Avaliação experimental dos Brokers Kafka e Apache Flume no Contexto de Big Data

Matheus Orlandi de Castro[1], Cristiano Bertolini[1], Evandro Preuss[1]

[1]Universidade Federal de Santa Maria - UFSM
Centro de Educação Superior Norte - CESNORS, Frederico Westphalen, RS

mo.castro@hotmail.com, cristiano.bertolini@ufsm.br,
evandro.preuss@gmail.com

Abstract. *The data volume is getting bigger and different solutions have been proposed to deal with a large volume of data. This paper presents a performance evaluation of the Kafka and Flume brokers in the context of Big Data. We proposed a benchmark to evaluate the brokers with simulated data and a real application. When we use previous stored data the broker Flume performed better than Kafka and when the streaming is used to generate data Kafka has a better performance.*

Resumo. *Com o grande crescimento do volume de dados gerados, diversas soluções estão sendo criadas para que se consiga tratar, de maneira eficiente, este grande volume de informação. Este artigo apresenta uma proposta para análise de desempenho dos brokers kafka e Flume no contexto de Big Data. Foi realizado um benchmark com dados simulados e uma aplicação real. Observou-se que quando o cenário é composto por dados previamente armazenados, o Flume é mais rápido, já quando se trata de streaming de dados em tempo real o Kafka possui um desempenho superior.*

1. Introdução

No início da era digital, a quantidade de dados gerados era muito pequena e o armazenamento de dados localmente suportava as demandas de então. Com o passar do tempo a necessidade de uma maior capacidade computacional se tornou necessária, surgindo novos conceitos de armazenamento e processamento, na qual se destaca a ideia de paralelismo, computação e armazenamento de dados distribuídos. Este modelo de arquitetura é baseado no uso de *clusters* em que cada máquina tem seu próprio processador, disco, etc, várias máquinas trabalham em conjunto para realizar as atividades de computação e armazenamento demandadas [Chen et al. 2014].

Com o grande crescimento do volume de dados, desde a sua geração, passando pela aquisição, análise e armazenamento, uma nova área de pesquisa chamada *Big Data* está se consolidando. Esta área está tendo uma atenção especial da comunidade acadêmica, governamental e também do ambiente empresarial devido a vasta possibilidade da extração de informações valiosas para os mais diversos domínios de aplicação. Isso é possível através da análise de grandes quantidades de dados, que se usada de forma adequada, pode trazer lucros para uma determinada corporação ou uma melhora significativa na gestão de recursos governamentais, bem como uma gama de novas sub áreas para pesquisas e implementações de novas soluções no presente e futuro.

De acordo com Gantz e Reinsel, até o ano de 2011 o volume de dados criados e copiados aumentou nove vezes nos cinco anos anteriores, representando um total de 1,8ZB (Zettabyte) [Gantz and Reinsel 2011]. No presente, com o crescimento exponencial de serviços web e dispositivos móveis, como *smart phones*, estes dados tendem a crescer de forma muito acentuada. Segundo Gantz e Reinsel, espera-se que este volume dobre a cada dois anos até 2020 [Gantz and Reinsel 2012].

Hoje em dia, o termo *Big Data* está relacionado ao rápido crescimento das companhias de serviços para *Internet*. Por exemplo, o Google processa dados de centenas de *petabyte* (PB), Facebook gera mais de 10 PB de dados por mês, Baidu, uma empresa chinesa, dezenas de PB, e Taobao, uma subsidiária do Alibaba gera dados de dezenas de *Terabyte* (TB) para o comércio *on-line* por dia [Chen et al. 2014].

Observando essa demanda o presente trabalho teve como objetivo principal realizar uma análise de desempenho de dois *brokers*, Apache Kafka e Apache Flume, por meio de *benchmarks*. Esses *softwares* estão enquadrados na fase de aquisição, eles são responsáveis por escalonar o envio de dados por meio de mensagens e entregar ao seu destinatário. No contexto de *Big Data*, os dados podem ser adquiridos pro meio de arquivos de *log*, base de dados ou *streaming*, sendo fundamental o uso de *brokers* para a coleta destes dados, para posterior processamento, análise e armazenamento.

Além disso foi desenvolvido um protótipo de um *software* capaz de simular um fluxo grande de dados e também realizar trocas de mensagens entre *brokers*, para que se possa fazer uma avaliação deles. Para isso foi necessário realizar a configuração dos *brokers* além da instalação do serviço Hadoop, que é uma estrutura voltada para ambientes distribuídos que permite realizar o armazenamento de grandes conjuntos de dados estruturados e não estruturados. Como os *brokers* são apenas intermediários, o Hadoop é necessário para que se consiga implementar o ambiente esperado para a realização do trabalho.

Para este trabalho, os dois *brokers* avaliados foram o Apache Kafka e Apache Flume, os motivos para esta escolha se dá pelo fato de que ambos são *open source* e são amplamente utilizados em grandes provedores de serviços, como o LinkedIn, Amazon, Netflix, etc. Entretanto, o principal motivo é o de definir o melhor *broker* para o projeto S.M.A.R.T., onde foi proposto um modelo para implementar uma estrutura modular para *Big Data*, voltado para pequenas e médias empresas, que visa simplificar a implantação de serviços nesta área para esta faixa empresarial [dos Anjos et al. 2015].

Inicialmente foi realizado um estudo teórico, que envolve os conceitos básicos de *Big Data*, sua diferença em comparação com outros conjuntos de dados e também suas características chaves. Também foi estudado o ambiente em que esta tecnologia está empregada, como a computação em nuvem e como elas interagem. Além disso, a configuração do ambiente necessário para a realização de experimentos iniciais e do protótipo a ser implementado no decorrer do TGSI.

Com isso, o presente artigo está organizado da seguinte forma: a Seção 2 apresenta alguns trabalhos relacionados, visando compor o estado da arte. Na Seção 3 é apresentado o referencial teórico, abordando conceitos de *Big Data*, *Cloud Computing* e também, dos *softwares* necessários para a realização do trabalho. Na Seção 4 é detalhada a metodologia para a realização das avaliações. Posteriormente, na Seção 5 será descrito os experimen-

tos executados e quais os resultados obtidos. Encerrando o artigo, são apresentadas as considerações finais e as referências utilizadas.

2. Trabalhos Relacionados

Nesta seção serão abordados alguns trabalhos que serviram como base para o estudo realizado. Serão apresentadas as principais características de trabalhos semelhantes, realizados por outros autores.

2.1. Throughput Performance of Java Messaging Services Using WebsphereMQ

Henjes [Henjes et al. 2006] apresenta um estudo em que investiga o desempenho de vazão do *Message Broker* WebSphereMQ, para isso foi considerado diferentes números de *subscribers*, *publishers*, mensagens de diferentes tamanhos, diferentes tipos de filtros e estes filtros de diferentes complexidades. Apartir disso foi proposto um modelo matemático que aproxima os resultados da medição, em que foi útil para a predição da taxa de transferência do servidor na prática, que depende fortemente de uma aplicação ou cenário específico.

Para a realização dos testes, o ambiente utilizado consiste em cinco computadores, sendo que quatro deles são máquinas de produção e um é utilizado para fins de controle, por exemplo, controlar trabalhos como a criação de cenários de testes e iniciar a execução. Para a execução do ambiente esperado, foi instalado o *Java SDK 1.4.0*, na sua configuração padrão. Nas experiências uma máquina é usada como um servidor JMS (*Java Message Service*) dedicado e os *publishers* executados em um ou dois computadores exclusivos e os *subscribers* também em ou dois computadores exclusivos.

Após os experimentos, Henjes [Henjes et al. 2006] constataram que pelo menos cinco *subscribers* e 5 *publishers* são necessários para utilizar totalmente a CPU do servidor e com isso conseguir a vazão máxima das mensagens, também que o tamanho da mensagem tem um impacto significativo sobre a vazão de dados no servidor. Outros pontos observados pelos autores é de que o número de tópicos dificilmente influencia na capacidade do servidor e que filtros diferentes impõem o mesmo esforço de filtragem no servidor JMS.

O trabalho proposto e o exposto a cima tem objetivos semelhantes, entretanto o apresentado nesta seção, realiza uma análise de desempenho de apenas um *broker*, o WebsphereMQ da IBM. Como diferencial, o trabalho proposto realizou esta análise entre dois *brokers* distintos trazendo como resultados em quais ambientes cada um tem maior desempenho.

2.2. Analysis of Real Time Stream Processing Systems Considering Latency

Córdova [Córdova 2014] realizou um estudo com o objetivo de avaliar duas das plataformas de código aberto mais notáveis na área de processamento de *Big Data*, Storm e Spark. Segundo o autor do estudo, ambos os sistemas oferecem capacidades de processamento em tempo real através de *micro-batches*, mas o funcionamento dos mecanismos, respostas de falhas, velocidade de processamento e muitas outras características são diferentes. Seu principal objetivo com este estudo, foi a de fornecer uma visão geral de muitas características de alto nível de ambos os sistemas, para poder compreender e avaliar as suas velocidades de processamento.

Para efetuar esta análise Córdova [Córdova 2014] realizou dois experimentos para ter uma visão mais ampla de como os sistemas se comportam. O primeiro experimento é uma avaliação comparativa dos sistemas, para isso foi utilizado um *cluster* localizado na Universidade de Toronto e executada três tarefas diferentes com dois conjuntos de dados de tamanhos distintos. O segundo experimento foi uma micro análise para determinar onde é gasto mais tempo nos sistemas.

Após a realização dos experimentos, Córdova [Córdova 2014] constatou que com registros pequenos o Storm foi em torno de 40% mais rápido em relação ao Spark. No entanto, com o aumento do tamanho dos registros o Spark conseguiu melhorar seu desempenho, chegando a superar o Storm. Após esta análise Cordova concluiu que ambos os sistemas possuem características muito semelhantes, com isso a escolha de um deles deve ser considerada com cuidado e de acordo com o que se pretende realizar.

A avaliação realizada por Córdova [Córdova 2014] foi executada utilizando serviços que são aplicados na fase de processamento de dados, em contrapartida o trabalho proposto realizou também uma avaliação, comparando dois *brokers* voltados para a etapa de aquisição e transferência de dados.

2.3. The analysis of the performance of RabbitMQ and ActiveMQ

Ionescu [Ionescu 2015] compara a velocidade de processamento para envio e recebimento de mensagens, carga de memória e as plataformas que os *brokers* RabbitMQ e ActiveMQ podem gerenciar. Duas aplicações desenvolvidas em java foram implementadas para que ocorra a simulação de envio e recebimento de mensagens.

Para a realização dos experimentos Ionescu [Ionescu 2015] utilizou duas máquinas virtuais criadas com o aplicativo VMWare, sendo que elas possuem as mesmas características e foram executas uma por vez, para que uma não afete a performance da outra. As máquinas virtuais rodam no sistema operacional Windows 7 x64 no servidor HP ProLiant BL680c G5, equipado com um processador Intel Xeon 7420 rodando a uma frequencia de 2,13GHz com quatro núcleos e 16GB de memória RAM. Para cada um dos *brokers* as duas aplicações foram executadas enviando e recebendo mensagens.

No primeiro teste, que busca verificar a degradação da performance do *broker* com o aumento do tamanho da mensagem, o ActiveMQ obteve um desempenho superior em relação ao RabbitMQ levando em conta o envio de mensagens, já no recebimento, o RabbitMQ conseguiu superar seu rival. Em uma avaliação diferente, que envolve o aumento no numero de clientes enviando mensagens para o *broker*, a performance de ambos teve um decréscimo, entretanto o RabbitMQ conseguiu um desempenho superior.

Com isso Ionescu [Ionescu 2015] concluiu que o *broker* ActiveMQ possui um desempenho superior quando se trata de receber mensagens, em contrapartida o RabbitMQ é mais rápido quando a tarefa a ser executada é a de enviar mensagens para uma aplicação cliente.

Em comparação com o trabalho proposto, ambos possuem muitas semelhanças, entre elas a de avaliação de desempenho entre dois mensageiros, a diferença básica entre os dois trabalhos está exatamente nos dois *brokers*, onde Ionescu [Ionescu 2015] utiliza o RabbitMQ e ActiveMQ e o trabalho proposto utiliza Apache Flume e o Apache Kafka.

3. Referencial Teórico

Nesta seção será apresentado o referencial teórico, nela serão frisados os conceitos relacionados ao âmbito de *Big Data* e *Cloud Computing*, além das tecnologias que foram empregadas no decorrer do TGSI.

3.1. Big Data

Devido a sua popularidade atual e por ser uma área relativamente nova no ramo de TI, sua definição ainda traz algumas divergências na literatura atual, entretanto o conceito dos quatro V's, velocidade, variedade, volume e valor é algo em comum na grande maioria dos estudos e publicações da área.

De acordo com Gantz e Reinsel [Gantz and Reinsel 2011] as tecnologias de *Big data* descrevem uma nova geração de tecnologias e arquiteturas, concebida para extrair economicamente valor a partir de volumes muito grandes de uma ampla variedade de dados, permitindo alta velocidade de captura, descoberta e/ou análise. Esta definição engloba a abordagem dos quatro V's, sendo considerada hoje, a mais completa.

Além disso *Big Data* possui algumas características chave, que juntas são conhecidas como uma cadeia de valor, essa cadeia é composta por quatro pilares principais que são: geração, seguido pela aquisição passando pelo armazenamento e por fim análise dos dados.

Na primeira etapa desta cadeia temos a geração de dados. Uma quantidade enorme de termos de pesquisa, mensagens em fóruns de *internet*, registros de conversas e mensagens de microblog, são gerados. Esses dados estão intimamente relacionado com a vida diária das pessoas, e têm características semelhantes de alto valor e baixa densidade. Tais dados podem não possuir valor individualmente, mas através da exploração de grandes dados acumulados, informações úteis, tais como hábitos e passatempos de usuários podem ser identificados, e com isso se torna possível prever certos comportamentos dos usuários [Chen et al. 2014].

Chegando na fase de aquisição, os dados gerados anteriormente são agregados em informações digitas para posterior armazenamento e análise. Intuitivamente, o processo de aquisição é composto por três sub-etapas, recolha de dados, transmissão de dados, e pré-processamento de dados. Não há uma ordem estrita entre a transmissão e o pré-processamento; Assim, as operações de pré-processamento podem ocorrer antes da transmissão e/ou depois da transmissão [Hu et al. 2014].

Após a aquisição dos dados, estas informações deve ser armazenadas em uma estrutura específica para esta área, com isso um subsistema de armazenamento de dados em uma plataforma de *Big Data*, deve organizar as informações coletadas em um formato conveniente para análise e extração de valor. Para este efeito, o sub-sistema de armazenamento de dados deverá fornecer dois conjuntos de características [Hu et al. 2014]:

- A infraestrutura de armazenamento deve acomodar informações de modo persistente e confiável.
- O subsistema de armazenamento de dados deve fornecer uma solução escalável e uma interface de acesso para consultar e analisar uma grande quantidade de dados.

Na última e mais importante fase da cadeia de valor da *Big Data* temos a análise de dados, ela aborda informações obtidas através de observação, medição, ou experiências

sobre um fenômeno de interesse. O objetivo da análise de dados é extrair o máximo de informação possível pertinente ao assunto em consideração [Hu et al. 2014].

3.2. Cloud Computing

Segundo Mell e Grance [Mell and Grance 2011] *Cloud Computing* é um modelo que permite, acesso ubíquo, conveniente e sob demanda de rede a um agrupamento de recursos compartilhados de computação configuráveis (por exemplo, redes, servidores, armazenamento, aplicações e serviços), que podem ser rapidamente fornecidos e liberados com esforço minimo de gerenciamento ou interação do provedor de serviço.

Ela refere-se tanto a aplicativos entregues como serviços através da *internet* quanto ao *software, hardware* e sistemas. Esses serviços podem ser referidos como Software como um Serviço (SaaS). Alguns fornecedores usam termos como IaaS (Infraestrutura como um Service) e PaaS (Plataforma como um Serviço) para descrever seus produtos [Armbrust et al. 2010].

A *Cloud Computing* possui diversos pontos que à caracteriza e diferencia de outros modelos de sistemas distribuídos, de acordo com [Zhang et al. 2010] os principais são:

- Sem investimento inicial: a computação em nuvem usa um modelo de preços *pay-as-you-go*. Um prestador de serviço não precisa investir em infraestrutura para começar a se beneficiar da *Cloud Computing*. Ele simplesmente aluga recursos da nuvem de acordo com as suas necessidades e paga pelo seu uso.
- Reduzido custo operacional: os recursos em um ambiente em nuvem podem ser rapidamente alocados e desalocados de acordo com a demanda necessária. Assim, um provedor de serviços não precisa de capacidade computacional de acordo com sua carga de pico. Isto proporciona grande economia em vez que os recursos podem ser liberados quando a demanda de serviços é baixa.
- Altamente escalável: provém infraestrutura com grande quantidade de recursos e torna-os facilmente acessível. Um prestador de serviços pode facilmente expandir seu serviço a grandes escalas, a fim de lidar com rápido aumento da exigência.
- Fácil acesso: serviços hospedados na nuvem são geralmente baseados na *web*. Portanto, eles são facilmente acessíveis através de uma variedade de dispositivos com conexões de *internet*, estes dispositivos não só incluem computadores *desktop* e *laptop*, mas também de telefones celulares, etc.
- Reduzido riscos de negócios e despesas de manutenção: por terceirizar o serviço de infraestrutura para a nuvem, um prestador de serviço transfere seus riscos de negócios (tais como falhas de *hardware*) aos fornecedores de infraestrutura, que muitas vezes têm uma melhor experiência e estão melhor equipados para gerir esses riscos. Além disso, um prestador de serviços pode reduzir a manutenção de hardware e os custos de formação de pessoal.

3.3. Sistemas Distribuídos

Segundo [Coulouris et al. 2013] um sistema distribuído é aquele no qual os componentes localizados em computadores interligados em rede se comunicam e coordenam suas ações apenas passando mensagens. Essa definição leva às seguintes características especialmente importantes dos sistemas distribuídos:

- Concorrência: em uma rede de computadores, a execução concorrente de programas é o principal paradigma utilizado. Por exemplo duas máquinas compartilhando recursos como páginas da Web ou arquivos quando necessário. Esta capacidade pode ser ampliada com a adição de mais computadores na rede.
- Relógio global: quando os programas precisam cooperar, eles coordenam suas ações trocando mensagens. Esta coordenação depende de uma noção de tempo compartilhada para que as ações dos programas ocorram. Porém, existem limites para a precisão com a qual os computadores podem sincronizar seus relógios em uma rede.
- Falhas independentes: todos os computadores são passíveis de falhas, tanto de *hardware* quanto de *software*, por isso, é responsabilidade de todo o profissional desta área pensar em soluções para contornar estes problemas, com isso, caso uma máquina falhe, as outras continuam funcionando e fornecendo as funcionalidades esperadas por todo o sistema sem que o usuário perceba o problema.

Além disso, [Coulouris et al. 2013] também observou que os principais desafios desta arquitetura estão relacionados a heterogeneidade dos componentes computacionais e também, a segurança e escalabilidade.

3.4. Apache ZooKeeper

ZooKeeper é um serviço centralizado para manter as informações de configuração, proporcionando sincronização distribuída, e prestação de serviços ao grupo. Todos estes serviços são utilizados de uma forma ou de outra por outras aplicações distribuídas. Cada vez que estes serviços são implementados, uma grande quantidade de trabalho para corrigir erros é inevitável. Mesmo quando feito corretamente, estes serviços se tornam muito frágeis a qualquer mudança necessária, se tornando muito complexos [Apache d].

Um servidor ZooKeeper é uma máquina que mantém uma cópia do estado de todo o sistema e persiste esta informação em arquivos de *log* locais. Um conjunto Hadoop muito grande pode ser suportado por múltiplos servidores ZooKeeper (neste caso, um servidor principal sincroniza os servidores de nível superior). Cada máquina cliente se comunica com um dos servidores ZooKeeper para recuperar e atualizar as informações de sincronização [IBM].

Esta arquitetura permite que o ZooKeeper proporcione um elevado rendimento e com baixa latência, entretanto o tamanho da base de dados que o ZooKeeper pode gerir é limitado pela memória [Hortonworks].

3.5. Apache Hadoop

De acordo com White [White 2012], quando um conjunto de dados supera a capacidade de armazenamento de uma única máquina física, torna-se necessário particionar estes dados em máquinas separadas. Estes sistemas de arquivos que gerenciam o armazenamento através de uma rede de computadores são chamados de sistemas de arquivos distribuídos, por distribuir seus dados em máquinas em uma estrutura de rede se tornam muito mais complexos que sistemas de discos de uma única máquinas, um dos maiores desafios neste padrão está em fazer o sistema tolerar falhas de um determinado nó sem perda de informação.

Com isso foi desenvolvido o Hadoop Ditributed File System (HDFS), ele é um sistema projetado para armazenar, de forma confiável e ágil, grandes conjuntos de dados e transmitir estes dados em alta velocidade para aplicações do usuário. Em um *cluster*, diversos *hosts* ligados diretamente por meio de uma rede, trabalham em conjunto para conseguir realizar estas tarefas [Shvachko et al. 2010].

Em vez de confiar em hardware para proporcionar alta disponibilidade, a biblioteca foi desenvolvida para detectar e tratar falhas na camada de aplicação, de modo fornecer um serviço altamente disponível no topo de um conjunto de computadores, cada um dos quais pode ser propenso a falhas [Kumar 2015].

Um *cluster* HDFS tem dois tipos de nós que operam em um modelo mestre de trabalho: um *namenode* (mestre) e um número de *datanodes* (trabalhadores). O *namenode* gere o *namespace* do sistema de arquivos. Ele mantém a árvore de arquivos e os metadados para todos os arquivos e diretórios na árvore. Esta informação é armazenada persistentemente no disco local na forma de dois arquivos: a imagem do *namespace* e um *log* de edição. O *namenode* também sabe em qual *datanode* em que todos os blocos para um determinado arquivo estão localizados. No entanto, ele não armazena a localização dos blocos persistentemente, porque esta informação é reconstruída a partir dos *datanodes* quando o sistema é iniciado [White 2012]. Sua arquitetura pode ser observada na Figura 1

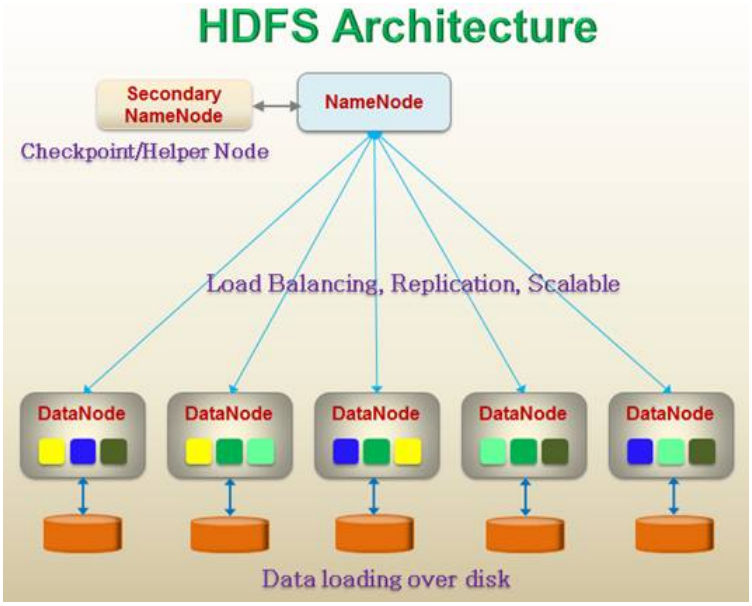

Figura 1. Arquitetura do HDFS [Kumar 2015]

3.6. Brokers

Nesta seção será apresentado os dois *brokers* que foram avaliados neste trabalho, detalhando o funcionamento e a arquitetura de cada um. Além de outras soluções semelhantes a estes.

RabbitMQ: é uma implementação livre e completa do protocolo AMQP (Advanced Message Queuing Protocol), este protocolo foi desenvolvido para sanar problemas de interoperabilidade entre muitas outras soluções de mensagens diferentes,

que foram desenvolvidos por fornecedores diferentes, como IBM, TIBCO e Microsoft [Boschi and Santomaggio 2013].

Além disso, segundo [Jones et al. 2011], é um serviço implementado para realizar o controle do envio e recebimento de mensagens, seu código foi desenvolvido em uma linguagem chamada Erlang e é construído sobre a plataforma Open Telecom, que é um aplicativo voltado para servidores escrita na mesma linguagem. Com a criação de um grupo de *publishers* e *subscribes* que podem acessar os nós de mensagens, consegue-se criar uma rede informações que pode abranger áreas de pequena e grande escala.

ActiveMQ: é um *message broker* desenvolvido em Java, voltado para comunicação remota entre sistemas que qu utilizam a especificação JMS (Java Message Service). Sua API é compatível com diversas linguagens, como C, C++, .NET, Perl, PHP, Python, Ruby, etc [Snyder et al. 2011].

Seu principal objetivo, segundo [Snyder et al. 2011] é fornecer padrões para aplicações orientados a mensagens independente da linguagem utilizada, oferecendo alta disponibilidade, desempenho, escalabilidade, confiabilidade e segurança para ambientes corporativos. Esses requisitos são essencias para que as mensagens enviadas alcancem seus destinatários.

WebsphereMQ: tem como características principais a flexibilidade combinada com confiabilidade, escalabilidade e segurança. Proporcionando um grande número de opções de implementação. Os aplicativos que interagem com esta ferramenta podem ser desenvolvidos em diversas linguagens de programação [Taylor et al. 2012].

3.6.1. Apache Kafka

Apache Kafka é um *software* de mensagens implementado como um transmissor de *logs* distribuído, adequado para consumo *offline* e *online* de mensagens. É um sistema de mensagens inicialmente desenvolvido pelo LinkedIn para coleta e entrega de grandes volumes de dados de eventos e *logs* com baixa latência [Thein 2014]. De acordo com Garg e Nishant [Garg 2013] as principais características que descrevem o Apache Kafka são:

- Mensagens persistentes: Para obter o valor real de *Big Data*, qualquer tipo de perda de informações não pode ser concedido. *Apache Kafka* é desenhado com estruturas de disco que fornecem desempenho em tempo constante, mesmo com grandes volumes de mensagens armazenadas, que está na ordem de TB.
- Alta capacidade: *Kafka* é projetado para suportar milhões de mensagens por segundo com baixa latência.
- Distribuído: *Apache kafka* apoia explicitamente o particionamento de mensagens sobre servidores e distribui o consumo ao longo de um *cluster* de máquinas, mantendo a semântica de ordenação por partição.
- Múltiplo suporte ao cliente: *Apache Kafka* suporta uma fácil integração de clientes de diferentes plataformas, como *Java, .NET , PHP , Ruby e Python*.
- Tempo real: mensagens produzidas pelos segmentos de produtores devem ser imediatamente visíveis para os tópicos de consumo, este recurso é fundamental para sistemas baseados em eventos como sistemas de processamento de eventos complexos.

Foi projetado para permitir que um único *cluster* sirva como a espinha dorsal de dados para uma grande organização. Ele pode ser expandido de forma elástica e transparente sem tempo de inatividade. Os fluxos de dados são divididos e distribuídos por um conjunto de máquinas para permitir uma maior capacidade em relação a uma única máquina [Apache c]. Na Figura 2 pode-se observar o modo de trabalho do *broker*.

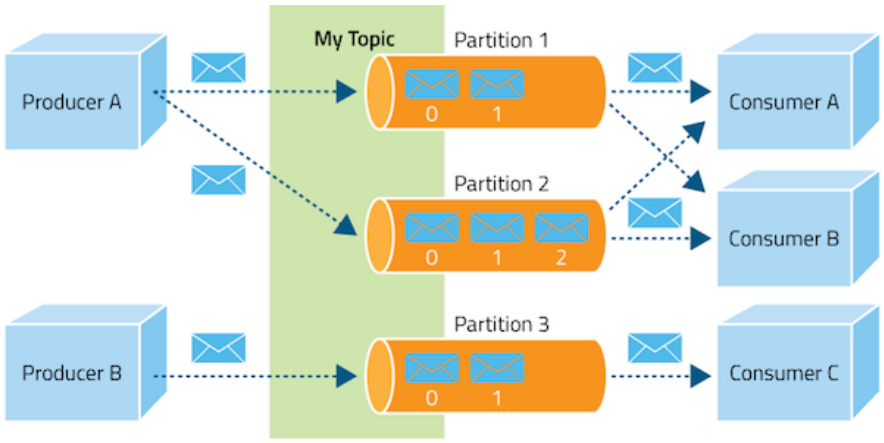

Figura 2. Arquitetura do Apache Kafka [Cloudera 2015]

3.6.2. Apache Flume

Flume é um serviço distribuído para coletar, agregar e mover grandes quantidades de dados de *logs*, de forma eficiente e confiável. Ele tem uma arquitetura simples e flexível voltado para *streaming* de dados, é robusto e tolerante a falhas e possui muitos mecanismos ajustáveis de *failover* e recuperação. Ele usa um modelo de dados simples que permite a aplicação analítica *online* [Apache b].

O uso do *Apache Flume* não se restringe apenas a agregação de dados de *logs*. Seus parâmetros de fonte de dados são configuráveis, com isso ele pode ser usado para o transporte de grandes quantidades de dados de eventos além de dados de tráfego de rede, midias sociais, mensagens de e-mail e praticamente qualquer fonte de dados possível.

A arquitetura do *flume* é muito simples de ser entendida, de acordo com Hoffman [Hoffman 2013], os três principais componentes dele são: as fontes, que são responsáveis por fazer a coleta dos dados que serão transmitidos, após isso, estes dados são transportados por meio de um canal de comunicação e por fim chegam ao *sink*, que é o último componente do *flume* que um determinado dado percorre, chegando neste ponto os arquivos podem ser distribuídos em uma base de dados não relacional, em um sistema de arquivos distribuídos, etc. Seu modo de funcionamento pode ser observado na Figura 3

Esta transferência de dados pelo *flume* é chamado de evento. Um evento é composto de zero ou mais cabeçalhos e um corpo. O cabeçalho possui pares de chave/valor que pode ser usado para tomar decisões de rotas ou carregar outras informações estruturadas, como uma *timestamp* de um evento ou o nome do *host* em que um evento foi originado. Já o corpo é um arranjo de *bytes* que contem os dados propriamente ditos, este arranjo é como uma *string* com codificação UTF-8 [Hoffman 2013].

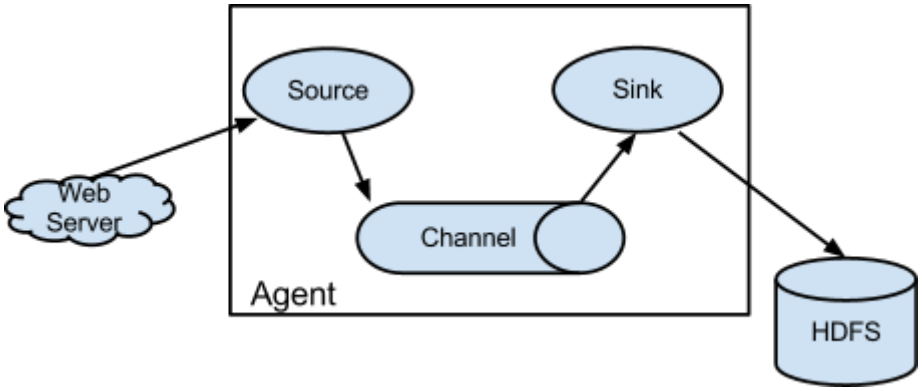

Figura 3. Arquitetura do Apache Flume [Apache b]

4. Desenvolvimento do estudo

Esta seção apresenta a análise de desempenho dos dois *brokers*. Inicialmente foi utilizado *scripts* desenvolvidos em *shell script* que sejam capazes de gerar arquivos de diversos tamanhos e em grandes quantidades. Posteriormente um protótipo de uma aplicação capaz de coletar dados reais e realizar a troca de mensagens entre os *brokers* foi desenvolvido para que se consiga uma análise mais precisa.

Com isso foi necessário realizar a configuração dos *brokers* além da instalação do *software Hadoop*, que é uma estrutura voltada para ambientes distribuídos que permite realizar análises e armazenamento de grandes conjuntos de dados estruturados e não estruturados. Como os *brokers* são apenas intermediários, o Hadoop é necessário para que se consiga implementar o ambiente esperado para a realização da análise de desempenho entre os *brokers*.

A Figura 4 mostra como o protótipo interage com o *broker* Flume. Os dados gerados foram coletados por um ou mais agentes do Flume, sendo que cada agente pode possuir uma ou mais *sources, channels e sinks*.

O *source* foi responsável por coletar os dados gerados pelos *scripts* e pela aplicação e armazenar no canal configurado. O canal utilizado foi a memória principal, por ser mais rápida que as outras possibilidades oferecidas pelo *broker* [Apache a]. Quando terminado o transporte ao longo do flume estes dados foram armazenados no sistema de arquivos do Hadoop, através da configuração do *Sink*.

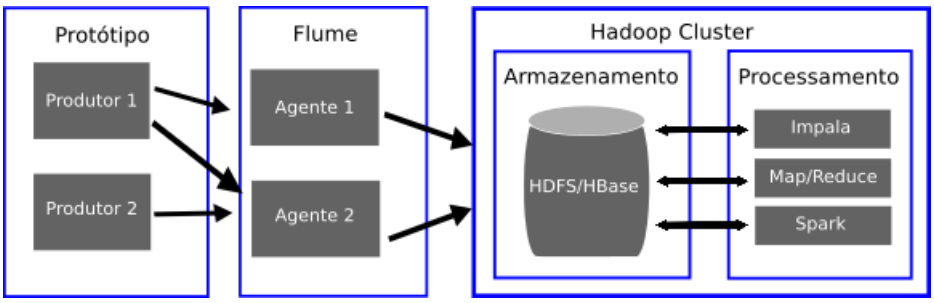

Figura 4. Esquema de interação entre o protótipo, Apache Flume e Hadoop

Da mesma forma, a Figura 5 demonstra como o protótipo interage com o Kafka

com a diferença apenas no funcionamento do *broker*. Considerando que o kafka não é capaz de realizar a coleta dos dados gerados pelos *scripts* e pela aplicação, neste cenário então, o protótipo foi capaz de além de produzir os dados, alimentar o *broker* enviando mensagens para uma ou mais partições definidas previamente.

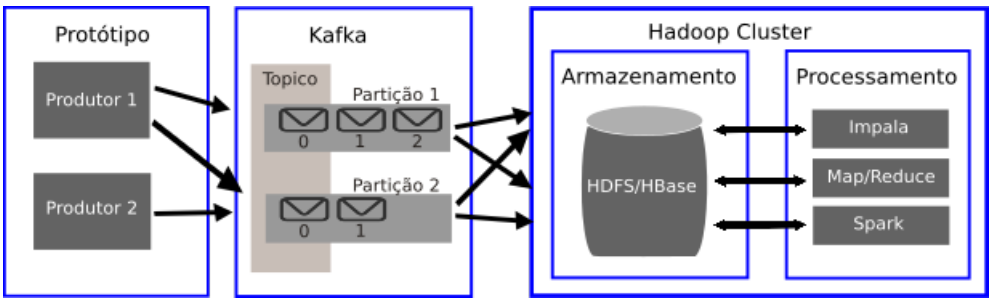

Figura 5. Esquema de interação entre o protótipo, Apache Kafka e Hadoop

4.1. Instalação e Configuração do Ambiente

Para realizar os experimentos para avaliação dos *brokers* foi necessária a instalação de algumas ferramentas e serviços que implementem o ambiente esperado, dentre eles pode-se citar o NTP *(Network Time Protocol)* e o protocolo de rede SSH *(Secure Shell)*. Na Figura 6 o ambiente de execução dos experimentos.

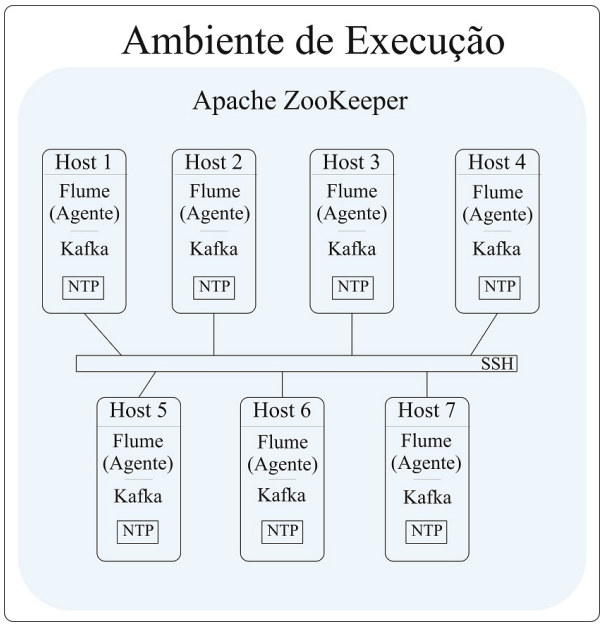

Figura 6. Ambiente onde foi executado os experimentos

O servidor NTP foi necessário para que todos os *hosts* de um determinado *cluster* se mantenham sempre com a data e hora em sincronia, buscando estas informações em um servidor remoto. Isso é crucial devido que, como um *cluster* possui diversas máquinas trabalhando em conjunto e muitas operação executadas entre elas são executada de acordo com um tempo pré-programado, um relógio atrasado pode trazer diversos problemas, por exemplo, impedir uma tarefa de execução de uma cópia de segurança em umas das máquinas.

Já o protocolo SSH teve como principal função realizar a comunicação entre os hospedeiros de um determinado *cluster*, este protocolo é utilizado principalmente pela segurança que disponibiliza, dando a possibilidade de criptografar a conexão entre cliente e servidor, seu funcionamento ocorre geralmente pela porta 22.

Após a instalação e configuração dos dois protocolos citados anteriormente, chega-se no ponto onde, de fato, os serviços principais que foram utilizados no trabalho serão instalados e configurados. Primeiramente o Apache Hadoop, serviço responsável por armazenar os dados gerados pelos *scripts* e transferidos pelos *brokers*. Com ele funcionando corretamente foi implantado o primeiro dos dois *brokers*, o Apache Flume, para que ele funcione, três parâmetros principais devem ser informados:

- *source*: responsável por coletar os dados gerados pelos *scripts* e aplicação e armazenar no *channel* configurado;
- *channel*: encarregado pelo tráfego dos dados coletados pela *source*, podendo ser um arquivo, uma base de dados, outro *broker* ou a memória principal da máquina;
- *sink*: o propósito da *source* é o de extrair os eventos armazenados no *channel* e armazena-los em uma base de dados, outro agente ou um sistema de arquivos.

Para a implantação do segundo *broker*, primeiramente foi instalado e configurado o Apache ZooKeeper. Este serviço é uma dependência do Kafka que tem como principal função prover uma sincronia de configurações entre os diversos *hosts* localizados no *cluster* promovendo uma arquitetura de alta disponibilidade por meio de serviços de redundância.

Por fim chegamos ao segundo *broker* que foi utilizado no trabalho, o *Apache Kafka*, assim como o *Flume*, seu objetivo principal é realizar o envio de mensagens de uma fonte para algum consumidor, para que ele funcione corretamente. Foi desenvolvido um produtor responsável por coletar os dados e enviar para o kafka e um consumidor responsável por coletar os dados transmitidos pelo *broker*.

4.2. Geração de Dados

Após a configuração do ambiente, foram gerados dados para a realização dos experimentos. Inicialmente foram utilizados dados simulados, ou seja, criados por meio de *scripts* em *shell script* que foram capazes de criar os seguintes conjuntos de dados: (i) muitos arquivos pequenos; (ii) poucos arquivos extremamente grandes e (iii) uma mescla de muitos arquivos pequenos e grandes.

A Figura 7 apresenta os *scripts* implementados para a geração dos dados. O primeiro *script* foi o responsável por gerar muitos arquivos pequenos, onde o número de arquivos criados é controlado pelo laço *for* e pode ser alterado de acordo com a necessidade, o tamanho dos arquivos são definidos por duas variáveis, a *block size* (BS), que é o tamanho de cada bloco que será copiado e em seguida temos o *count*, que é quantos blocos serão copiados. Para a geração de apenas um ou poucos arquivos extremamente grandes, foi utilizado o segundo *script*. A diferença entre ele e o primeiro está apenas no limite do laço *for* e na variável *block size*, sendo que ela será alterada para que se consiga um arquivo com o tamanho esperado. No exemplo exposto, será criado um arquivo com o tamanho de 1 GB. Para conseguir gerar arquivos de tamanhos mesclados foi utilizado o comando *$RANDOM* que ele é capaz de gerar números aleatórios entre 0 e 32767. Para

obter valores de um intervalo numérico específico será utilizado *RANDOM* $\%$ X, onde X será o limite máximo obtido por meio do resto da divisão pelo número aleatório.

```
1 - for ((i=1;i<=100000;i++));
        do dd if=/dev/urandom of=/home/arquivos/arquivo_$i.txt
        bs=$(($RANDOM % 100 + 1)) count=100;
    done


2 - for ((i=1;i<=1;i++));
        do dd if=/dev/urandom of=/home/matheus/arquivo_$i.txt
        bs=10000000 count=100;
    done


3 - for ((i=1;i<=100;i++));
        do dd if=/dev/urandom of=/home/matheus/arquivo_$i.txt
        bs=$(($RANDOM % 10000)) count=$(($RANDOM % 10000));
    done
```

Figura 7. Scripts de geração de arquivos.

A geração dos arquivos é feita a partir do arquivo *urandom*, pertencente da própria distribuição Linux, ele é capaz de fornecer caracteres de forma aleatória e ilimitada.

Além destes arquivos sintéticos gerados pelos *scripts*, foi realizado experimentos com fluxo de dados ou *streaming*, estes fluxos foram obtidos através da API do próprio Twitter que será descrita posteriormente.

4.3. Métricas

Durante a execução dos experimentos, algumas formas de mensurar seu desempenho foram utilizadas, observando alguns trabalhos relacionados pode-se observar diversas semelhanças neste sentido, como por exemplo, a vazão de dados durante a execução de uma determinada tarefa, isto é, a taxa de transferência de dados que cada *broker* é capaz de oferecer. Neste trabalho foram utilizadas como unidades de medida, *Megabyte* e *Kilobyte*.

Assim, foram definidas as seguintes métricas:

- **Vazão de dados**: verifica qual a quantidade de *bytes* por segundo cada *broker* consegue transportar, neste caso, quanto maior for seu resultado melhor será seu desempenho, independente do experimento executado.
- **Consumo de processamento**: verifica o quanto cada *broker* estressa a máquina em que ele está sendo executado. Sua medição é feita pela porcentagem do consumo do processador. Neste caso um desempenho melhor é definido por uma taxa de consumo mais baixa.
- **Taxa de erros durante a transmissão de dados**: verifica qual *broker* apresenta mais problemas durante a execução, esta medição é feita pelo número de mensagens que não conseguiram chegar ao final do processo de transferência, neste caso, quanto menor for o número de falhas melhor o desempenho.

Estas métricas são usadas para se ter o real desempenho de cada um dos *brokers* avaliados em diferentes condições de uso, configurações e cenários. Desta forma, pode-se realizar o estudo entre ambos *brokers* e foi possível verificar qual deles leva vantagem em relação ao outro em cada cenário executado.

4.4. Aplicação

Além da geração de dados sintéticos, foi desenvolvida uma aplicação capaz de coletar dados reais e entregar para os *brokers* que foram avaliados. Para o seu desenvolvimento foi utilizada a API (Interface de Programação de Aplicação) *Twitter streaming API*, que tem como principal funcionalidade a possibilidade de recuperar qualquer *tweet* público. Esta mesma API foi utilizada por [Córdova 2014] para realizar uma análise de desempenho entre dois serviços pertencentes a fase de processamento de dados.

A aplicação em questão teve como finalidade recuperar *tweets* públicos além da possibilidade de conseguir filtrar os *tweets* de acordo com parâmetros informados e ainda, agregar essas informações e formata-las em um padrão aceito pelos *brokers*. Além disso, para possibilitar o funcionamento foram geradas duas *Consumer Key*, sendo uma delas secreta e dois *Tokens* de acesso, sendo um deles secreto também. Na Figura 8 o esquema de funcionamento da aplicação desenvolvida.

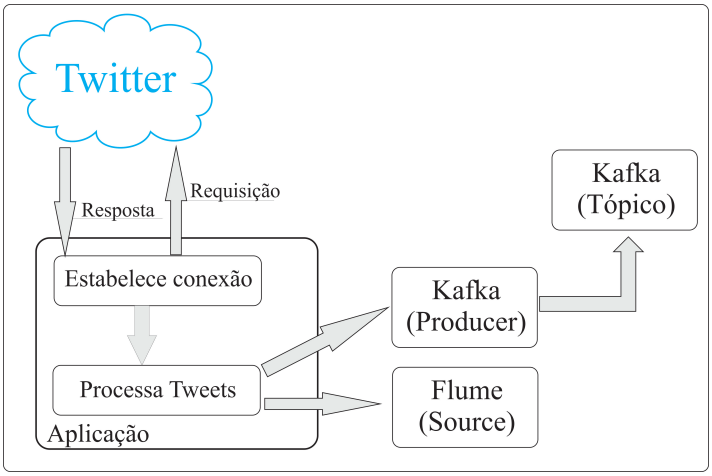

Figura 8. Esquema de funcionamento da aplicação.

A *Consumer Key* é utilizada para a autenticação do usuário cadastrado no *Twitter*, para conseguir acesso a API *Twitter Streaming*, já os *Tokens* tem a função de autenticar as requisições enviadas ao *Twiter* para a coleta das mensagens [Twitter]. Estes parâmetros foram informados no código fonte para permitir o funcionamento da aplicação.

Durante a execução dos experimentos, a aplicação coletou *Tweets* públicos sem a utilização de nenhum parâmetro de busca, como por exemplo, alguma *Hashtag* ou alguma *string* específica que compõe a mensagem.

Para o *Kafka* a aplicação foi integrada diretamente com o *driver* produtor do *broker* e do produtor para o tópico configurado. O tópico pode ser definido como uma categoria de mensagens, que neste caso possuirá apenas um configurado para os sete *brokers* do *cluster*. Já para o *Flume* basta a correta configuração dos três parâmetros principais, o *source*, *channel* e o *sink* para que seja possível seu funcionamento. Esta aplicação foi

responsável por gerar o *streaming* de dados, em tempo real, para a realização dos experimentos.

5. Experimentos

Nesta seção serão apresentados os experimentos realizados e quais os resultados obtidos através deles, tendo como objetivo principal o de mensurar o desempenho dos dois *brokers* que foram avaliados, em diferentes cenários. Primeiramente foram realizados experimentações com arquivos sintéticos, isto é, gerados de forma artificial por meio de *scripts* em *Shell Script*.

Estes arquivos tiveram quantidade e tamanho definidos de maneira aleatória, posteriormente experimentos utilizando a API Twitter *streaming*, que foi responsável por coletar dados reais e em tempo real, para cada um dos experimentos três execuções foram realizadas aproveitando o resultado médio. As particularidades de cada experimento serão detalhadas a seguir.

5.1. Setup dos Experimentos

Para a realização dos experimentos foram utilizadas as versões 0.8.2.2 do Apache Kafka, 3.4.8 do Apache Zookeeper e 1.6.0 do Apache Flume, configurados em 7 nós no *cluster*.

Durante a avalização dos *brokers*, foram executados quatro experimentos distintos. Primeiramente foi realizado utilizando uma grande quantidade de arquivos simulados, com tamanho médio de 11,8 *Kilobytes*, o objetivo deste é simular a transferência de arquivos de *logs* de servidores.

O segundo experimento foi executado utilizando, também arquivo simulados, porém com um tamanho médio de 1009 *Megabytes*. Aqui o propósito é simular arquivos extremamente grandes, como por exemplo, o transporte de arquivos multimídia.

O terceiro experimento foi executado utilizando uma mescla de arquivos pequenos com arquivos grandes, também simulados, o tamanho médio dos arquivos gerados foi de 4,6 *Megabytes*, variando de 10 *Kilobytes* até 10 *Megabytes*. Este experimento busca simular a transferência de arquivos comuns gerados por usuários, com tamanho similar a arquivos de fotos, músicas, mensagens de *email* e documentos de texto.

Por fim, o quarto experimento foi realizado por meio da aplicação desenvolvida, coletando *tweets* reais do Twitter em tempo real, durante 4 horas. Estes diversos formatos de experimentos, foram necessários para que fosse possível verificar em quais ambientes cada *broker* teria mais desempenho em relação ao outro. Na tabela abaixo os experimentos com maiores detalhes.

Tabela 1. Setup dos Experimentos

	Nº de Arquivos	Tempo de Execução	Nº de Brokers	Tam. Total
Experimento 1	2.650.578	N/A	7	31,32 GB
Experimento 2	28	N/A	7	28,26 GB
Experimento 3	8323	N/A	7	38,31 GB
Experimento 4	N/A	4 horas	3	N/A

Como mostra a tabela, o experimento 4 não possui um número fixo de arquivos, isso se dá pelo fato que a aplicação coleta fluxo de dados, por isso também, o tamanho

total transportado varia para cada um dos *brokers* após as quatro horas de execução. Outro ponto à ser observado está ligado aos três primeiros experimentos, estes utilizando arquivos sintéticos, não possui um tempo de execução fixo, pois varia para cada um dos *brokers*.

5.2. Ambiente Físico

Os experimentos foram realizados em um *cluster* composto por sete computadores *Dell Optiplex GX270*, cada um deles equipados com processador *Pentium 4 HT* 2.80 GHz, 3GB de memória RAM e 160GB de armazenamento, cada computador possui software Linux Rocks 6.1 *Emerald Boa*. A conexão entre os nós é feita através de interfaces de 1000Mbps. Este *cluster* é de propriedade do GPPD (grupo de Processamento Paralelo e Distribuído) da UFRGS.

A utilização de um *cluster* foi necessária para minimizar qualquer limitação física de processamento, e com isso garantir que os resultados obtidos durante os experimentos, tenha como único limitador o próprio *broker*.

5.3. Resultados

Após a realização dos experimentos, pode-se constatar algumas características específicas de cada *broker*, como em quais ambientes cada um leva vantagem sobre o outro. A figura 9, mostra a vazão de dados utilizando arquivos sintéticos, ou seja, gerados por meio de *scripts*.

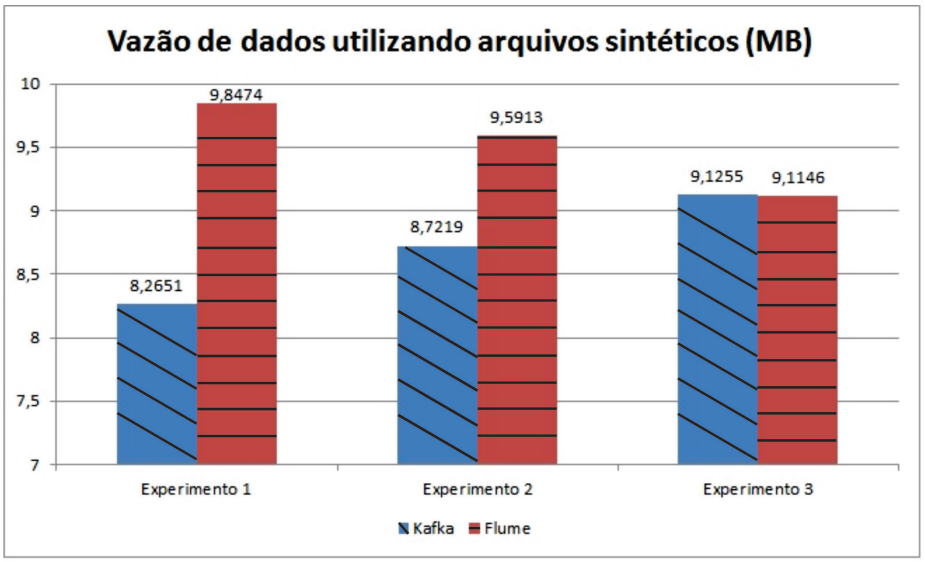

Figura 9. Vazão de dados utilizando arquivos gerados a partir de scripts

Observando os resultados, pode-se notar uma ampla vantagem do Apache Flume em dois dos experimentos e um resultado próximo ao seu concorrente em apenas uma das execuções, o primeiro experimento em que o Flume leva vantagem, consistem em uma quantidade muito grande de arquivos pequenos, mais especificamente 2650578 de arquivos, com um tamanho médio de 11,8 KB, neste cenário a vantagem do Flume em relação ao kafka foi de 19% ou 1,5823 MB/s.

O segundo experimento consiste em um número muito pequeno de arquivos, porém com um tamanho médio muito maior em relação ao experimento anterior, 1009MB. Assim como na primeira experiência o Apache Flume teve um desempenho superior ao seu concorrente tendo uma vazão de dados 10% ou 870 KB/s maior.

O terceiro e último experimento utilizando arquivos sintéticos, é composto por 8323 arquivos com tamanho variando de 10 *Kilobytes* até 10 *Megabytes*. Nesta execução, ambos os *brokers* tiveram desempenho praticamente igual, sendo o Kafka apenas 0,11% ou 10,9 KB/s.

O último experimento executado foi utilizando a aplicação desenvolvida, aqui o protótipo foi responsável por coletar postagens no *Twitter* em tempo real e entregar aos dois *brokers*, para esta execução, não foi configurado nenhum filtro de consulta, ou seja, foi coletado qualquer *tweet* público disponível. O experimento teve duração de quatro horas para cada um dos *brokers* e após o término pode-se constatar uma vazão de 21,6% ou 126,72 KB/s superior do Kafka em relação ao Flume, como mostra a figura 10.

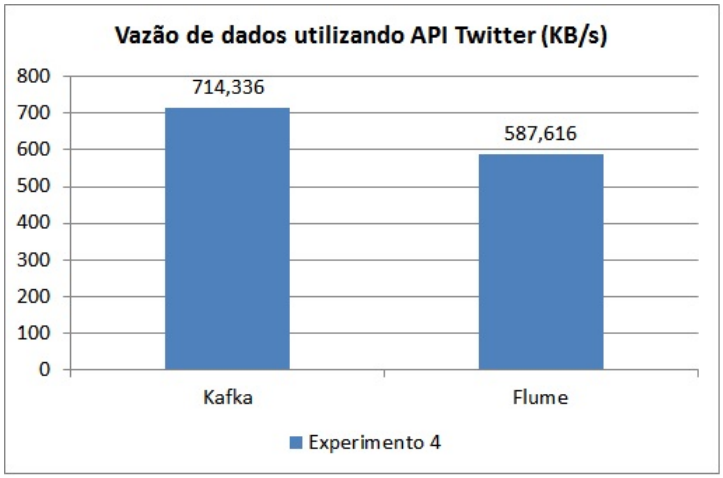

Figura 10. Vazão de dados utilizando a aplicação desenvolvida

No quesito consumo de processamento, os dois *brokers* obtiveram resultados semelhantes, nos experimentos utilizando a aplicação, a porcentagem de uso do processador, para cada um dos nós, se manteve próximo de 5%, já para os testes utilizando arquivos sintéticos, o Flume manteve uma média de 10% durante a execução, já o Kafka, demonstrou uma economia de 5% em relação ao seu concorrente. Durante a execução dos experimentos, não foi detectada nenhuma falha no decorrer do transporte dos dados.

Nas três primeiras experiências, para o Flume, foi utilizado como *source* o diretório onde estava armazenado os arquivos e como o canal de transmissão a memória, por ter um desempenho superior em relação aos outros tipos existentes. Para o quarto experimento a *source* era alimentada pela aplicação e como canal de transmissão, assim como nos experimentos anteriores, a memória foi utilizada. Para o Kafka, foi desenvolvido um *driver* produtor responsável por coletar os arquivos armazenados utilizados nos três primeiros experimentos e um outro produtor integrado com a aplicação desenvolvida, para ambos os casos, foi configurado um tópico para cada instância do *broker*.

Por fim, observando os resultados obtidos, é possível relaciona-los com algumas

características de cada *broker*, por exemplo, o Kafka tem como foco o processamento de *streaming* de dados em tempo real. a API *producer* contida nele, permite que aplicações desenvolvidas enviem fluxo de dados para os seus tópicos de maneira eficiente. Isso explica o seu melhor desempenho no quarto experimento, utilizando a aplicação desenvolvida.

Já nos experimentos anteriores, o Flume obteve um desempenho superior. O principal motivo para isso, se dá pelo fato de que sua arquitura mais simples tem como foco o processamento de arquivos de *logs* de servidores, através da coleta e agregação destes dados. Por isso este *broker* não depende de nenhum outro *software* para realizar esta coleta, bastando apenas a correta configuração do *source*.

6. Considerações Finais

A primeira etapa do trabalho buscou apresentar o referencial teórico, mostrar alguns trabalhos relacionados, a instalação e configuração de todo o ambiente necessário para a realização dos experimentos. Além disso, foram definidas algumas métricas para serem utilizadas durante a execução dos *benchmarks* e como os dados necessários para a execução dos *brokers* deverão ser gerados.

Já na segunda etapa, foi apresentado um ambiente simulado para a geração de dados de *log* e *streaming*, para posterior coleta através dos dois *brokers* avaliados. Ainda o desenvolvimento da aplicação proposta e a realização dos experimentos visando obter resultados para o comparativo de desempenho entre eles.

No que diz respeito aos *brokers*, pode-se notar alguns pontos principais em relação ao desempenho, levando em conta principalmente a vazão de dados. De uma maneira geral, quando se trata de transferência de conjuntos de arquivos o Apache Flume leva vantagem em quase todos os cenários sendo alcançado pelo seu concorrente em apenas um deles. Entretanto, quando se trata de *streaming* de dados em tempo real, a situação se inverte, sendo o Kafka 21,6% mais rápido que seu rival.

Estes resultados vão de encontro com o foco de aplicação de cada *broker*, onde o Flume tem sua arquitetura desenvolvida visando um melhor desempenho na coleta e agregação de *logs* de servidores e o Kafka com o seu desenvolvimento voltado para o transporte de *streaming* de dados em tempo real.

Ao término do trabalho, acredita-se que os principais objetivos elencados foram alcançados, mesmo com algumas dificuldades encontradas duranta a fase de implementação, entre estas dificuldades pode-se citar a limitação quanto ao número de conexões máxima permitida pela API Twitter Streaming, permitindo a execução em apenas três das sete máquinas do *cluster*.

Como trabalho futuro, uma possibilidade seria a de realizar uma análise de desempenho em serviços responsáveis por fazer a análise dos dados coletados.

Referências

Apache. Apache Flume: developer guide. `https://flume.apache.org/FlumeDeveloperGuide.html/`. Acessado: 03/12/2016.

Apache. Apache Flume: user guide. `https://flume.apache.org/FlumeUserGuide.html/`. Acessado: 28/05/2016.

Apache. Apache Kafka: documentação. http://kafka.apache.org/. Acessado: 28/05/2016.

Apache. Apache ZooKeeper: documentação. https://zookeeper.apache.org/. Acessado: 28/05/2016.

Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., et al. (2010). A view of cloud computing. *Communications of the ACM*, 53(4):50–58.

Boschi, S. and Santomaggio, G. (2013). *RabbitMQ Cookbook*. Packt Publishing Ltd.

Chen, M., Mao, S., and Liu, Y. (2014). Big data: A survey. *Mobile Networks and Applications*, 19(2):171–209.

Cloudera (2015). Cloudera distribution of apache kafka. http://www.cloudera.com/documentation/kafka/latest/topics/kafka.html/. Acessado: 28/05/2016.

Córdova, P. (2014). Analysis of real time stream processing systems considering latency.

Coulouris, G., Dollimore, J., Kindberg, T., and Blair, G. (2013). *Sistemas Distribuídos-: Conceitos e Projeto*. Bookman Editora.

dos Anjos, J., Assunção, M. D., Bez, J., Geyer, C., de Freitas, E. P., Carissimi, A., Costa, J. P. C., Fedak, G., Freitag, F., Markl, V., et al. (2015). Smart: An application framework for real time big data analysis on heterogeneous cloud environments. In *Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing (CIT/IUCC/DASC/PICOM), 2015 IEEE International Conference on*, pages 199–206. IEEE.

Gantz, J. and Reinsel, D. (2011). Extracting value from chaos. *IDC iview*, 1142:1–12.

Gantz, J. and Reinsel, D. (2012). The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east. *IDC iView: IDC Analyze the future*, 2007:1–16.

Garg, N. (2013). *Apache Kafka*. Packt Publishing Ltd.

Henjes, R., Menth, M., and Zepfel, C. (2006). Throughput performance of java messaging services using webspheremq. In *26th IEEE International Conference on Distributed Computing Systems Workshops (ICDCSW'06)*, pages 26–26. IEEE.

Hoffman, S. (2013). *Apache Flume: Distributed Log Collection for Hadoop*. Packt Publishing Ltd.

Hortonworks. Apache ZooKeeper: how zookeeper works? http://hortonworks.com/apache/zookeeper/#section_2. Acessado: 28/05/2016.

Hu, H., Wen, Y., Chua, T.-S., and Li, X. (2014). Toward scalable systems for big data analytics: a technology tutorial. *Access, IEEE*, 2:652–687.

IBM. Apache ZooKeeper: how does it work? http://www-01.ibm.com/software/data/infosphere/hadoop/zookeeper/. Acessado: 28/05/2016.

Ionescu, V. M. (2015). The analysis of the performance of rabbitmq and activemq. In *RoEduNet International Conference-Networking in Education and Research (RoEduNet NER), 2015 14th*, pages 132–137. IEEE.

Jones, B., Luxenberg, S., McGrath, D., Trampert, P., and Weldon, J. (2011). Rabbitmq performance and scalability analysis, project on cs 4284: Systems and networking capstone. *Virginia Tech*.

Kumar, A. (2015). Getting familiarized with the hadoop distribution file system. `http://www.developer.com/db/getting-familiarized-with-the-hadoop-distribution-file-system.html/`. Acessado: 28/05/2016.

Mell, P. and Grance, T. (2011). The nist definition of cloud computing.

Shvachko, K., Kuang, H., Radia, S., and Chansler, R. (2010). The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10. IEEE.

Snyder, B., Bosnanac, D., and Davies, R. (2011). *ActiveMQ in action*, volume 47. Manning.

Taylor, M. E. et al. (2012). *WebSphere MQ Primer: An Introduction to Messaging and WebSphere MQ*. IBM Redbooks.

Thein, K. M. M. (2014). Apache kafka: Next generation distributed messaging system. *International Journal of Scientific Engineering and Technology Research*, 3(47):9478–9483.

Twitter. Twitter Developer: documentation. `https://dev.twitter.com/docs/`. Acessado: 03/12/2016.

White, T. (2012). *Hadoop: The definitive guide*. "O'Reilly Media, Inc.".

Zhang, Q., Cheng, L., and Boutaba, R. (2010). Cloud computing: state-of-the-art and research challenges. *Journal of internet services and applications*, 1(1):7–18.