

› Precisamos reiniciar o trabalho para adicionar mais trabalhadores (a partir do Samza 0.9).

› A expansão de tópicos precisa ser explicitamente tratada.

» **Use um consumidor Kafka baseado no Apache Helix.** Por fim, foi decidido usar um consumidor *Kafka* baseado em *Helix*. Nesse caso, estamos usando o *Apache Helix* para atribuir partições aos trabalhadores, e cada trabalhador usa o *SimpleConsumer* para replicar dados.

Prós:

› Adicionar e excluir tópicos **é** muito fácil.

› Adicionar e excluir nós ao *cluster MirrorMaker* **é** muito fácil.

› Nunca precisamos reiniciar o *cluster* por um motivo operacional (apenas para atualizações).

› **É** altamente confiável e estável.

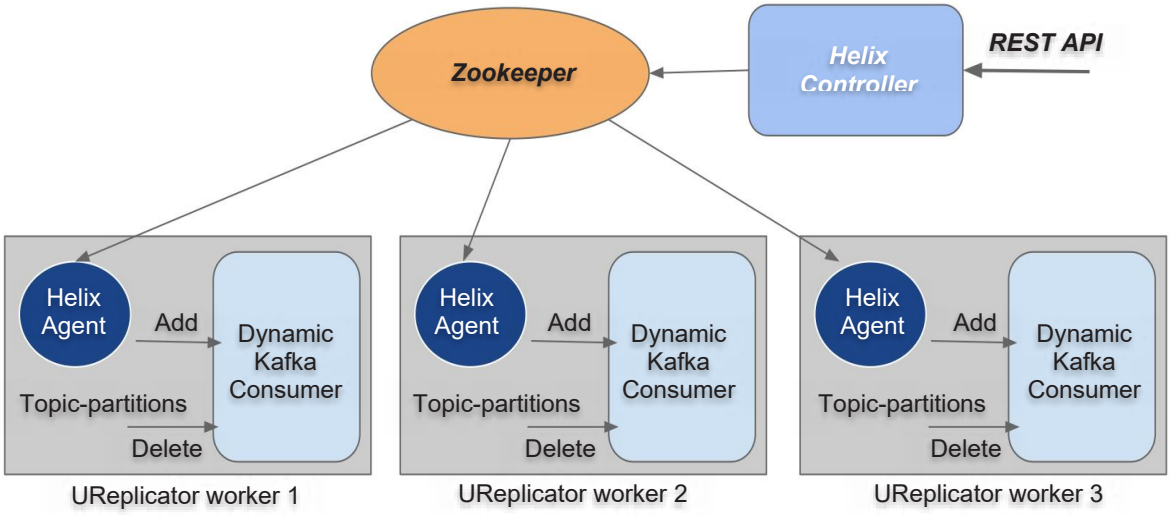
Contras:

› Isso introduz uma dependência do *Helix*. (Isso **é** bom, pois o *Helix* em si **é** muito estável e podemos usar um cluster *Helix* para vários clusters *MirrorMaker*).

3.2.4. Visão geral do *uReplicator*

A Figura 32 apresenta a visão geral do *uReplicator*.

Figura 32. *uReplicator*



Fonte: Soman *et al.* (2016).

Os vários componentes do *uReplicator* funcionam de maneiras diferentes em relação **à** confiabilidade e estabilidade:

1. O **controlador *Helix uReplicator*:** na verdade um *cluster* de nós, tem várias responsabilidades:

   › Distribuir e atribuir partições de tópicos a cada processo de trabalho.

   › Lidar com adição / exclusão de tópicos/partições.

   › Lidar com adição / exclusão de trabalhadores do *uReplicator*.

   › Detectar falhas do nó e redistribuir essas partições de tópicos específicas.

   O controlador usa o *Zookeeper* para realizar todas essas tarefas. Ele também expõe uma API REST simples para adicionar/excluir/modificar tópicos a serem espelhados.

2. Um **trabalhador do *uReplicator*:** semelhante a um processo de trabalho no recurso de espelhamento do *Kafka*, replica determinado conjunto de partições de tópicos do *cluster* de origem para o cluster de destino. Em vez de um processo de reequilíbrio, o controlador *uReplicator* determina a atribuição do *uReplicator*. Além disso, em vez de usar o consumidor de alto nível *Kafka*, usamos uma versão simplificada chamada *DynamicKafkaConsumer*.

3. Um **agente Helix:** para cada trabalhador do uReplicator **é** notificado sempre que houver uma alteração (adição / exclusão de partições de tópicos). Por sua vez, ele notifica o DynamicKafkaConsumer para adicionar / remover partições de tópicos.

4. Existe **uma** instância ***DynamicKafkaConsumer*:** que **é** uma modificação do consumidor de alto nível, em cada trabalhador do *uReplicator*. Ele remove a parte de reequilíbrio e adiciona um mecanismo para adicionar / excluir partições de tópicos em tempo real.

Por exemplo, digamos que queremos adicionar um novo tópico a um *cluster* existente do *uReplicator*. O fluxo de eventos **é** o seguinte:

» O administrador *Kafka* adiciona o novo tópico ao controlador usando o seguinte comando:

```
$ curl –X POST http://localhost:9000/topics/testTopic
```

» O controlador *uReplicator* calcula o número de partições para *testTopic* e mapeia partições de tópicos para trabalhadores ativos. Em seguida, ele atualiza os metadados do *Zookeeper* para refletir esse mapeamento.

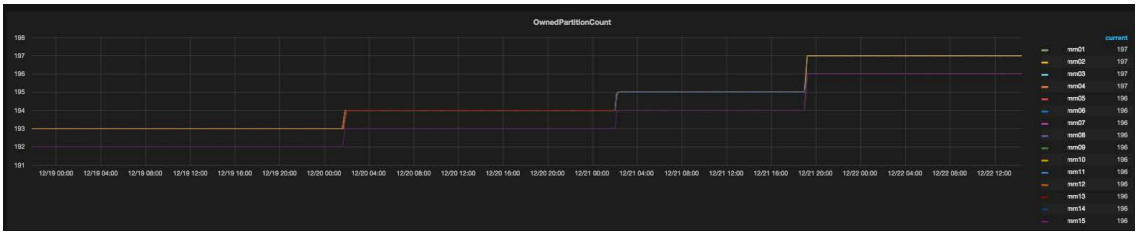
» Cada agente *Helix* correspondente recebe um retorno de chamada com notificação da adição dessas partições de tópicos. Por sua vez, este agente invoca o *addFetcherForPartitions* função de *DynamicKafkaConsumer*.

» O *DynamicKafkaConsumer* posteriormente registra essas novas partições, localiza os corretores líderes correspondentes e os adiciona aos *threads* de busca para iniciar o espelhamento de dados.

3.2.5. Impacto na estabilidade geral

Desde o lançamento inicial do *uReplicator* em produção no *Uber*, não foi visto nenhum problema com ele (em contraste com uma interrupção de algum tipo quase toda semana antes de sua implementaç**ão).** O gráfico ilustrado na Figura 33 mostra o cenário de adição de novos tópicos **à** lista de permissões da ferramenta de espelhamento em produção. O primeiro gráfico mostra o total de partições de tópicos pertencentes a cada trabalhador do *uReplicator*. Essa contagem aumenta para cada novo tópico sendo adicionado.

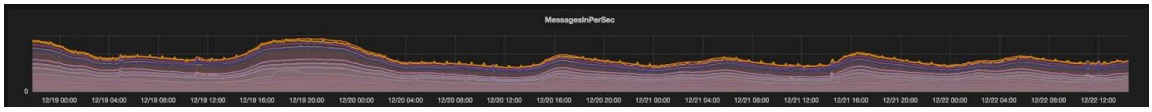
Figura 33. Gráfico 1



Fonte: SOMAN *et al.* (2016).

O segundo gráfico (Figura 34) mostra o tráfego correspondente do *uReplicator* fluindo para o *cluster* de destino. Não há período de inatividade ou picos de carga, como no antigo *Kafka MirrorMaker*:

Figura 34. Gráfico 2



Fonte: Soman et al. (2016).

No geral, os benefícios do *uReplicator* incluem:

» **Estabilidade**: **o** reequilíbrio agora acontece apenas durante a inicialização e quando um nó **é** adicionado ou excluído. Além disso, afeta apenas um subconjunto das partições de tópicos em vez de causar inatividade completa como antes.

- **Escalabilidade mais fácil**: adicionar um novo nó a um *cluster* existente agora **é** muito mais simples. Como a atribuição de partição agora **é** estática, podemos mover inteligentemente apenas um subconjunto de partições para o novo nó. Outras partições de tópicos permanecem inalteradas.
- **Operação mais fácil**: a nova ferramenta de espelhamento do *Uber* suporta lista de permissões dinâmicas. Agora, não precisamos reiniciar o cluster ao adicionar/excluir/expandir tópicos do *Kafka*.
- **Zero perda de dados**: o *uReplicator* garante zero perda de dados, pois confirma pontos de verificação somente após a persistência dos dados no cluster de destino.

Desde o início, o *uReplicator* tem sido uma adição valiosa **à** missão da equipe de plataforma de *streaming* de conectar diferentes partes do ecossistema da *Uber Engineering* por meio de mensagens e do modelo de publicação e assinatura (usando *Kafka*).

REFERÊNCIAS

ALLIED. **7 Surprising Facts About Cloud Computing**. 2015. Disponível em: https://www.alliedtelecom.net/facts-about-cloud-computing/. Acesso em 23 mar. 2020.

BELSHE, M.; PEON, R. **Hypertext Transfer Protocol Version 2**. 2015. Disponível em: https://httpwg.org/specs/rfc7540.html. Acesso em 19 mar. 2020.

CONFLUENT. **Featuring Apache Kafka in the Netflix Studio and Finance World.** 2020. Disponível em: https://www.confluent.io/blog/how-kafka-is-used-by-netflix/. Acesso em 28 mar. 2020.

CONVENTIONAL COMMITS. **Conventional Commits 1.0.0-beta.**2. 2020. Disponível em: https://www.conventionalcommits.org/en/v1.0.0-beta.2/. Acesso em 29 out. 2020.

CROCKFORD, D. **Introducing JSON**. 2020. Disponível em: https://www.json.org/json-en.html. Acesso em 19 mar. 2020.

CRUZ, G. **Como é trabalhar como Desenvolvedor(a) Back-End, por Giovanni Cruz**. 2017. Disponível em: https://medium.com/trainingcenter/como-%C3%A9-trabalhar-como-desenvolvedor-backend-b40255d75626. Acesso em 23 mar. 2020.

DAVIS, D; VENNAM, B**. Knative 101: Exercises designed to help you achieve an understanding of Knative.** 2019. Disponível em: https://developer.ibm.com/tutorials/knative-101-labs/. Acesso em 24 mar. 2020.

DEVMEDIA. **O que é HTML5.** 2012. Disponível em: https://www.devmedia.com.br/o-que-e-o-html5/25820. Acesso em 20 mar. 2020.

ECMA. **ECMAScript Language Specification**. 2018. Disponível em: http://ecma-international.org/ecma-262/9.0/index.html#Title. Acesso em 19 mar. 2020.

FREECODCAMP. **Introduction to NPM Scripts**. 2018. Disponível em: https://www.freecodecamp.org/news/introduction-to-npm-scripts-1dbb2ae01633/. Acesso em 29 out. 2020.

FORSUM, G. **Backend infrastructure at Spotify**. 2013. Disponível em: https://labs.spotify.com/2013/03/15/backend-infrastructure-at-spotify/. Acesso em 21 mar. 2020.

FOWLER, C. O programador apaixonado. Casa do Código, 1º ed, 2014.

GITHUB. 2020. Disponível em: https://github.com/nodejs/node. Acesso em 29 out. 2020.

GRUNT. 2020. Disponível em: https://gruntjs.com/. Acesso em 29 out. 2020.

GULP. 2020. Disponível em: https://gulpjs.com/. Acesso em 29 out. 2020.

HOSTGATOR BRASIL**. Tudo o que você precisa para ser um programador front-end**. 2018. Disponível em: https://www.hostgator.com.br/blog/tudo-o-que-voce-precisa-para-ser-um-programador-front-end/. Acesso em 23 mar. 2020.

IBM. **Cloud Storage**. 2019. Disponível em: https://www.ibm.com/cloud/learn/cloud-storage. Acesso em 25 mar. 2020.

IBM. **Hypervisors**. 2019. Disponível em: https://www.ibm.com/cloud/learn/hypervisors. Acesso em 25 mar. 2020.

IBM. **Istio**. 2019. Disponível em: https://www.ibm.com/cloud/learn/istio. Acesso em 24 mar. 2020.

IBM. **Microservices.** Disponível em: https://www.ibm.com/cloud/learn/microservices. Acesso em 24 mar. 2020.

IBM. **Virtual Machines (VMs)**. 2019. Disponível em: https://www.ibm.com/cloud/learn/virtual-machines. Acesso em 25 mar. 2020.

IBM. **Desktop-as-a-Service (DaaS).** 2018. Disponível em: https://www.ibm.com/cloud/learn/desktop-as-a-service. Acesso em 25 mar. 2020.

IBM. **The state of container-based application development**. 2017. Disponível em: https://www.ibm.com/cloud-computing/info/container-development/. Acesso em 24 mar. 2020.

IBM Cloud. Containerization Explained. 2019. Disponível em: https://www.youtube.com/watch?v=0qotVMX-J5s&feature=emb_logo. Acesso em 29 out. 2020.

IBM Cloud. Kubernetes Explained. 2019. Disponível em: https://www.youtube.com/watch?v=aSrqRSk43lY&feature=emb_logo. Acesso em 29 out. 2020.

IBM Cloud. Implantações Kubernetes: comece rapidamente. 2019. Disponível em: https://www.youtube.com/watch?v=Sulw5ndbE88&feature=emb_logo. Acesso em 29 out. 2020.

IBM Cloud. Virtualization Explained. 2019. Disponível em: https://www.youtube.com/watch?v=FZR0rG3HKIk&feature=emb_logo. Acesso em 29 out. 2020.

IBM Cloud. GPUs: Explained. 2019. Disponível em: https://www.youtube.com/watch?v=LfdK-v0SbGI&feature=emb_logo. Acesso em 29 out. 2020.

IGTI BLOG. **O que faz um Desenvolvedor Full Stack?** 2018. Disponível em: https://www.igti.com.br/blog/o-que-faz-um-desenvolvedor-full-stack/. Acesso em 23 mar. 2020.

IPERIUS BACKUP BRASIL. **Computação em nuvem: Inovações em TI para impulsionar os negócios**. 2019. Disponível em: https://www.iperiusbackup.net/pt-br/computacao-em-nuvem-inovacoes-em-ti-para-impulsionar-os-negocios/. Acesso em: 24 mar. 2020.

KATZ, B. **How Much Does It Cost to Fight in the Streaming Wars?** 2019. Disponível em: https://observer.com/2019/10/netflix-disney-apple-amazon-hbo-max-peacock-content-budgets/. Acesso em 28 mar. 2020.

KIM, J. **Kubernetes Networking: a lab on basic networking concepts**. 2019. Disponível em: https://developer.ibm.com/tutorials/kubernetes-networking-101-lab/. Acesso em 24 mar. 2020.

LINDLEY, C. **Front-End Developer Handbook 2018**. 2018. Disponível em: https://frontendmasters.com/books/front-end-handbook/2018/. Acesso em 18 mar. 2020.

LINTHICUM, D. **The essential guide to software containers for application development**. 2020. Disponível em: https://techbeacon.com/enterprise-it/essential-guide-software-containers-application-development. Acesso em 24 mar. 2020.

MAMP. 2020. Disponível em: https://www.mamp.info/en/windows/. Acesso em 29 out. 2020.

MATTOKA, M. **Are you sure you're using microservices?** 2019. Disponível em: https://blog.softwaremill.com/are-you-sure-youre-using-microservices-f8d4e912d014. Acesso em 28 mar. 2020.

MDN. **CSS Reference**. 2020. Disponível em: https://developer.mozilla.org/en-US/docs/Web/CSS/Reference. Acesso em 19 mar. 2020.

MONTEIRO, C. **Desenvolvimento full stack: o que significa ser um profissional completo**. 2019. Disponível em: https://movile.blog/desenvolvimento-full-stack-o-que-significa-ser-um-profissional-completo/. Acesso em 23 mar. 2020.

NARKNEDE, N; SHAPIRA, G; PALINO, T. **Kafka: The definitive guide: real-time data and streams Processing at scale.** 2017. O'Reilly Media, Inc.

NEEMAN, T. **Debug and log your Kubernetes applications.** 2019. Disponível em: https://developer.ibm.com/tutorials/debug-and-log-your-kubernetes-app/. Acesso em 24 mar. 2020.

NETFLIX. **Evolution of the Netflix Data Pipeline**. 2016. Disponível em: https://netflixtechblog.com/evolution-of-the-netflix-data-pipeline-da246ca36905. Acesso em 28 mar. 2020.

NETFLIX. **Spark and Spark Streaming at Netfix-(Kedar Sedekar and Monal Daxini, Netflix)**. 2015. Disponível em: https://www.slideshare.net/SparkSummit/spark-and-spark-streaming-at-netfix-sedakar-daxini. Acesso em 28 mar. 2020.

NODEJS. 2020. Disponível em: https://nodejs.org. Acesso em 29 out. 2020.

OLIVEIRA, W. **Como é trabalhar como Desenvolvedor Front-End, por Felipe Fialho**. 2017. Disponível em: https://medium.com/trainingcenter/como-%C3%A9-trabalhar-como-desenvolvedor-front-end-por-felipe-fialho-1e3efbadef90. Acesso em 23 mar. 2020.

PAGANI, T. **Documentos acessíveis com WAI-ARIA em HTML5.** 2011. Disponível em: https://tableless.com.br/documentos-acessiveis-com-aria-em-html5/. Acesso em 20 mar. 2020.

PARIS-WHITE, D. **Scale security while innovating microservices fast.** 2018. Disponível em: https://www.ibm.com/cloud/blog/scale-security-innovating-microservices-fast. Acesso em 24 mar. 2020.

PINTEREST. 2020. Disponível em: https://i.pinimg.com/564x/57/b8/05/57b805a0f624c0837b10e0c4cc5b1c55.jpg. Acesso em 29 out 2020.

REDMONK. **The Continued Rise of Apache Kafka.** 2017. Disponível em: https://redmonk.com/fryan/2017/05/07/the-continued-rise-of-apache-kafka/. Acesso em 28 mar. 2020.

SHAPLAND, R; COLE, B. **What are cloud containers and how do they work?** 2019. Disponível em: https://searchcloudsecurity.techtarget.com/feature/Cloud-containers-what-they-are-and-how-they-work. Acesso em 24 mar. 2020.

SOMAN, C. *et al.* **uReplicator: Uber Engineering's Robust Apache Kafka Replicator**. 2016. Disponível em: https://eng.uber.com/ureplicator-apache-kafka-replicator/. Acesso em 29 mar. 2020.

STORYBOOK. Build bulletproof UI components faster. 2020. Disponível em: https://storybook.js.org/. Acesso em 29 out. 2020.

TAKE BLOG. **Como se tornar um desenvolvedor back-end? 4 dicas para começar a sua carreira**. 2019. Disponível em: https://take.net/blog/devs/desenvolvedor-back-end/. Acesso em 23 mar. 2020.

TUTORIALSTEACHER.COM. **JavaScript Tutorial**. 2020. Disponível em: https://www.tutorialsteacher.com/javascript/javascript-tutorials. Acesso em 29 out. 2020.

TUTORIALSTEACHER.COM. **Node.js Process Model**. 2020. Disponível em: https://www.tutorialsteacher.com/nodejs/nodejs-process-model. Acesso em 26 mar. 2020.

VENNAM, S. **Kubernetes Clusters: Architecture for Rapid, Controlled Cloud App Delivery.** 2019. Disponível em: https://www.ibm.com/cloud/blog/new-builders/kubernetes-clusters-architecture-for-rapid-controlled-cloud-app-delivery. Acesso em 24 mar. 2020.

WACHAL, M. **Digital transformation with streaming application development.** 2019. Disponível em: https://blog.softwaremill.com/digital-transformation-with-streaming-application-development-100fb4189149. Acesso em 28 mar. 2020.

WAMP SERVER. 2020. Disponível em: https://www.wampserver.com/en/. Acesso em 29 out. 2020.

WARSKI, A. **Evaluating persistent, replicated message queues.** 2017. Disponível em: https://softwaremill.com/mqperf/. Acesso em 28 mar. 2020.

WARSKI, A. **Event sourcing using Kafka.** 2018. Disponível em: https://blog.softwaremill.com/event-sourcing-using-kafka-53dfd72ad45d. Acesso em 28 mar. 2020.

WEBPACK. 2020. Disponível em: <https://webpack.js.org/. Acesso em 29 out. 2020.

WHATWG. **DOM Living Standard**. 2020. Disponível em: https://dom.spec.whatwg.org/. Acesso em 19 mar. 2020.

WHATWG. **HTTP Living Standard**. 2020. Disponível em: https://html.spec.whatwg.org/multipage/introduction.html#introduction. Acesso em 19 mar. 2020.

WHATWG. **URL Living Standard**. 2020. Disponível em: https://url.spec.whatwg.org/. Acesso em 19 mar. 2020.

WODEHOUSE, C. **The role of a front-end web developer: creating user experience & interactivity**. 2015. Disponível em: https://www.upwork.com/hiring/development/front-end-developer/. Acesso em 18 mar. 2020.

W3C BRASIL. **Cartilha de Acessibilidade na Web.** 2020. Disponível em: https://ceweb.br/cartilhas/cartilha-w3cbr-acessibilidade-web-fasciculo-I.html. Acesso em 20 mar. 2020.

W3 SCHOOLS. **Cloud Computing Architecture**. 2020. Disponível em: https://www.w3schools.in/cloud-computing/cloud-computing-architecture/. Acesso em 23 mar. 2020.

W3TECHS. **Usage statistics of PHP for websites**. 2020. Disponível em: https://w3techs.com/technologies/details/pl-php. Acesso em 21 mar. 2020.

WOODIE, A. **The Real-Time Rise of Apacha Kafka.** 2016. Disponível em: https://www.datanami.com/2016/04/06/real-time-rise-apache-kafka/. Acesso em 28 mar. 2020.