Monitoramento de Tráfego e Detecção de Ameaças por Sistemas Distribuídos de Processamento de Fluxos: uma Análise de Desempenho *

Martin Andreoni Lopez,
Antonio Gonzalez Pastana Lobato,
Otto Carlos Muniz Bandeira Duarte

[1]Grupo de Teleinformática e Automação – GTA
Universidade Federal do Rio de Janeiro – UFRJ
Rio de Janeiro – RJ – Brasil

{martin, antonio, otto}@gta.ufrj.br

Abstract. *Real-time stream processing extracts knowledge from large continuous streams of data in applications such as Internet of things and traffic monitoring. This article proposes a structural analysis and a performance evaluation of native stream processing systems Storm and Flink and the micro-batch processing Spark Streaming. These systems process continuous streams of Big Data with low latency and high throughput. Conventional tools are not suitable to treat streams, due to it processes batches of data in different time. The article analyzes the architecture and compares the features of the main open source systems for the processing streams. The experimental comparison of throughput efficiency and resilience to node failures of the three systems is held in an anomaly detection application. Results show that performance of native stream processing systems is up to 15 times higher than micro-batch processing systems, but only micro-batch systems can recover from failures without loss.*

Resumo. *O processamento de fluxos em tempo real extrai conhecimento de grandes fluxos contínuos de dados, como os de aplicações de Internet das coisas e monitoramento de tráfego. Este artigo propõe uma análise estrutural e avaliação de desempenho dos sistemas de processamento de fluxos nativos Storm e Flink e de processamento em micro-lotes Spark Streaming. Esses sistemas processam fluxos contínuos de grandes massas de dados com baixa latência e alta vazão. Ferramentas convencionais não são apropriadas para tratar fluxos, pois processam lotes de dados em tempo diferenciado. O artigo analisa as arquiteturas e compara as características dos principais sistemas de código aberto para o processamento de fluxo. A comparação experimental da eficiência de vazão e da resiliência a falhas de nós dos três sistemas é realizada em uma aplicação de detecção de ameaças. Os resultados obtidos mostram que a vazão de análise de mensagens nos sistemas de processamento nativo de fluxos é até 15 vezes mais alta que em sistemas de processamento por micro-lotes, porém somente o sistema por micro-lotes consegue se recuperar de falhas sem perdas.*

1. Introdução

Monitoramento baseado em sensores, monitoramento de rede, processamento de tráfego Web, controle e gerência de dispositivos móveis, detecção de ameaças de segurança [Andreoni Lopez et al. 2016] fazem parte de aplicações que geram uma grande

*Este trabalho foi realizado com recursos do CNPq, CAPES, FAPERJ e FINEP.

quantidade de dados para serem processados em tempo real. Estas aplicações são caracterizadas por um fluxo (*stream*) composto por uma sequência não limitada de eventos ou tuplas [Stonebraker et al. 2005]. Estas aplicações de monitoramento em tempo real são complexas de serem realizadas e com o advento da Internet das coisas a área será ainda mais solicitada, pois a estimativa para o número de sensores que estarão conectados em rede em 2025 é de 80 bilhões de sensores [Clay 2015]. Dados com esta ordem de grandeza não podem ser processados de forma centralizada.

Nos últimos anos, as plataformas de processamento distribuídas têm sido empregadas para análise de grandes massas de dados (*big data*). Estas plataformas distribuídas se servem majoritariamente da técnica de mapeamento e redução (*MapReduce*) e, em particular, da implementação *Hadoop*. As aplicações processadas em lotes (*batch*) pelo *Hadoop* correspondem a perguntas (*queries*) ou transações (*transactions*) realizadas em uma base de dados armazenada e passiva, na qual somente o estado atual é importante. Além disso, assume-se que as aplicações não possuem requisitos de tempo real, os elementos de dados estão sincronizados e que as perguntas possuem uma resposta exata.

Apesar sucesso do *Hadoop* como modelo de programação para processamento em lotes de grandes massas de dados, ele é inapropriado para processamento de fluxo em tempo real. As aplicações de monitoramento em tempo real requerem o processamento distribuído de fluxo e, portanto, diferem em diversos aspectos das aplicações convencionais processadas pelas plataformas distribuídas atuais. O monitoramento usualmente requer a análise de diversas fontes de fluxos externas, por exemplo de sensores, e tem a função de gerar alertas de condições anormais. Além disso, a característica de tempo real é intrínseca às aplicações de processamento de fluxo e uma grande quantidade de disparos de alertas é prevista. Os fluxos de dados são ilimitados, a análise dos fluxos requer dados históricos em vez de apenas os mais atuais reportados e os dados chegam de forma assíncrona. Nos casos de alta taxas de entrada é comum procurar filtrar os dados mais importantes descartando os demais e, portanto, soluções aproximadas se fazem necessárias. Assim, para atender estas aplicações que demandam processamento distribuído em larga escala de fluxo em tempo real, modelos de processamento distribuído têm sido propostos e esta nova área tem recebido uma enorme atenção dos pesquisadores.

A aplicação de monitoramento de tráfego para detecção de ameaças de segurança cibernética em tempo real é beneficiada por estes novos modelos de processamento de fluxos. Os sistemas de detecção e prevenção de intrusão atuais não são eficazes, já que 85% das ameaças levam semanas para serem detectadas e até 123 horas para que uma reação seja executada após a detecção [Ponemon e IBM 2015]. Empregar novos Sistemas de Processamento de Fluxos Distribuídos (*Distributed Stream Processing Systems* - DSPS) em tempo real para aplicações de segurança é fundamental e no futuro com o avanço da Internet das coisas isto será imperativo.

Algumas plataformas de uso geral e de código aberto estão disponíveis e procuram atender às necessidades de processar dados de forma contínua e ser capaz de definir aplicativos de processamento de fluxo personalizados para os casos específicos. Estas plataformas de uso geral fornecem uma interface de programação de aplicação (*Application Programing Interfaces* - API), escalabilidade e tolerância a falhas.

Neste trabalho são descritos e analisados os sistemas de processamento distribuído de fluxo em tempo real nativos, o Apache Storm [Toshniwal et al. 2014] e o Apache Flink [Carbone et al. 2015], e em micro-lotes, o Apache Spark Streaming [Zaharia et al. 2013]. A arquitetura de cada um dos sistemas analisados é discutida com profundidade e também é apresentada uma comparação conceitual que mostra as diferenças entre essas plataformas de

código aberto. Além disso, experimentos sobre uma aplicação em tempo real de detecção de ameaças desenvolvida pelos autores [Lobato et al. 2016] foram realizados. Esses experimentos avaliam a vazão do processamento de dados e o comportamento dos sistemas quando ocorre uma falha de nó. Os resultados são analisados e comparados com o modelo conceitual de cada plataforma avaliada.

O restante do artigo está organizado da seguinte forma. A Seção 2 discute os trabalhos relacionados. O processamento de fluxo é apresentado na Seção 3. Os sistemas analisados são apresentados na Seção 4. Na Seção 5 são discutidos os resultados obtidos. Por fim, a Seção 6 conclui o artigo.

2. Trabalhos Relacionados

A área de sistemas distribuídos de processamento em tempo real de fluxo é recente e as avaliações de desempenho e comparações entre sistemas são ainda pouco explorada na literatura científica. Existem sistemas de processamento de fluxo mais antigos e descontinuados como o Aurora [Carney et al. 2002], o Projeto da Stanford STREAM [Arasu et al. 2004] e o projeto desenvolvido pela Yahoo o Apache S4 [Neumeyer et al. 2010] que serviram de base para a maioria dos sistemas atuais. Entre as soluções comerciais atuais é possível encontrar o Google Millwheel [Akidau et al. 2013],

Hesse e Lorenz comparam as plataformas Apache Storm, Flink, Spark Streaming e Samza [Hesse e Lorenz 2015]. A comparação se restringe a descrição da arquitetura e dos seus principais elementos e, portanto, não são feitas avaliação de desempenho práticas. Gradvohl e outros analisam e comparam os sistemas MillWheel, S4, Spark Sreaming e Storm, focando no aspecto de tolerância a falhas em sistemas de processamento on-line [Gradvohl et al. 2014]. Porém, o artigo se restringe a discussões conceituais sem nenhuma experimentação. Landset e outros realizam um resumo das ferramentas utilizadas em grandes massas de dados (*Big Data*) [Landset et al. 2015], no qual são apresentadas as arquiteturas dos sistemas de processamento de fluxo. No entanto, o maior foco deste artigo é nas ferramentas de processamento em lotes que se servem das técnicas de mapeamento e redução e não em processamento de fluxo.

Roberto Coluccio e outros mostram a viabilidade prática e o bom desempenho do uso de sistemas distribuídos de processamento de fluxo para o monitoramento de fluxo de sinalização do sistema de número 7 (*Signalling System number 7* - SS7) em aplicações de comunicação celular GSM (*Global System for Mobile Communications*) máquina-a-máquina (*Machine-to-Machine*) - M2M) [Coluccio et al. 2014]. Os desempenhos de dois sistemas de processamento de fluxo são analisados e comparados: do Storm e do Quasit, um protótipo da Universidade de Bolonha. O principal resultado do trabalho é comprovar a viabilidade prática real do Storm conseguir processar com folga e em tempo real uma grande quantidade de dados de uma aplicação de telefonia móvel.

Nabi *et. al* comparam a plataforma Apache Storm com o sistema IBM InfoSphere Streams em uma aplicação de processamento de mensagens de correio eletrônico [Nabi et al. 2014]. Os resultados mostram um melhor desempenho da InfoSphere comparado ao Apache Storm no que se refere vazão e ocupação de CPU. No entanto, o sistema InfoSphere é proprietário da IBM e neste trabalho o foco são as ferramentas de código aberto. Lu e outros propõem um *benchmark* [Lu et al. 2014] criando um primeiro passo na comparação experimental de plataformas de processamento de fluxo. Eles realizam uma comparação da latência e da vazão dos sistemas Apache Spark Streaming e o Apache Storm. Porém, o artigo não fornece resultados do comportamento dos sistemas em relação à tolerância a falhas dos sistemas.

Dayarathna e Suzumura [Dayarathna e Suzumura 2013] comparam o desempenho da

vazão de *job*, consumo de CPU e memória, e utilização de rede para os sistemas de processamento de fluxo System S, S4 e Esper. Um aspecto importante foi o critério usado para escolha dos sistemas que diferem em seus modelos arquiteturais, pois o System S segue o modelo gerente/trabalhadores, o S4 possui uma arquitetura descentralizada e simétrica que segue o modelo de atores e, finalmente, o Esper é completamente diferente porque é apenas um componente do processamento de fluxo. Um importante resultado foi o pior desempenho do sistema S4 que tem que instanciar elementos processadores na sua arquitetura paralela. Embora a análise através de *benchmarks* seja interessante, praticamente todos os sistemas analisados estão descontinuados ou não possuem atualmente popularidade significativa.

Diferentemente dos artigos citados anteriormente, as seções que se seguem descrevem as diferenças arquiteturais dos sistemas de código aberto Apache Storm, Apache Flink e o Apache Spark Streaming. Além disso, é feita uma análise de desempenho foca na vazão e no paralelismo de uma aplicação de detecção de intrusão em um conjunto de dados construído pelos autores. Também é avaliado a reação e a tolerância dos sistemas quando um dos nós falha. Finalmente é realizado um resumo crítico das principais características de cada um dos sistemas analisados e também comentado como as características influenciam nos resultados obtidos.

3. O Processamento de Fluxos

O processamento de fluxo de dados é modelado através de um grafo. O grafo contém fontes de dados que continuamente emitem as amostras que são processados por nós conectados que realizam alguma computação real sobre os itens. Um fluxo de dados ψ é um conjunto infinito de dados, $\psi = \{D_t | 0 \leq t\}$ onde um ponto D_t é um conjunto de atributos com uma estampa de tempo explícito ou implícito. Formalmente um ponto de dados é $D_t = (\mathbf{V}, \tau_t)$, onde $\mathbf{V}$ é um p-tupla, na qual cada valor corresponde a um atributo, e τ_t é a estampa de tempo para o t-ésimo dado. Os nós fontes emitem tuplas ou mensagens que são então recebidos por nós que realizam processamento, chamados de elementos de processamento (EP). Cada EP recebe dados em suas filas de entrada, executa alguma computação na entrada usando seu estado local e produz uma saída para suas filas de saída. O conjunto de nós fontes e de processamento criam uma rede lógica de elementos de processamento de fluxo conectados em um gráfico acíclico dirigido (GAD). O GAD é uma representação gráfica de um conjunto de tarefas e dos nós que as executam como mostrado na Figura 1.

Stonebraker *et.al.* [Stonebraker et al. 2005] ressaltam os requerimentos mais importantes para as plataformas distribuídas de processamento de fluxo. A mobilidade dos dados é o termo usado para identificar como os dados se movimentam através dos elementos de processamento. A mobilidade de dados é a chave fundamental para a manter baixa latência, já que as operações de bloqueio, como feita no processamento em lotes em plataformas como o *Hadoop*, diminuem a mobilidade de dados. Portanto, devido ao grande volume de dados, é necessário particionar os dados para que eles sejam tratados em paralelo.

Alta disponibilidade e recuperação de falhas são fundamentais nos sistemas de processamento de fluxos. Devido ao requerimento de baixa latência, a recuperação deve ser rápida e eficiente, fornecendo garantias de processamento. Logo, as plataformas de processamento de fluxos devem fornecer mecanismos para proporcionar resistência contra imperfeições da transmissão, incluindo atrasos, dados perdidos ou fora da sequência, que são comuns em fluxos de dados e garantias de processamento, as quais são associadas com os algoritmos de alta disponibilidade. Algoritmos como *backup* ativo, *backup* passivo e apoio ao montante são utilizados pelos DSPEs para fornecer garantias de processamento de dados entre componentes com falha.

Além disso, as plataformas devem possuir a capacidade de armazenar dados de forma

eficiente, acessar e modificar as informações de estado, e combiná-las com dados dos fluxos em tempo real. A capacidade de processar dados armazenados e combiná-los com dados de processamento em fluxo permite ajustar e verificar os algoritmos para obter melhor desempenho. Para integração contínua, o sistema deve usar uma linguagem uniforme ao lidar com qualquer tipo de dados. Linguagens de alto nível e ferramentas para consulta são necessários para desenvolver aplicativos no topo das plataformas de processamento de fluxo.

Os sistemas de processamento devem ter um mecanismo de execução altamente otimizado, para oferecer resposta em tempo real para aplicações com alto volume de dados. Isso significa ter a capacidade para processar desde dezenas até centenas de milhares de mensagens por segundo com baixa latência, da ordem de microssegundo. Para se obter esse bom desempenho de processamento, as plataformas devem minimizar a sobrecarga (*overhead*) de comunicação entre processos distribuídos na transmissão dos dados no grafo de processamento.

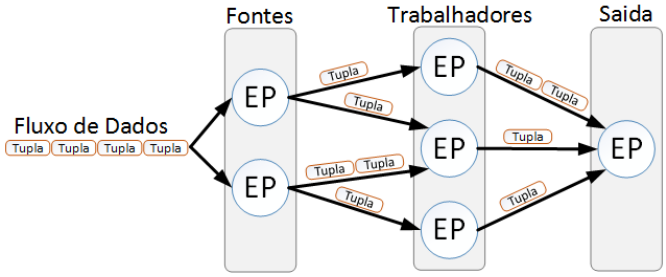


Figura 1: Rede lógica de Elementos Processadores (EP) interligados criando um gráfico acíclico dirigido. O fluxo de dados é recebido pelos EP fontes, logo são processados pelos trabalhadores e finalmente são agregados na saída.

3.1. Tipos de Processamento

Existem três principais abordagens para processar dados: o processamento em Lotes, o processamento de micro-lotes e o processamento de fluxos. A análise de grandes conjuntos de dados estáticos, que são recolhidos ao longo de um período é comumente realizada por processamento em lotes (*batch*). Neste esquema, os dados são recolhidos, armazenados nas fontes de dados e depois processados. No entanto, esta técnica apresenta grandes latências, com respostas superiores a 30 segundos, enquanto algumas aplicações requerem processamento em tempo real com respostas da ordem de sub-segundo [Rychly et al. 2014]. Uma evolução desta técnica para realizar processamentos quase em tempo real é o processamento de micro-lotes (*micro-batch*). A ideia é tratar o fluxo como uma sequência de pequenos blocos de dados em lotes. Em intervalos de tempo pequenos, o fluxo de entrada é juntado em blocos de dados e entregue ao sistema de lote para ser processado. Por sua vez, a técnica de processamento de fluxo de dados (*streaming processing*) analisa uma sequência massiva de dados ilimitados que são continuamente gerados.

Uma vantagem de processamento de fluxo é a sua expressividade, já que o fluxo não é limitado por qualquer abstração não natural. Além disso, como as mensagens são processadas imediatamente após da chegada, a latências destes sistemas são melhores que a dos sistemas em micro-lotes. Os sistemas de processamento de fluxos têm geralmente melhor desempenho em tempo real, mas a tolerância a falhas é mais custosa, já que deve ser realizada para cada uma das mensagens processadas.

Dividir o fluxo em micro-lotes inevitavelmente reduz a expressividade do sistema. Algumas operações, especialmente de gestão de estado, união e divisão, são mais difíceis de implementar, já que o sistema manipula um lote inteiro. Ao contrário do processamento de fluxo,

a tolerância a falhas e balanceamento de carga são muito mais simples, porque o sistema envia todos os lotes a um nó trabalhador e se algo der errado, basta usar um outro nó diferente.

3.2. Tolerância a Falhas

A alta disponibilidade é essencial para a maioria das aplicações de processamento de fluxo em tempo real. O sistema deve ser capaz de se recuperar de uma falha de maneira suficientemente rápida para que o processamento normal continue sem afetar o sistema global. Por isso, existe uma grande preocupação por a garantia no processamento dos dados [Kamburugamuve et al. 2013]. Em sistemas de grande escala de computação distribuída diversos motivos podem gerar falhas, seja por falhas de nós, da rede, erro de *Software* ou limitações dos recursos. Os sistemas tradicionais de processamento em lotes, como o Apache Hadoop, onde altas latências são aceitas, não necessitam de recuperações de falhas de maneira rápida. Já nos sistemas em tempo real, uma falha representa perda de dados, já que os dados não estão armazenados no sistema. Portanto, para esses sistemas, a rápida recuperação é importante para evitar a perda de informações.

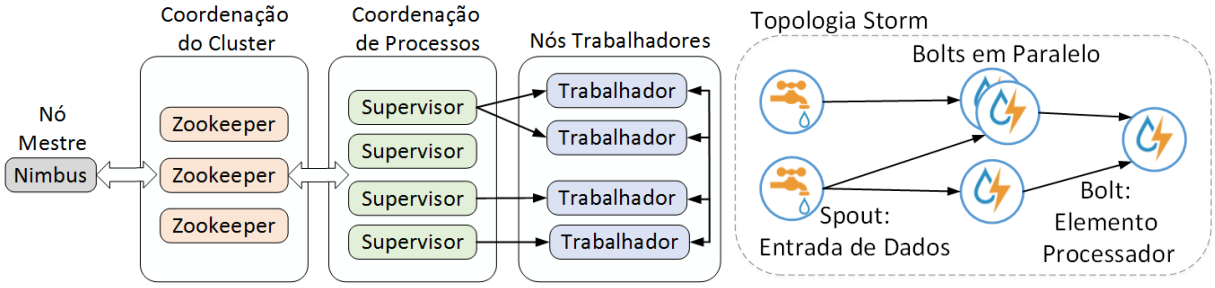
Existem três distintas semânticas de entrega de mensagens, exatamente uma vez, pelo menos uma vez e até uma vez (*Exactly-once*, *At-least-once* e *At-most-once*). As semânticas dizem respeito à garantia que o sistema dá sobre o processamento ou não de dados. Quando ocorre uma falha na transmissão e no processamento de dados, pode ocorrer o reenvio dos dados para que nenhuma informação seja perdida. A semântica mais simples é a até uma vez, na qual não há recuperação de erros e ou os dados são processados uma vez, ou são perdidos. Na semântica pelo menos uma vez, a correção de erro é feita de maneira conjunta para um grupo de amostras, dessa maneira, caso ocorra erro com alguma dessas amostras, o grupo inteiro é repetido. Essa semântica é menos custosa do que a exatamente uma vez, que requer um reconhecimento para cada amostra processada.

4. Sistemas Analisados

4.1. O Sistema Apache Storm

O Storm [Toshniwal et al. 2014] é um processador de fluxo de dados (*Data Streaming Processor*- DSP) em tempo real. A abstração dos fluxos é chamada Tuplas, com um identificador e o dado propriamente dito. Este processador é composto por topologias que formam Grafos Acíclicos Dirigidos (GAD) compostos por elementos de entradas, os *spouts*, e elementos de processamento, os *bolts*, como mostrado na Figura 2b. Cada um desses elementos pode ter várias instâncias em paralelo. Uma topologia funciona como um grafo de dados, que são processados em forma de fluxo à medida que os dados avançam. A ligação entre dois nós do grafo de processamento é definida através do tipo de agrupamento de dados utilizado.

No Storm existem oito tipos de agrupamento de fluxos de dados que representam como os dados são passados para o próximo nó do grafo de processamento, a topologia, e para as suas instâncias em paralelo, que realizam a mesma lógica de processamento. Os principais tipos de agrupamento são: o aleatório, o por campo e o total. No agrupamento aleatório, as amostras do fluxo de dados são passadas para apenas uma instância do próximo *bolt* da topologia, sendo feita de maneira aleatória a decisão de qual instância processará a amostra. No agrupamento por campo, cada *bolt* ficará responsável por todas as amostras que apresentarem a mesma chave da tupla. Desta forma, um *bolt* ficará responsável por todas as amostras de um certo tipo e conseguirá concentrar as informações referentes a esse tipo. Por fim, no agrupamento total, todas as amostras são enviadas para todas as instâncias em paralelo do mesmo *bolt*, ou seja, as



(a) Coordenação de processos no Storm. **(b) Topologia no Storm.**

Figura 2: a) O *Zookeeper* monitora o estado dos nós trabalhadores através dos supervisores e informa esse estado para o nó mestre tomar decisões de alocação de processos da topologia. b) Topologia com elementos de entrada, *spouts* e de processamento *bolts*, que podem ter diversas instâncias em paralelo.

amostras são replicadas e processadas por todos os elementos de processamento em paralelo do próximo nó do grafo de processamento.

A Figura 2a mostra como é feita a coordenação de processos no Storm. Um dos nós do *cluster* distribuído do Storm é o mestre, controlador, que executa o Nimbus. O Nimbus recebe as informações da topologia, definidas pela aplicação programada pelo usuário, e coordena a instanciação de cada processo necessário para atender à especificação da topologia, ou seja, coordena a instanciação dos *spouts* e *bolts* e de seus processos paralelos. Esses processos executam nos nós trabalhadores. Cada nó trabalhador executa um supervisor, que monitora os processos da topologia e informa o estado para o Zookeeper. O Zookeeper possui então uma visão geral do estado da topologia e repassa essa informação para o Nimbus que pode decidir realocar processos, como no caso de falha por exemplo.

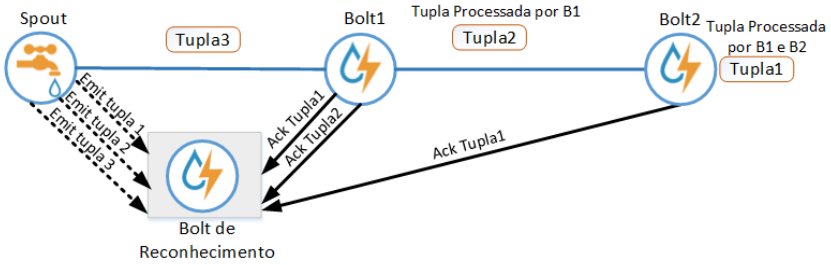


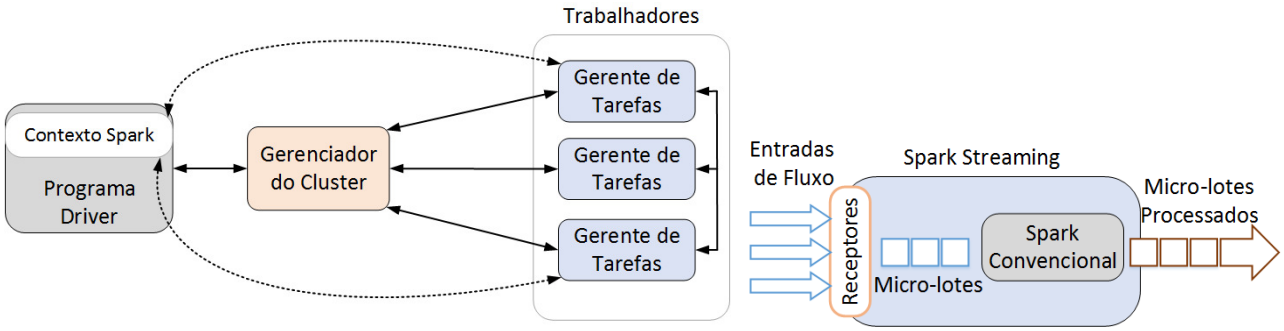
Figura 3: Semântica de entrega de mensagens pelo menos uma vez (*At-least-once*) utilizada no Apache Storm. Cada vez que uma tupla é emitida pelo Spout, um registro é emitido ao Bolt de reconhecimento. Logo quando a tupla é processada por um Bolt um Ack é emitido reconhecendo o processamento.

O apache Storm usa um mecanismo de *backup* de fluxo acima e reconhecimentos de registros para garantir que as mensagens sejam re-processadas após uma falha. Para cada registo que é processado um reconhecimento (*ACK*) é enviado para o operador anterior. A fonte da topologia mantém um backup de todos os registros que gera. Uma vez reconhecidos todos os registros até o nó destino, a fonte pode descartar os backups. Se uma falha acontecer, significa que nem todos os reconhecimentos foram recebidos ou que reconhecimento não veio dentro de um período pré-definido. Em seguida, os registos são reenviados pela fonte. O processo é ilustrado na Figura 3. Assim, é garantido que os dados não são perdidos, no entanto registos duplicados podem circular através do sistema. Essa garantia, conhecida como entrega de pelo menos uma vez (*At-least-once*) apresenta um bom desempenho, mas não garante a consistência

do estado, qualquer manipulação estado é delegada ao usuário. Neste modo, as mensagens podem ser perdidas se a tarefa falhar ou *timeout* dos reconhecimentos for excedido. Este modo não requer nenhum tratamento especial e as mensagens são processadas na ordem produzida.

4.2. O Sistema Apache Spark Streaming

O Spark é um projeto iniciado pela universidade de Berkeley e é uma plataforma para processamento de dados distribuída. O Spark suporta diferentes bibliotecas, entre elas que se encontra o Spark Streaming [Zaharia et al. 2013] para o processamento de fluxos de dados. A abstração do fluxo é chamado de Conjunto de Dados Distribuídos Resilientes (*Resilient Distributed Dataset*), que são definidos como um conjunto de objetos particionados através dos nós do *cluster*. A Figura 4b mostra o processamento de fluxos no Spark. O fluxo de dados é a entrada para o Spark Streaming, que cria os micro-lotes em forma de RDDs. Esses lotes são passados para o Spark Convencional, que faz o processamento. Um trabalho (*Job*) no Spark é definido como uma computação paralela que consiste em múltiplas tarefas, e uma tarefa é uma unidade de trabalho que é enviada ao executor. O Spark foi desenvolvido em Scala e Java e contem APIs para programar com essas linguagens e também Python.



(a) Arquitetura de Cluster no Spark Streaming **(b) Modelo de micro-lotes.**

Figura 4: a) b) Os fluxos são recebidos pelos receptores que os convertem em micro-lotes, logos são enviados ao Spark Convencional para serem processados.

A disposição do *cluster* é mostrada na Figura 4a. As aplicações dentro do Spark são executadas como processos independentes no *cluster* que são coordenadas pelo do Programa Driver, responsável de executar a função `main()` do aplicativo e criar o `Contexto Spark`. O Contexto Spark se conecta a vários tipos de gerenciadores de *cluster*, como o próprio do Spark, Mesos ou o Hadoop YARN (*Yet Another Resource Negotiator*). Os gerenciadores são responsáveis por alocar recursos entre aplicativos. Uma vez conectado, o Spark adquire executores em nós do *cluster*, dentro do gerente de tarefas, que são processos que executam processamento e armazenamento de dados, equivalentes aos trabalhadores do Storm. Em seguida, o código do aplicativo é enviado para os executores, e finalmente, o Contexto Spark envia tarefas para os executores. Um mecanismo semelhante ao mecanismo descrito no Storm, onde cada processo trabalhador é executado dentro de uma topologia, pode ser aplicado no Spark, onde as aplicações são equivalentes às topologias. Uma desvantagem deste conceito no Spark é a troca de mensagens entre os diferentes programas do Contexto Spark, que unicamente é feito de maneira indireta como por exemplo escrever dados em um arquivo.

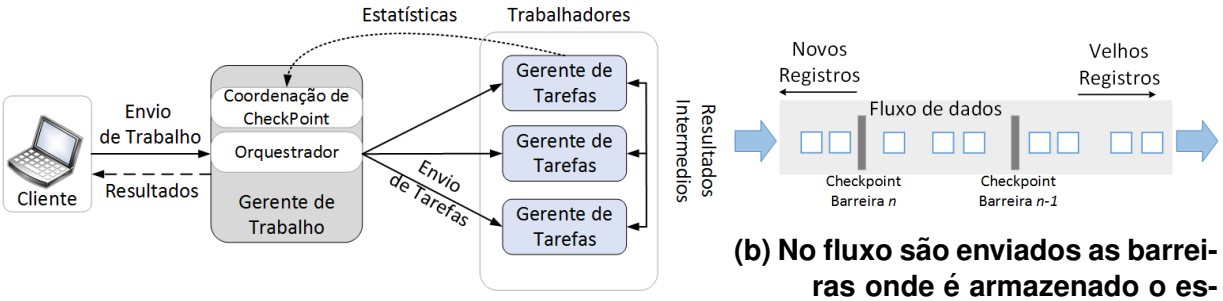
O Spark Streaming possui semântica de entrega exatamente uma vez (*Exactly-once*). A ideia é a mesma seguida no processamento em lotes, onde uma tarefa, *Job* ou micro-lotes, são processados em vários nós do trabalhador. Em uma falha, o processamento do micro-lote pode ser simplesmente recalculado, uma vez que são persistentes e imutáveis. Em caso de falhas

de nós no Spark o cálculo é discretizado em pequenas tarefas que podem ser executados em qualquer nó sem afetar a execução. Logo, as tarefas que falharam pode ser relançado em nós paralelo no *cluster*, distribuindo uniformemente a tarefa sem que seja afetado o desempenho. Esse procedimento é chamado de Recuperação Paralela.

A semântica de exatamente uma vez reduz o *overhead* apresentado na confirmação continua de todos os registros, no entanto, apresenta algumas desvantagens. O processamento de um micro-lote leva mais tempo em operações fluxo abaixo (*downstream*). A configuração de cada um dos micro-lotes pode demorar mais tempo que o analise de fluxo convencional. Como consequência, muitos micro-lotes são armazenados na fila de processamento. A latência também é um problema no processamento de micro-lotes, enquanto sub-segundo de latência é viável para aplicações simples, aplicações com múltiplas operações levam a latência até a ordem dos segundos.

4.3. O Sistema Apache Flink

O Apache Flink [Carbone et al. 2015] é uma plataforma de processamento híbrido, suportando tanto processamento de fluxos assim como processamento em lotes. Ao contrário do Spark Streaming o núcleo do Flink é o processamento de fluxos, fazendo o processamento em lotes uma classe especial de aplicação. A garantia de mensagens fornecida pelo apache Flink é exatamente uma vez. Essa plataforma é escrita em Java e Scala e apresenta API para trabalhar com essas linguagens. O esquema conceitual do Flink é mostrado na Figura 5a. Semelhante ao Apache Storm, o Flink utiliza o modelo de mestre-trabalhador. O gerente de trabalho é a interface com as aplicações dos clientes, tendo responsabilidades semelhantes ao nó mestre do Storm. O gerente de trabalho recebe tarefas dos clientes, as organiza para os nós trabalhadores, assim como mantem o estado de todas as execuções e o estado de cada trabalhador. O estado dos trabalhadores é informado através do mecanismo *Heartbeat*. Os gerentes de tarefas executam as tarefas atribuídas pelo gerente de trabalho e troca informações entre os outros trabalhadores quando necessário. Cada diferenciador de tarefas fornece certo número de espaços de processamentos ao *cluster* que são utilizados para realizar as tarefas de maneira paralela.



(a) O gerente de trabalho recebe os trabalhos desde o cliente, logo divide o trabalho em tarefas e envia para os trabalhadores que comunicam as estatísticas e os resultados.

(b) No fluxo são enviados as barreiras onde é armazenado o estado do fluxo, o operador envia o estado ao coordenador dos pontos de verificação no Gerente de Trabalho.

Figura 5: a) Arquitetura do Apache Flink e b) Modelo de Tolerância a Falhas baseados em *checkpoints* do Flink.

A abstração dos fluxos é chamada DataStream que são sequências de registros parcialmente ordenados. Os DataStreams podem ser criados a partir de fontes externas, como filas de mensagens, fluxos de *sockets*, entre outros ou invocando operações em outros fluxos de dados. Os Datastreams suportam vários operadores, tais como a função de mapeamento, filtragem e redução em forma de funções que são aplicadas de forma incremental para registro

gerando novos fluxos de dados ou DataStreams. Cada operador pode ser paralelizado, colocando instâncias paralelas para executar em diferentes partições do respectivo fluxo, assim, é permitindo a execução distribuída de transformações sobre os fluxos.

A abordagem de tolerância a falhas adotada pelo Flink é baseada em *snapshot checkpoints* distribuídos que mantém o estado dos trabalhos. Os *snapshots* atuam como pontos de verificação consistente aos quais o sistema pode voltar em caso de uma falha como mostrado na Figura 5b. As barreiras são primeiramente injetadas nas fontes e fluem através do grafo como parte do fluxo de dados, em conjunto com os registros de dados. Uma barreira, que indica o início de um ponto verificação, separa registos em dois grupos: aqueles que são parte da imagem atual e as que fazem parte da imagem seguinte. As barreiras disparam instantâneos (*Snapshots*) do estado quando passam através de operadores. Quando um operador recebe a barreira, armazena o estado do fluxo correspondente e envia ao coordenador dos pontos de verificação dentro do Gerente de Trabalho do Flink. Em caso de uma falha de programa por um nó, rede ou falha de software, o Flink para o envio de dados por fluxos. O sistema, em seguida, reinicia os operadores e os redefine para o último *checkpoint* bem sucedido. Os fluxos de entrada são redefinidos ao ponto instantâneo armazenado do estado. Todos os registros que são processados no fluxo reiniciado são garantidos para não ter sido parte do estado verificado anterior, garantindo a entrega de exatamente uma vez.

A Tabela 1 apresenta um resumo das características ressaltadas durante a comparação das ferramentas de processamento de fluxo. Os modelos de programação utilizados nos sistemas analisados podem ser classificados como composicional e declarativo. A abordagem composicional fornece blocos de construção básicos, como *spouts* e *bolts* no Storm, e devem ser conectados juntos a fim de criar uma topologia. Já os operadores no modelo declarativo são definidos como funções de ordem superior, o que permite escrever código funcional com tipos abstratos e, a partir disso, o sistema criará automaticamente o grafo de processamento. A definição da função a ser realizada no fluxo de dados no modelo declarativo é passada como parâmetro na declaração da variável que abstrai as mensagens do fluxo.

Tabela 1: Resumo da comparação dos sistemas de processamento fluxo.

	Storm	**Flink**	**Spark Streaming**
Abstração de Mensagens	Tupla	DataStream	Resilient Distributed Dataset (RDD)
Linguagem Primaria de Implementação	Java/Closure	Java/Scala	Java/Scala
Organizador de Tarefas	Nimbus/YARN	YARN/Mesos	YARN/Mesos
Semântica de Processamento de Mensagens	Pelo menos uma vez	Exatamente uma vez	Exatamente uma vez
Mecanismo de Garantia de Mensagens	Upstream Backup	Check-point	Replica, Recuperação Paralela
API	Composicional	Declarativa	Declarativa
Subsistema para gerencia falhas	Nimbus,Zookeeper	Não	Não
Gerenciamento de Estados	Sem Estados	Operações com estado	Armazenamento de Estados

5. Resultados Experimentais

No processamento de fluxos em tempo real é importante obter requerimentos mínimos de latência, assim como também é imprescindível que os sistemas críticos sejam tolerantes a falhas. Esta seção avalia a vazão dos três sistemas apresentados: o Apache Storm versão 0.9.4, o Apache Flink versão 0.10.2 e o Apache Spark Streaming versão 1.6.1. Os experimentos utilizaram um ambiente com oito máquinas virtuais. As máquinas virtuais executam sobre uma máquina com processador Intel Xeon E5-2650 a 2.00 GHz e com 64 GB de memória RAM. A topologia utilizada é a de um nó mestre e sete trabalhadores para os três sistemas avaliados. Os resultados são obtidos com um intervalo de confiança do 95%. Para inserir dados a altas taxas nos sistemas de processamento de fluxos foi utilizado um mediador (*Broker*) de mensagens que opera como um serviço de Produtor/Consumidor, o Apache Kafka na versão 0.8.2.1. No Kafka, as tuplas ou eventos são chamados de mensagens, nome que será usado daqui em diante. O Kafka abstrai o fluxo de mensagens em tópicos que agem como diferentes *buffers* ou filas, adequando taxas distintas de produção e consumo. Logo, os Produtores gravam os dados em tópicos e os consumidores leem os dados a partir desses tópicos.

O conjunto de dados utilizado nas experimentações é o de uma aplicação de detecção de anomalia que foi criado pelos autores [Lobato et al. 2016]. O conjunto de dado é replicado de forma a avaliar a máxima vazão na qual os sistemas conseguem processar. Contudo, foi programada em Java uma aplicação detecção de ameaças mediante uma rede neural.

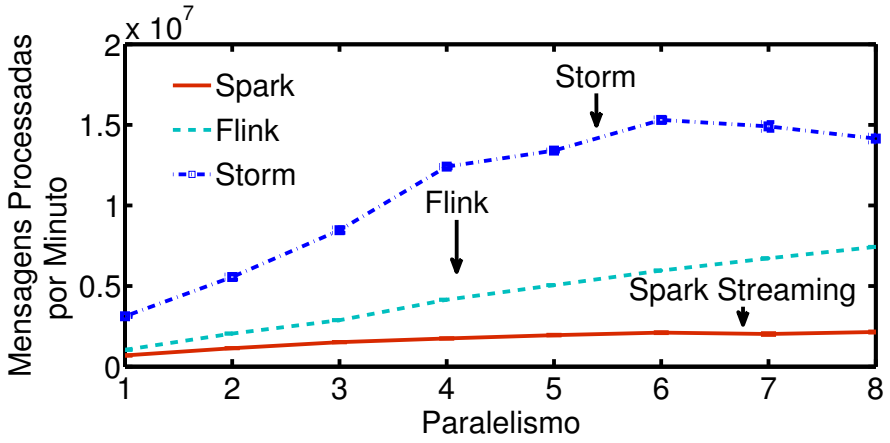
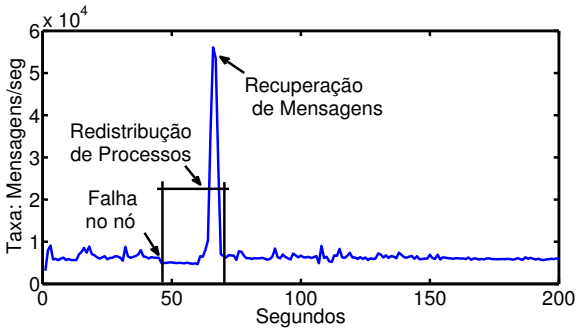


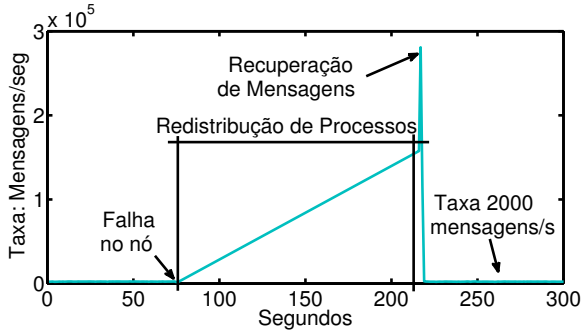
Figura 6: Resultados de vazão das plataformas em número de mensagens recebidas por minuto em função do paralelismo, em número de núcleos de processamento.

O primeiro experimento avalia o desempenho das plataformas no que se refere vazão. Para isso, o conjunto de dados é injetado no sistema na sua totalidade e replicado quantas vezes sejam necessárias. A taxa de consumo e processamento que cada uma das plataformas é obtida. Foi utilizado a variação do parâmetro paralelismo, que representa o número total de núcleos oferecidos ao aglomerado (*cluster*) para processar amostras em paralelo. A Figura 6 mostra os resultados da experimentação, no qual é possível verificar que o Apache Storm apresenta uma melhor vazão. Para apenas um único núcleo, ou seja, sem paralelismo, o Storm já apresenta um desempenho melhor em relação ao Flink e o Spark Streaming de pelo menos 50%. O Flink apresenta um crescimento totalmente linear, mas com valores sempre abaixo do Apache Storm. O desempenho do Apache Spark Streaming em comparação com O Storm e o Flink é muito menor, isso é devido a utilização dos micro-lotes, já que cada lote é armazenado para ser posteriormente processado. O comportamento do Apache Storm é totalmente linear até o paralelismo de quatro núcleos. Depois tem um crescimento menor até o paralelismo de seis,

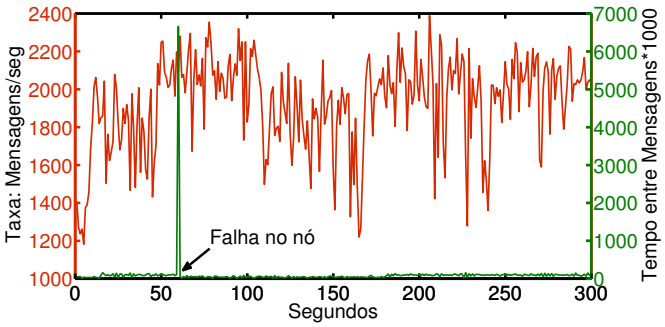
onde pode ser percebido que o sistema satura. Este comportamento foi observado também no Apache Spark Streaming com o mesmo paralelismo de seis núcleos.



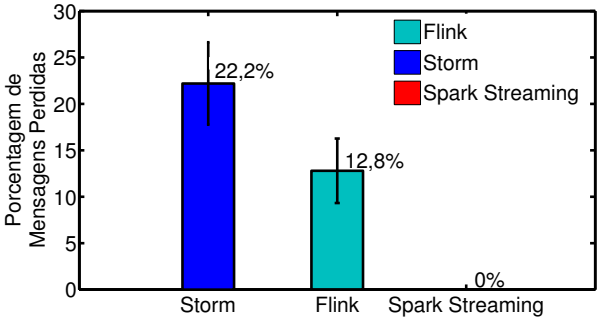
(a) Comportamento da ferramenta Storm ante uma falha de nó.



(b) Comportamento da ferramenta Flink ante uma falha de um nó.



(c) Comportamento da ferramenta Spark Streaming ante uma falha de um nó.



(d) Perda de mensagens durante uma falha de um nó entre os Sistemas.

Figura 7: Comportamento dos Sistemas durante uma falha de um nó produzida aos 50 segundos. Os sistemas Storm (a) e Flink (b) depois da detecção da falha redistribuem os processos e recuperam as mensagens. O sistema Spark Streaming (c) apresenta um comportamento semelhante antes e depois da falha, não perdendo Mensagens. A Figura (d) mostra a comparação da perda de mensagens durante a falha de um nó entre o Apache Storm e o Apache Flink.

O segundo experimento mostra o comportamento dos sistemas ante uma falha de um nó. Foram enviadas mensagens em uma taxa constante para analisar o comportamento dos sistemas durante a falha. A falha de nó foi simulada desligando uma máquina virtual. As Figuras 7a, 7b e 7c mostram o comportamento dos três sistemas antes e depois de uma falha de um nó trabalhador aos 50 segundos. O Apache Storm, após da detecção da falha, demora um tempo na redistribuição dos processos. Esse tempo se deve a comunicação com o *zookeeper* . O *zookeeper*, que possui uma visão geral do aglomerado (*cluster*), informa o estado para o Nimbus dentro do Storm, que realoca os processos em outros nós. Logo após essa redistribuição, o sistema recupera as mensagens do Kafka aproximadamente aos 75 segundos. No entanto, embora o sistema consiga se recuperar rapidamente da falha do nó, durante o processo existe uma perda significativas de mensagens. Um comportamento semelhante é observado no Apache Flink. Ao detectar a falha aproximadamente aos 50 segundos o sistema redistribui os processos para os nós que estão ativos. Esse processo é feito internamente pelo Flink sem a ajuda de nenhum subsistema, diferentemente do Apache Storm que se serve do do *zookeeper*.

Na Figura 7b é possível observar que o período de tempo no qual o Flink redistribui os processos, é muito maior que o tempo consumido no apache Storm. Por isso que a recuperação das mensagens é também maior, perdendo algumas mensagens durante o tempo de

redistribuição de processos. O comportamento do Apache Spark Streaming durante a falha é mostrado na Figura 7c. Aqui é possível observar que quando a falha ocorre, aproximadamente aos 50 segundos, o comportamento do sistema após a detecção é basicamente o mesmo do de antes da falha. Isso é devido a que as tarefas quando existe uma falha em um nó trabalhador é discretizado em pequenas tarefas que são rapidamente em nós trabalhadores paralelamente, distribuindo a tarefa sem afetar o desempenho. É importante destacar que este comportamento não apresentou perda de mensagens durante a falha. Assim, apesar do desempenho inferior do Apache Spark Streaming, ele é uma boa escolha em aplicações nas quais a resiliência e o processamento de todas as mensagens são necessários.

A Figura 7d mostra a comparação da perda de mensagens entre o Apache Storm e o Apache Flink, mostrando que o Apache Spark não apresentou perda durante uma falha. A medida mostra a percentagem de mensagens perdidas pelos sistemas, como as mensagens emitidas pelo Apache Kafka em relação as mensagens analisadas pelos Sistemas. Assim, o Apache Flink apresenta uma menor perda de mensagens durante uma falha com aproximadamente um 12,8% em relação ao 22,2% no Storm. O resultado é obtido com um intervalo de confiança do 95%.

6. Conclusão

Este artigo descreve e compara os três principais sistemas distribuídos de processamento de fluxos de código aberto: o Apache Storm, o Apache Flink e o Apache Spark Streaming. Deve ser ressaltado que o Storm e o Flink são sistemas originalmente projetados para processarem fluxos enquanto o Spark Streaming é uma extensão do Spark, que processa em lotes, para processar em micro-lotes e atender às aplicações de processamento de fluxo. Os sistemas se mostraram semelhantes em algumas características, tais como o fato de que todos os sistemas são executados dentro de uma máquina virtual Java e possuem o modelo mestre-trabalhador. Uma análise de desempenho da

Também foi realizado um outro experimento para mostrar a tolerância dos sistemas à falha de um nó. Nesse caso foi mostrado que o Spark Streaming, pelo seu modelo de processamento de micro-lotes, consegue se recuperar da falha sem perder nenhuma mensagem, uma vez que ele armazena o estado do processamento integral dos micro-lotes e distribui o processamento interrompido de maneira homogênea entre as outras máquinas trabalhadoras. Já os sistemas nativos de processamento de fluxo como o Storm e o Flink perdem mensagens apesar de utilizarem mecanismos mais complexos de recuperação de falhas. O Apache Flink mediante um algoritmo de *checkpoint* apresenta uma menor taxa de mensagens perdidas, aproximadamente um 12,8%, durante o a redistribuição de processos após de uma falha. O Storm perde 10% a mais, aproximadamente 22,2% de mensagens já que utiliza um subsistema, o *zookeeper*, para a sincronização dos nós. Portanto, a escolha da plataforma a ser usada deve levar em conta a aplicação, usando uma relação de compromisso entre a latência em função das mensagens analisadas, e da facilidade de recuperação de falhas.

Referências

Akidau, T., Balikov, A., Bekiroğlu, K., Chernyak, S., Haberman, J., Lax, R., McVeety, S., Mills, D., Nordstrom, P. e Whittle, S. (2013). Millwheel: Fault-tolerant stream processing at internet scale. *Proc. VLDB Endow.*, 6(11):1033–1044.

Andreoni Lopez, M., Mattos, D. M. F. e Duarte, O. C. M. B. (2016). An elastic intrusion detection system for software networks. *Annals of Telecommunications*, páginas 1–11.

Arasu, A., Babcock, B., Babu, S., Cieslewicz, J., Datar, M., Ito, K., Motwani, R., Srivastava, U. e Widom, J. (2004). Stream: The stanford data stream management system. Technical Report 2004-20, Stanford InfoLab.

Carbone, P., Fóra, G., Ewen, S., Haridi, S. e Tzoumas, K. (2015). Lightweight asynchronous snapshots for distributed dataflows. *Computing Research Repository (CoRR)*, abs/1506.08603.

Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Seidman, G., Stonebraker, M., Tatbul, N. e Zdonik, S. (2002). Monitoring streams: A new class of data management applications. Em *Proceedings of the 28th International Conference on Very Large Data Bases*, VLDB '02, páginas 215–226. VLDB Endowment.

Clay, P. (2015). A modern threat response framework. *Network Security*, 2015(4):5–10.

Coluccio, R., Ghidini, G., Reale, A., Levine, D., Bellavista, P., Emmons, S. P. e Smith, J. O. (2014). Online stream processing of machine-to-machine communications traffic: A platform comparison. Em *IEEE Symposium on Computers and Communication (ISCC)*, páginas 1–7.

Dayarathna, M. e Suzumura, T. (2013). A performance analysis of system s, s4, and esper via two level benchmarking. Em *Quantitative Evaluation of Systems*, páginas 225–240. Springer.

Gradvohl, A. L. S., Senger, H., Arantes, L. e Sens, P. (2014). Comparing distributed online stream processing systems considering fault tolerance issues. *Journal of Emerging Technologies in Web Intelligence*, 6(2):174–179.

Hesse, G. e Lorenz, M. (2015). Conceptual survey on data stream processing systems. Em *IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS)*, páginas 797–802.

Kamburugamuve, S., Fox, G., Leake, D. e Qiu, J. (2013). Survey of distributed stream processing for large stream sources. `http://grids.ucs.indiana.edu/ptliupages/publications/survey_stream_processing.pdf`. Acessado: 10/04/2016.

Landset, S., Khoshgoftaar, T. M., Richter, A. N. e Hasanin, T. (2015). A survey of open source tools for machine learning with big data in the hadoop ecosystem. *Journal of Big Data*, 2(1):1–36.

Lobato, A., Lopez, M. A. e Duarte, O. C. M. B. (2016). Um sistema acurado de detecção de ameaças em tempo real por processamento de fluxos. Em *XXXIV SBRC a ser publicado*, Salvador, Bahia.

Lu, R., Wu, G., Xie, B. e Hu, J. (2014). Stream bench: Towards benchmarking modern distributed stream computing frameworks. Em *IEEE/ACM 7th International Conference on Utility and Cloud Computing (UCC)*, páginas 69–78.

Nabi, Z., Bouillet, E., Bainbridge, A. e Thomas, C. (2014). Of streams and storms. *IBM White Paper*.

Neumeyer, L., Robbins, B., Nair, A. e Kesari, A. (2010). S4: Distributed stream computing platform. Em *IEEE International Conference on Data Mining Workshops (ICDMW)*, páginas 170–177. IEEE.

Ponemon, I. e IBM (2015). 2015 cost of data breach study: Global analysis. `www.ibm.com/security/data-breach/`. Acessado: 10/04/2016.

Rychly, M., Koda, P. e Smrz, P. (2014). Scheduling decisions in stream processing on heterogeneous clusters. Em *Eighth International Conference on Complex, Intelligent and Software Intensive Systems (CISIS)*, páginas 614–619.

Stonebraker, M., Çetintemel, U. e Zdonik, S. (2005). The 8 requirements of real-time stream processing. *ACM SIGMOD Record*, 34(4):42–47.

Toshniwal, A., Taneja, S., Shukla, A., Ramasamy, K., Patel, J. M., Kulkarni, S., Jackson, J., Gade, K., Fu, M., Donham, J., Bhagat, N., Mittal, S. e Ryaboy, D. (2014). Storm@twitter. Em *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, páginas 147–156. ACM.

Zaharia, M., Das, T., Li, H., Hunter, T., Shenker, S. e Stoica, I. (2013). Discretized streams: Fault-tolerant streaming computation at scale. Em *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, páginas 423–438. ACM.