# SPDSBench: Análise de Infraestruturas para Processamento em *Stream*⋆

Rúben Garcia[1][0000−0002−1850−7702] and José Simão[1,2][0000−0002−6564−593X]

[1] Instituto Superior de Engenharia de Lisboa
[2] INESC-ID Lisboa

**Resumo** O processamento em *batch* foi no passado dominante para a análise de grandes quantidades de dados. No entanto, esta abordagem pode não ser adequada quando se pretende que os dados sejam analisados o mais rapidamente possível, aplicando diferentes operações de transformação, sem necessidade de armazenamento dos dados para posterior deteção de eventos críticos, minimizando assim a latência de todo o processo. O processamento em *stream* é um paradigma que cumpre estes requisitos e que tem vindo a ser utilizado tanto em cenários de *cloud* como *edge* para análise de diferentes tipos de sensores, dados de mobilidade em cidades, entre outros. Comparar os sistemas de processamento de dados em *stream* (SPDS) disponíveis é um desafio multi-factor, com dependências desde o *hardware* ao tipo de operações envolvidas nas topologias, passando pelos diferentes componentes distribuídos de cada sistema. Este artigo apresenta trabalho em curso para desenvolver o um ambiente distribuído para teste de SPDS, SPDSBench, o qual possibilita a recolha de várias métricas relevantes, como o tempo de processamento de evento e recursos consumidos e uma topologia com três fontes de dados e usando múltiplos operadores. A ferramenta desenvolvida usa técnicas de *Infrastructure-as-Code* (IaC) para automatizar o aprovisionamento em diferentes ambientes distribuídos e fornecedores de *cloud*.

**Keywords:** Processamento em *stream* · Arquiteturas distribuídas de monitorização · Análise de desempenho

## 1 Introdução

Um sistema de processamento de dados em *stream* apresenta dois conceitos fundamentais: *stream* e topologia. Uma *stream* representa um fluxo continuo de dados sem limite, ou seja de tamanho desconhecido ou ilimitado [4]. Uma topologia é um grafo orientado sem ciclos de operações que processam e reencaminham os dados provenientes de uma *stream*, sejam recebidos ou transmitidos entre sistemas externos, ou entre operações. Existem atualmente vários sistemas de processamento de dados em *stream*, tais como, *Apache Storm* [1], *Apache Flink* [2], *Apache Kafka* [14], *Apache Heron* [18], *Apache Samza* [19] e *Hazelcast-Jet* [13]. Dado o conjunto alargado de SPDS existentes, escolher um destes sistemas pode

⋆ Comunicação

tornar-se difícil dada a vasta popularidade que estes têm vindo a adquirir ao longo do tempo. Contudo, a popularidade não é um factor que faça decidir qual SPDS usar numa situação que envolva a análise de grandes quantidades de dados em tempo real, uma vez que o mais importante destes sistemas são as suas características e o desempenho.

Os desafios que surgem na escolha do SPDS estão principalmente relacionados com o modelo de programação e com o desempenho dos diferentes componentes envolvidos [23]. É necessário compreender as operações que estes sistemas disponibilizam de forma a tornar mais ou menos complexo expressar as transformações a executar sobre os dados. Cada sistema apresenta vários níveis de abstracção no seu modelo de programação o que pode fazer com que, uma mesma topologia pode ser descrita com o encadeamento de operações como *filter*, *map*, *reduce* e *join*, como também o uso da linguagem SQL para descrever a mesma topologia. Por outro lado, perceber que tipo de infraestrutura é necessária para alojar os vários componentes envolventes de cada SPDS , e quais os componentes estritamente necessários para o seu correcto funcionamento. Para além da infraestrutura, a facilidade da instalação destes componentes é um factor importante quando se pretende escalar horizontalmente estes sistemas.

Alguns dos trabalhos realizados no âmbito da avaliação e comparação entre os vários SPDS têm como foco principal o estudo do desempenho destes sistemas, nomeadamente a latência e a taxa de transferência de eventos, apresentando a natureza dos dados para análise e descrevendo qual a topologia a ser executada [3,24,25]. Apresentam também a infraestrutura e configurações usadas durante a execução da topologia sendo possível medir valores de desempenho dos sistemas. Karimov et al. [17] propõe uma nova abordagem a ser aplicada para análise de desempenho de topologias permitindo assim uma análise mais precisa e correcta, principalmente em operadores que necessitam de manter estado durante períodos de tempo até serem aplicadas operações atribuídas, por exemplo, agregações ou junções entre *streams* com recurso a *windowing*. Estes trabalhos apesar de descreverem as topologias e configurações usadas nos momentos de execução para o estudo de desempenho, não apresentam uma arquitectura com o foco em estender o seu uso com outros componentes e simplificação do processo de instalação destes sistemas e execução da topologia garantido a replicabilidade e extensibilidade.

Eeste artigo apresenta trabalho em curso para desenvolver o sistema SPDS-Bench com os seguintes objetivos:

1. Análise de desempenho de alguns SPDS, tais como *Apache Storm*, *Apache Flink* e *Apache Kafka*, relativamente a aspectos quantitativos e qualitativos que possam ajudar na escolha do sistema mais indicado para o processamento de dados em *stream*;
2. Disponibilização de um *benchmark open-source* com o intuito de estender o seu uso a outros SPDS e diferentes topologias;
3. Simplificação dos processos de instalação destes sistemas, bem como gestão de infraestrutura de modo a garantir a sua replicabilidade e extensibilidade;

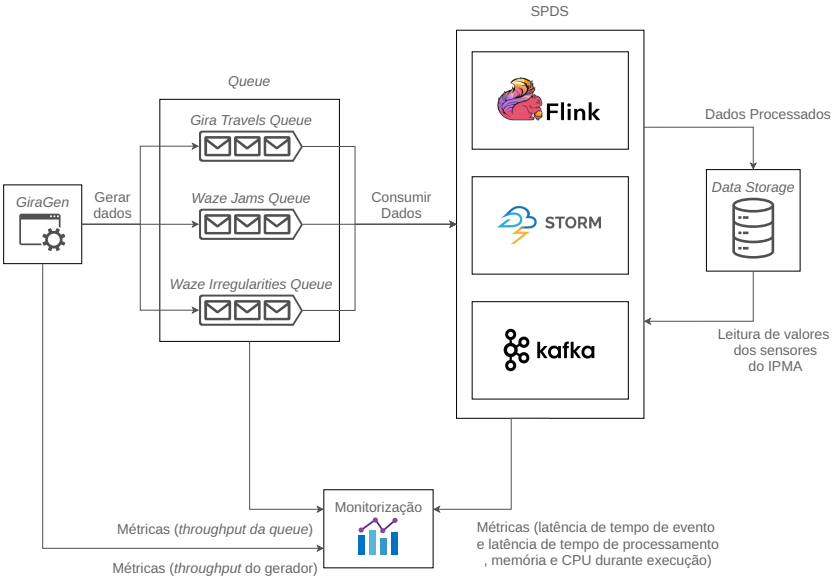



Figura 1: Arquitetura do SPDSBench

A topologia de base na versão atual do benchmark usa dados do desafio "Existem padrões de utilização das bicicletas partilhadas em Lisboa?"no âmbito do projeto de dados abertos do LxDataLab [5]. Estes dados são compostos por viagens realizadas nas bicicletas Gira, juntamente com informação sobre o trânsito e irregularidades do Waze [26] e os valores médios dos sensores meteorológicos do IPMA. Para além da topologia, foram também desenvolvidos processos de instalação destes sistemas e gestão de infraestrutura em *clouds* públicas, nomeadamente na *Amazon Web Services* (AWS). Por fim, foram desenvolvidos painéis de visualização em tempo real de desempenho dos SPDS.

## 2 Arquitetura do SPDSBench

A arquitectura do SPDSBench é apresentada na Figura 1, sendo composta pelos seguintes componentes:

- **Gerador de dados** (*GiraGen*), que é responsável por gerar novos dados com uma frequência variável, tendo como base os dados do *Gira Travels*, *Waze Jams* e *Waze Irregularities*;
- **Queue**, nomeadamente RabbitMQ [21], que será responsável por receber os dados provenientes do gerador de dados e servirá como ponto de recolha dos dados pelos SPDS ;
- **SPDS**, nomeadamente o *Apache Storm*, *Apache Flink* e *Apache Kafka*;
- **Data Storage**, nomeadamente *Redis* [22], que será responsável por manter os dados do IPMA e receber os dados processados pelos SPDS;

– **Monitorização**, composta pelo *Telegraf* [16], *InfluxDB* [15] e *Grafana* [12] que será responsável por recolher métricas dos SPDS , a nível das latência do tempo de evento e do tempo de processamento, utilização da memória, CPU e tráfego. Este também irá recolher métricas relativas ao *throughput* nas filas de mensagens, e também do *GiraGen*.

O *GiraGen* é uma aplicação de geração de dados tendo por base os dados abertos disponibilizados pelo *LxDataLab* [5]. Este permite configurar o *throughput* do gerador, apesar de que o limite real depende da capacidade computacional da infraestrutura de execução. Na *Queue*, optou-se por utilizar uma fila de mensagens que permitisse que estas ficassem temporariamente em memória até serem consumidas para que este componente não se torne num ponto de estrangulamento da arquitectura. Apesar desta arquitectura seguir um desenho mais próximo de um sistema real, este componente pode ser substituído por outro componente que sirva o mesmo propósito.

No *Data Storage*, optou-se por utilizar um armazenamento em memória pela mesma razão mencionada anteriormente. Apesar deste componente servir o propósito tanto de escrita dos dados processados como de leitura de valores, está divido em (i) escrita dos dados processados com base em fila de mensagens ou *message broker*, ou um sistema de armazenamento de dados, relacional ou não relacional; (ii) leitura de valores com base num sistema de armazenamento de dados, relacional ou não relacional.

### 2.1 Componentes e Operadores dos SPDS

Os SPDS abordados neste trabalho são compostos por um conjunto de elementos de mais baixo nível, nomeadamente *Spouts & Bolts* [10], *Process Function* [8] e *Processor API*, que pertencem ao *Storm*, *Flink* e *Kafka Stream*, respectivamente. Estes elementos têm como objectivo oferecer um maior controlo sobre a construção da lógica de processamento dos dados em *stream*, como por exemplo, gerir o estado nas funções de processamento e fazer as ligações de *streams* entre funções.

Num nível de abstração superior existe um modelo de programação para tornar fluente o desenvolvimentos das topologias, nomeadamente *Streams API* [11], *DataStream API* [7] e *Streams DSL* [9]. Estes modelos são um conjunto de operadores, tais como *Map*, *Filter*, *Aggregation*, *Window* e *Join*, de modo a descrever a topologia desenvolvida para o processamento de dados em específico. A Figura 2 apresenta uma compilação das operações suportadas pelas sistemas em análise.

### 2.2 Topologia de referência

A topologia usada para analisar o desempenho dos diferentes SPDS usa dados de três tipos de dados. O *Gira Travels* representa uma viagem nas bicicletas Gira entre duas estações, contendo o percurso realizado. O *Waze Jams* representa um momento em que foi detectado trânsito numa localidade, dando informações,

| Operadores \ API | *Storm - Streams API* | *Flink - DataStream API* | *Kafka - Streams DSL* |
|---|---|---|---|
| *Map* | ✓ | ✓ | ✓(1) |
| *FlatMap* | ✓ | ✓ | ✓(1) |
| *MapToPair* | ✓ | | |
| *FlatMapToPair* | ✓ | | |
| *KeyBy* | | ✓ | |
| *Filter* | ✓ | ✓ | ✓ |
| *Inverse Filter* | | | ✓ |
| *Fold* | | ✓ | |
| *Reduce* | ✓ | ✓ | ✓ |
| *Aggregate* | ✓ | ✓ | ✓ |
| *ReduceByKey* | ✓ | (2) | |
| *AggregateByKey* | ✓ | (2) | |
| *GroupBy* | | | ✓ |
| *GroupByKey* | ✓ | | ✓ |
| *Window Join* | ✓ | ✓ | ✓ |
| *Window Aggregation* | ✓ | ✓ | ✓ |
| *Window Reduce* | ✓ | ✓ | ✓ |
| *Window Fold* | | ✓ | |
| *Window CoGroup* | ✓(*ByKey*) | ✓ | ✓ |
| *Interval Join* | | ✓ | (3) |
| *Connect* | | ✓ | |
| *CoMap* | | ✓ | |
| *CoFlatMap* | | ✓ | |
| *Split* ou *Branch* | ✓ | ✓ | ✓ |
| *Select* | | ✓ | |

Figura 2: (1) Como a API do *Kafka Streams* tem como base o conceito de tópicos do *Kafka*, estas operações para além de ser possível fazer um mapeamento no *key*/*value*, também permite fazer as mesmas operações somente no *value*; (2) As operações *Aggregate* e *Reduce* já têm ser uma *stream* particionada (*KeyBy*); (3) É possível fazer um *interval join* com as operações normais de *window join* desde que se defina um espaço de tempo anterior ao tempo da *window*

tais como, o atraso, o tamanho, a velocidade, o tipo de estrada, o início e o fim do trânsito e a estrada. O *Waze Irregularities* representa uma situação irregular sobre o trânsito nos locais onde costuma surgir trânsito, fornecendo dados parecidos aos do *Waze Jams*, mas categorizando ainda mais o respectivo acontecimento. O IPMA, dado os respectivos sensores, representa um valor médio numa hora qualquer.

Com o objetivo de avaliar os diferentes SPDS, foi definida uma topologia que tem em conta estes dados. O resultado que se pretende obter durante a execução da topologia é correlacionar as viagens nas bicicletas Gira com os dados de trânsito e irregularidades do *Waze* e os valores médios dos vários sensores das estações meteorológicas. Por forma a enriquecer os dados para a produção final dos resultados deste processamento, é utilizado o tempo de evento do Gira de

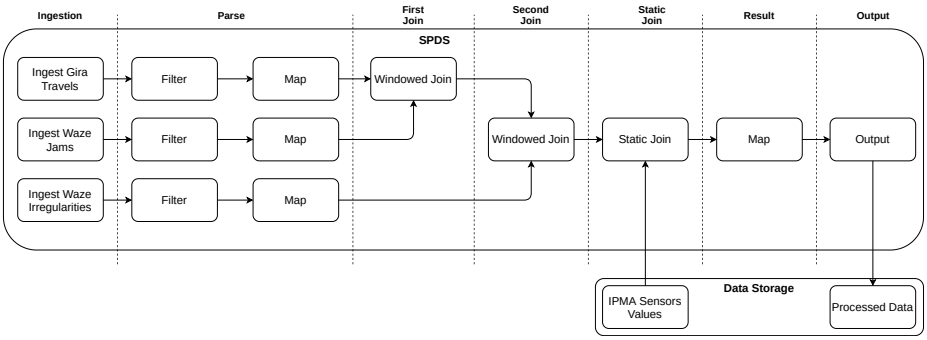



Figura 3: Topologia *Gira Travels Pattern*

forma a adquirir os valores dos vários sensores do IPMA. A Figura 3 apresenta um esquema da topologia a ser implementada e executada em cada SPDS .

A topologia está divida em sete fases funcionais, sendo estas *Ingestion*, *Parse*, *First Join*, *Second Join*, *Static Join*, *Result* e *Output* cobrindo assim vários operadores disponíveis nestes sistemas.

### 2.3 Instalação e Monitorização de métricas

Os SPDS por norma são instalados em modo *cluster*, dando garantias de desempenho e disponibilidade, sendo que são sistemas que podem vir a precisar de imensos recursos computacionais. Em cada sistema, existem componentes que são requisitos para a construção do *cluster*, nomeadamente:

- ***Apache Flink*** - É necessário pelo menos um *Job Manager*, para coordenação de execução distribuída, e pelo menos um *Task Manager* para executar as tarefas que lhe são fornecidas. Para garantir uma maior disponibilidade, é comum ter mais que um *Job Manager*. Neste caso, um *Job Manager* é o *Leader*, enquanto que os restantes estão em *standby*;
- ***Apache Storm*** - É necessário, pelo menos, um nó *Nimbus*, que é responsável pela distribuição do código no *cluster*, atribuir tarefas a outras instâncias e monitorização. É também necessário pelo menos um *Supervisor*, que executa as tarefas atribuídas pelo *Nimbus*. Por fim, é necessário pelo menos um nó com o *Zookeeper* para a coordenação entre *Nimbus* e *Supervisors*;
- ***Apache Kafka*** - É necessária pelo menos uma instância de *Zookeeper* para gerir e coordenar as várias instâncias de *Kafka*, e pelo menos uma instância de *Kafka*, sendo que é recomendado existirem pelo menos três instâncias para replicação de mensagens e tolerância a falhas.

Para que o SPDSBench consiga analisar e visualizar os vários componentes que constituem estes sistemas, são usados um conjunto de ferramentas, nomeadamente, *Telegraf* [16], *InfluxDB* [15] e *Grafana* [12]. O *Telegraf* é um agente que colecta métricas dos mais variados *inputs*, como por exemplo, métricas do

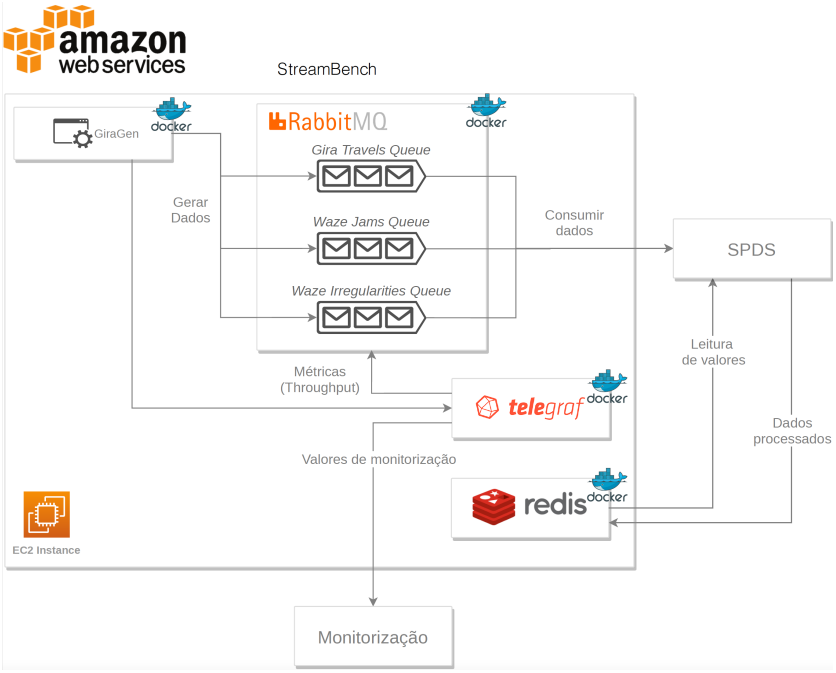



Figura 4: Componentes do *SPDSBench* para geração de dados e monitorização no *AWS*

sistemas (memória, CPU, disco), JMX [20], *StatsD* [6], entre outros com base em *plugins* disponíveis. O *InfluxDB* é um *open source time-series database* que permite manter dados de série temporal, como são os casos de métricas de um sistema. O *Grafana* é uma plataforma *web* que permite visualizar e analisar qualquer tipo de fonte de dados no âmbito de monitorização. Este permite construir painéis para visualização das várias métricas dos vários sistemas e criação de alarmísticas, com recurso ao *InfluxDB* ou outra fonte de dados quaisquer.

## 3 Implantação em *Cloud* e Resultados

O processo de instalação do SPDSBench num *cluster* envolve vários passos antes da topologia de referência, ou outra, poder ser executada. Atualmente o SPDSBench está pronto a ser instalado no fornecedor de *cloud* pública *Amazon Web Services*, como mostra a Figura 4, através de passos automatizados pelas ferramentas *Terraform* e *Ansible*.

O *Terraform* é a ferramenta de *Infrastructure-as-Code* que permite configurar e provisionar as máquinas virtuais do serviço EC2 que dão suporte ao SPDSBench , definindo valores como o tipo de instâncias, o número de instâncias, o tipo e espaço em disco, e configurações da rede. Esta ferramenta também simplifica o processo de escalar horizontalmente e verticalmente as máquinas virtuais,

alterando somente o número instâncias e o tipo de instância que contenham mais recursos computacionais. O *Ansible* é a ferramenta que simplifica a automatização dos processos de instalação dos SPDS a executar, da mesma forma que os restantes componentes do SPDSBench tais como o gerador de dados, *queue*, *data storage* e monitorização nas máquinas virtuais previamente criadas pelo *Terraform*.

Os passos de configuração no AWS são semelhantes aos de outros fornecedores de *cloud* e estão disponíveis em repositório GitHub [3]. Os vários componentes do sistema estão disponíveis em imagens Docker as quais são distribuídas por diferentes máquinas virtuais. No total o sistema usa atualmente 126 vCPU e 252 GB de memória, através de máquinas virtuais do tipo C5 [4]. Este tipo de instâncias beneficiam de CPU de alto desempenho fazendo com que sejam instâncias optimizadas para computação. O processo de instalação possibilita a utilização de 1, 2 ou 4 nós de computação de topologias. Os passos de instalação automática são:

1. Inicializar as instâncias de computação (máquinas virtuais do serviço EC2) e instalar os componentes obrigatórios de cada SPDS .
2. Inicializar instâncias de EC2 e instalar os componentes do SPDSBench para geração de dados, recolha de métricas e monitorização.
3. Migrar os dados dos valores dos sensores do IPMA para o *Redis*.
4. Submeter a topologia *Gira Travels Pattern*, no caso do *Apache Storm* e *Apache Flink*, ou instalar a aplicação que contém a topologia *Gira Travels Pattern*, no caso do *Apache Kafka*;
5. Instalar o *GiraGen* numa instância de EC2;

A execução da topologia nos SPDS teve uma duração entre 10 a 15 minutos, sendo que os resultados extraídos são cerca de 5 minutos da execução. O SPDSBench é utilizado logo após terminar o aprovisionamento das instâncias para instalar e executar a topologia. O máximo *throughput* sustentável obteve-se ao ajustar o *throughput* dos eventos que o *GiraGen* produz para as filas e monitorizando as latências de tempo de evento e de processamento. Caso as latências aumentassem constantemente isso seria indicativo de que, para as condições presentes durante a execução da topologia, seria necessário reajustar novamente o *throughput* do *GiraGen* até que as latências apresentassem valores estáveis. A Tabela 1 apresenta os resultados obtidos em relação ao máximo *throughput* sustentável e os valores usados de *throughput* durante a execução da topologia por cada número de nós.

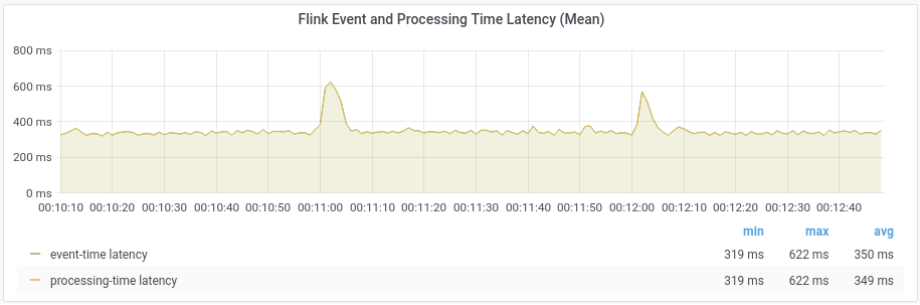
Com a topologia de referência foram executados os diferentes três SPDS. A Figura 5 apresenta os resultados da latência média usando para cada sistema o seu *throughput* sustentável. Por uma questão de espaço é apresentado o resultado para 4 nós computacionais. Os resultados mostram que para a topologia de referência o *Apache Flink* apresenta latências de tempo de evento e de processamento muito inferiores aos restantes, independentemente do número de nós

[3] https://github.com/RubenRibGarcia/infrastructure-and-programming-models-for-stream-data-analysis

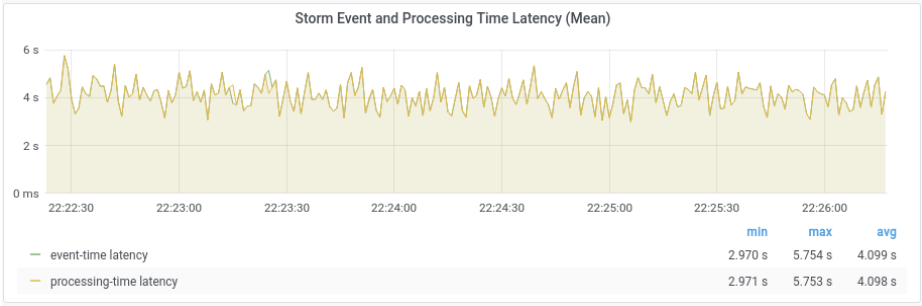
[4] https://aws.amazon.com/pt/ec2/instance-types/c5/

Tabela 1: Tabela de máximo *throughput* sustentável (eventos/s)

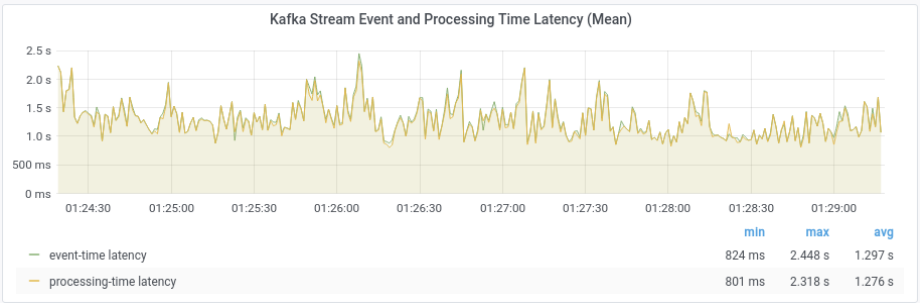
| SPDS - Componente | 1 nó | 2 nós | 4 nós |
|---|---|---|---|
| *Apache Storm - Supervisor* | 3000 | 3500 | 4000 |
| *Apache Flink - Task Manager* | 2850 | 3500 | 4000 |
| *Apache Kafka - Kafka Streams Applications* | 1400 | 1600 | 1900 |




(a) *Flink*




(b) *Storm*

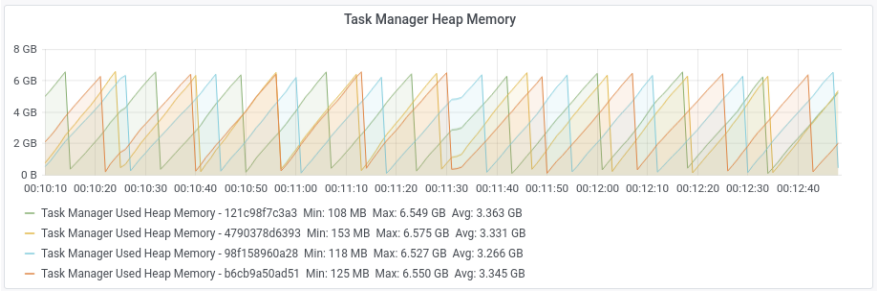



(c) *Kafka Stream*

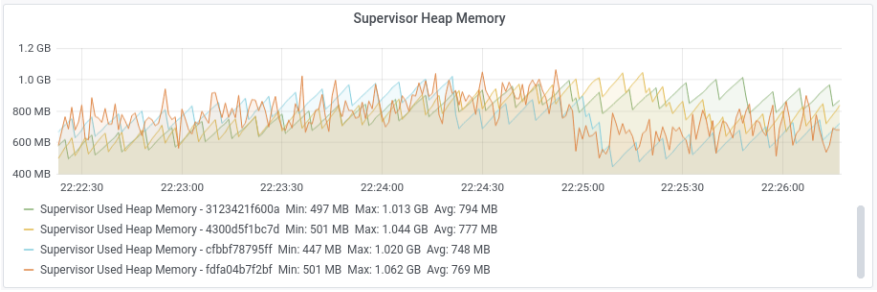
Figura 5: Latência para a execução da topologia de referência usando o *throughput* sustentável

utilizados durante os testes. No *Apache Storm* e no *Apache Kafka* à medida que é aumentado o número de nós, as latências apresentaram uma tendência a aumentar. A Figura 6 apresenta resultados para o uso de memória nos nós de computação.

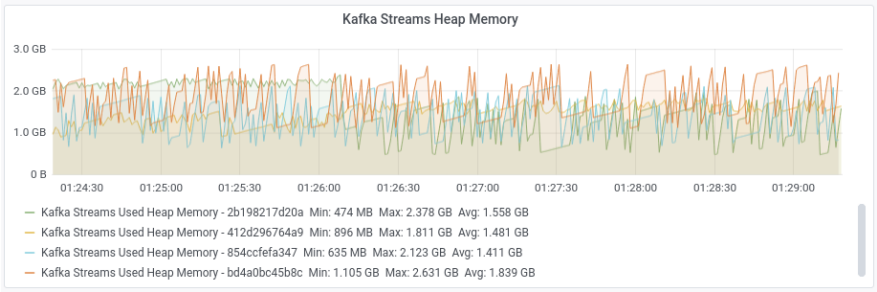
Estes nós foram configurados com 16 GB de memória física. Neste caso o *Apache Flink* é o que ocupa em média mais memória, aproximadamente 3 GB, chegando a passar dos 6 GB, cerca de 3 vezes mais que o *Kafka Stream* e 6 vezes mais que o *Apache Storm*.




(a) *Flink*




(b) *Storm*




(c) *Kafka Stream*

Figura 6: Memória consumida para execução da topologia de referência

## 4 Conclusões e Trabalho Futuro

Este trabalho apresenta a ferramenta SPDSBench e compara três sistema amplamente utilizados para processamento em *stream*, nomeadamente *Apache Storm*, *Apache Flink* e *Apache Kafka*. O SPDSBench é uma infraestrutura de teste extensível e com possibilidade de ser instalada em diferentes ambiente de cloud pública ou privada. Os resultados apresentados foram obtidos executando a ferramenta na AWS. Para trabalho futuro há algumas direçoes interessantes para explorar: (i) estender a ferramenta para conter outro conjunto de SPDS, tais como, *Apache Samza*, *Hazelcast-Jet* e *Apache Heron*; (ii) acrescentar outro tipo de *workloads*, tais como *burst* inicial e periódico no envio de eventos para os SPDS; (iii) realizar o estudo da topologia em relação às latências de tempo de evento e de processamento com ênfase na separação por fases, ou seja, medir as latências entre fases da topologia; (iv) expandir a gestão de infraestrutura para outros fornecedores de *clouds* públicas, nomeadamente, *Google Cloud Plataform* e *Azure*; (v) criar novas topologias de forma a existir uma maior variedade de casos de processamento de dados em *stream*.

## Referências


1. Batyuk, A., Voityshyn, V.: Apache storm based on topology for real-time processing of streaming data from social networks. In: 2016 IEEE First International Conference on Data Stream Mining Processing (DSMP). pp. 345–349 (2016). https://doi.org/10.1109/DSMP.2016.7583573
2. Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., Tzoumas, K.: Apache flink™: Stream and batch processing in a single engine. IEEE Data Eng. Bull. pp. 28–38 (2015)
3. Chintapalli, S., Dagit, D., Evans, B., Farivar, R., Graves, T., Holderbaugh, M., Liu, Z., Nusbaum, K., Patil, K., Peng, B.J., Poulosky, P.: Benchmarking streaming computation engines: Storm, flink and spark streaming. In: 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). pp. 1789–1792 (2016). https://doi.org/10.1109/IPDPSW.2016.138
4. Confluent: Streams Concepts (Dec 2019), https://docs.confluent.io/current/streams/concepts.html
5. Câmara Municipal de Lisboa: *LxDataLab* (Jan 2021), https://lisboainteligente.cm-lisboa.pt/lxdatalabcs/
6. Etsy: StatsD (Sep 2020), https://github.com/statsd/statsd
7. Foundation, A.S.: Flink - DataStream API - Overview (Sep 2020), https://ci.apache.org/projects/flink/flink-docs-release-1.11/dev/datastream_api.html
8. Foundation, A.S.: Flink - Process Function (Sep 2020), https://ci.apache.org/projects/flink/flink-docs-release-1.11/dev/stream/operators/process_function.html
9. Foundation, A.S.: Kafka Streams - Streams DSL (Sep 2020), https://kafka.apache.org/26/documentation/streams/developer-guide/dsl-api.html
10. Foundation, A.S.: Storm - Concepts (Sep 2020), https://storm.apache.org/releases/2.2.0/Concepts.html

11. Foundation, A.S.: Storm - Streams API - Overview (Sep 2020), https://storm.apache.org/releases/2.2.0/Stream-API.html
12. Grafana Labs: Grafana (Sep 2020), https://grafana.com/
13. Hazelcast: Hazelcast Jet (Dec 2019), https://jet-start.sh/
14. Hiraman, B.R., Viresh M., C., Abhijeet C., K.: A study of apache kafka in big data stream processing. In: 2018 International Conference on Information , Communication, Engineering and Technology (ICICET). pp. 1–3 (2018). https://doi.org/10.1109/ICICET.2018.8533771
15. Influxdata: InfluxDB (Sep 2020), https://www.influxdata.com/products/influxdb-overview/
16. Influxdata: Telegraf (Sep 2020), https://www.influxdata.com/time-series-platform/telegraf/
17. Karimov, J., Rabl, T., Katsifodimos, A., Samarev, R., Heiskanen, H., Markl, V.: Benchmarking distributed stream data processing systems. In: 2018 IEEE 34th International Conference on Data Engineering (ICDE). pp. 1507–1518 (2018). https://doi.org/10.1109/ICDE.2018.00169
18. Kulkarni, S., Bhagat, N., Fu, M., Kedigehalli, V., Kellogg, C., Mittal, S., Patel, J.M., Ramasamy, K., Taneja, S.: Twitter heron: Stream processing at scale. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. p. 239–250. SIGMOD '15, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2723372.2742788, https://doi.org/10.1145/2723372.2742788
19. Noghabi, S.A., Paramasivam, K., Pan, Y., Ramesh, N., Bringhurst, J., Gupta, I., Campbell, R.H.: Samza: Stateful scalable stream processing at linkedin. Proc. VLDB Endow. **10**(12), 1634–1645 (Aug 2017). https://doi.org/10.14778/3137765.3137770, https://doi.org/10.14778/3137765.3137770
20. Oracle: Java Management Extensions (JMX) (Sep 2020), https://www.oracle.com/technical-resources/articles/javase/jmx.html
21. Pivotal Software: RabbitMQ (Sep 2020), https://www.rabbitmq.com/
22. Redislabs: Redis (Dec 2019), https://redis.io/
23. Röger, H., Mayer, R.: A comprehensive survey on parallelization and elasticity in stream processing. ACM Comput. Surv. **52**(2) (Apr 2019). https://doi.org/10.1145/3303849, https://doi.org/10.1145/3303849
24. Shahverdi, E., Awad, A., Sakr, S.: Big stream processing systems: An experimental evaluation. In: 2019 IEEE 35th International Conference on Data Engineering Workshops (ICDEW). pp. 53–60 (2019). https://doi.org/10.1109/ICDEW.2019.00-35
25. van Dongen, G., Van den Poel, D.: Evaluation of stream processing frameworks. IEEE Transactions on Parallel and Distributed Systems **31**(8), 1845–1858 (2020). https://doi.org/10.1109/TPDS.2020.2978480
26. Waze: (2021), https://www.waze.com/, [Online; accessed july-2021]