INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Departamento de Engenharia de Electrónica e Telecomunicações e de Computadores

Infraestruturas e Modelos de Programação para Análise de Dados em Stream

Rúben Ribeiro Garcia

Licenciado em Ciência e Engenharia Informática

Dissertação para obtenção do Grau de Mestre
em Engenharia Informática e de Computadores

Orientador : Doutor José Manuel de Campos Lages Garcia Simão

Júri:

Presidente: Professor Doutor Tiago Miguel Braga da Silva Dias

Vogais: Professor Doutor Carlos Jorge de Sousa Gonçalves
Professor Doutor José Manuel de Campos Lages Garcia Simão

Julho, 2021

INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Departamento de Engenharia de Electrónica e Telecomunicações e de Computadores

Infraestruturas e Modelos de Programação para Análise de Dados em Stream

Rúben Ribeiro Garcia

Licenciado em Ciência e Engenharia Informática

Dissertação para obtenção do Grau de Mestre
em Engenharia Informática e de Computadores

Orientador : Doutor José Manuel de Campos Lages Garcia Simão

Júri:

Presidente: Professor Doutor Tiago Miguel Braga da Silva Dias

Vogais: Professor Doutor Carlos Jorge de Sousa Gonçalves
Professor Doutor José Manuel de Campos Lages Garcia Simão

Julho, 2021

Aos meus pais, Dalila e Francisco, por me terem incentivado e apoiado no meu contínuo desenvolvimento académico.

À minha avó (in memoriam) e avô pela preocupação e carinho demonstrado ao longo deste percurso.

À minha esposa, Joana, por ter-me apoiado e suportado ao longo de todo este tempo, e especialmente neste momento tão importante e tão critico.

Agradecimentos

Ao **Sr. Professor José Simão**, pela orientação, competência, profissionalismo e dedicação que tão importantes foram ao longo do desenvolvimento deste trabalho tão importante.

Ao **ISEL**, pela oportunidade de poder dar continuidade aos meus estudos académicos, ao realizar este curso e esta dissertação. Igualmente aos docentes das várias unidades curriculares do Mestrado em Engenharia Informática e de Computadores que realizei no empenho que colocaram no seu ensino.

Aos meus colegas de trabalho, pelo apoio e compreensão na realização do mestrado reconhecendo o esforço necessário na conciliação entre trabalho e estudos.

À minha esposa, Joana Baião, por ter-me apoiado e motivado constantemente durante todo o meu percurso académico, principalmente na realização deste trabalho.

À minha família, por terem estado sempre presentes e preocupados no decorrer do mestrado.

Aos meus amigos, **Ricardo Martinho**, **Rúben Fonseca**, **João Figueira** e **Miguel Barroso**, que mesmo após a licenciatura podemos continuar a partilhar bons momentos e apoiado uns ao outros.

Resumo

O processamento de dados em *batch* tem sido um paradigma no âmbito do *Big Data*, para a análise de grandes quantidades de dados de forma a permitir a transformação e a extracção de conhecimento nos mais variados casos de uso. Este paradigma, de uma forma geral, pode não ser o suficiente quando se pretende que os dados sejam processados o mais rapidamente possível de forma a se adaptarem e reagirem a alterações nos processos de análise dos dados e extracção de valor, uma vez que isto implica que sejam acumuladas grandes quantidades de dados ao longo do tempo, e só depois é que estes dados são enviados para serem processados, fazendo com que gerem enormes latências na obtenção de resultados. Neste trabalho são apresentados conceitos sobre processamento de dados em *stream*, juntamente com a análise de três sistemas de processamento de dados, nomeadamente *Apache Storm*, *Apache Flink* e *Apache Kafka*. De seguida, é apresentada uma proposta de arquitectura desenvolvida de forma a poder avaliar o desempenho de cada sistema e uma topologia de processamento de dados em *stream*, *Gira Travels Pattern*. Adicionalmente é proposto uma forma de automatizar o aprovisionamento da infraestrutura em *clouds* públicas e a instalação de cada sistema de processamento de dados. Segue-se o desenvolvimento da topologia nos sistemas deste trabalho, usando as interfaces de programação de aplicações *Spouts & Bolts*, *DataStream* e *Streams Domain-Specific Language*, respectivamente, realizando uma análise de esforço de desenvolvimento. Por último, esta topologia é executada nos sistemas para obter resultados sobre o desempenho, com base em métricas de latência de tempo de evento e de processamento, uso de CPU, uso de memória e tráfego (transmissão e recepção de pacotes).

Palavras-chave: Arquitecturas distribuídas; Fluxos de dados; Publish-subscribe / Arquitecturas baseadas em eventos; Linguagens de programação geral; Linguagens de programação distribuídas.

Abstract

Batch data processing has been a paradigm, within the scope of Big Data for the analysis of large amounts of data in order to allow the transformation and extraction of knowledge in the most varied use cases. This paradigm, in general, may not be enough when you want the data to be processed as quickly as possible in order to adapt and react to changes in the data analysis and value extraction processes, since this implies that large amounts of data are accumulated over time, and only after that this data is sent to be processed, causing them to generate huge latencies in obtaining results. In this project some concepts about data processing in stream are presented, together with the analysis of three data processing systems, namely Apache Storm, Apache Flink and Apache Kafka. Then, an architecture proposal developed in order to evaluate how these systems are installed in an infrastructure in public clouds, and a topology of stream data processing, Gira Travels Pattern, is presented. Additionally it is proposed a way to automate the provisioning of the infrastructure in public clouds and the installation of each data processing system. Then, the topology is developed in the systems of this project, using the Spouts & Bolts, DataStream and Streams Domain-Specific Language application programming interfaces , respectively, performing a development effort analysis. Finally, this topology is executed in the systems to obtain performance results, based on metrics of event and processing time latency, CPU usage, memory usage and traffic (packet transmission and reception).

Keywords: Distributed architectures; Data streams; Publish-subscribe / Event-Based architectures; General programming languages; Distributed programming languages.

Índice

Índice de Figuras **xvii**

Índice de Tabelas **xxi**

Índice de Listagens **xxiii**

Abreviaturas e Siglas **xxv**

Glossário de Termos **xxvii**

1 Introdução **1**

1.1 Problema 1

1.2 Objectivos 2

1.3 Trabalho Relacionado 2

1.4 Contribuições 3

1.5 Organização do documento 4

2 Estado da Arte e Tecnologias Relacionadas **5**

2.1 Sistemas de processamento de dados em *stream* 5

2.1.1 *Apache Storm* 6

2.1.2 *Apache Flink* 9

2.1.3 *Apache Kafka* 11

2.2 Ferramentas 13
2.3 Monitorização 13
2.4 Estudos comparativos 14
2.5 Elementos qualitativos de análise 16
2.6 Resumo 18

3 Arquitectura **19**
3.1 Elementos quantitativos de análise 19
3.2 Arquitectura proposta 20
3.3 Topologia *Gira Travels Pattern* 23
3.3.1 Definição dos dados 23
3.3.2 *Topologia* 23
3.4 Resumo 26

4 Desenvolvimento **27**
4.1 Infraestrutura 27
4.1.1 Requisitos dos SPDS em *cluster* 27
4.1.2 Implementação 28
4.1.2.1 *GQSM* 29
4.1.2.2 *Monitorização* 30
4.1.2.3 *Apache Storm* 31
4.1.2.4 *Apache Flink* 32
4.1.2.5 *Apache Kafka* 33
4.1.3 Instalação 34
4.2 Implementação da topologia 35
4.2.1 *Apache Storm* 36
4.2.2 *Apache Flink* 38
4.2.3 *Apache Kafka* 40
4.3 Resumo 42

5 Execução e Resultados **43**

5.1 Execução da topologia *Gira Travels Pattern* 43

5.2 Resultados 47

5.2.1 *Apache Storm* 48

5.2.2 *Apache Flink* 54

5.2.3 *Apache Kafka* 60

5.3 Análise qualitativa das API 67

5.4 Análise dos resultados de execução 69

5.5 Resumo 74

6 Conclusões e Trabalhos Futuros **75**

Referências Bibliográficas **77**

A Desenvolvimento da infraestrutura para os SPDS **1**

A.1 Diagrama geral da infraestrutura 2

A.2 Implementação e gestão da infraestrutura dos SPDS com *Terraform* . . . 3

A.2.1 Definicão de Terraform Module para *EC2* 3

A.2.2 Definição de instâncias para o SPDS *Apache Storm* 4

A.2.3 Definição de instâncias para o SPDS *Apache Flink* 6

A.2.4 Definição de instâncias para o SPDS *Apache Kafka* 8

A.3 Implementação da instalação dos SPDS com *Ansible* 11

A.3.1 Definicão de *Ansible Role* para instalação de *Docker containers* com *docker-compose* 11

A.3.2 Definicão de *Ansible Role* para instalação do *Storm* 14

A.3.3 Definicão de *Ansible Role* para instalação do *Flink* 16

A.3.4 Definicão de *Ansible Role* para instalação do *Kafka* 17

A.3.5 Definicão de *Ansible Playbook* para instalação do *Storm* 21

A.3.6 Definicão de *Ansible Playbook* para instalação do *Flink* 22

A.3.7 Definicão de *Ansible Playbook* para instalação do *Kafka* 23

B Resultados **25**

B.1 *Apache Storm* 25

B.2 *Apache Flink* 28

B.3 *Apache Kafka* 31

Índice de Figuras

2.1 Conceito de paralelismo de uma topologia no *Apache Storm* 6
2.2 Conceito de topologia no *Apache Storm* 7
2.3 Componentes de um *Cluster* de *Apache Storm* 8
2.4 Arquitectura do *Apache Flink* 9
2.5 Tarefas e paralelismo no *Apache Flink* 10
2.6 Arquitectura do *Apache Kafka Streams* 12
2.7 *Yahoo! Streaming Benchmark Architecture* 15

3.1 Arquitectura proposta 21
3.2 Esquema dos Dados 24
3.3 Topologia *Gira Travels Pattern* 25

4.1 Infraestrutura GQSM 29
4.2 Infraestrutura Monitorização 30
4.3 Infraestrutura para *Apache Storm* 31
4.4 Infraestrutura para *Apache Flink* 32
4.5 Infraestrutura para *Apache Kafka* 33
4.6 Classe *Observable* 35

5.1 *Apache Storm* - 1 *Supervisor* - Latências 48
5.2 *Apache Storm* - 1 *Supervisor* - CPU e memória 49

5.3 *Apache Storm* - 2 *Supervisor* - Latências 50
5.4 *Apache Storm* - 2 *Supervisor* - CPU e memória 51
5.5 *Apache Storm* - 4 *Supervisor* - Latências 52
5.6 *Apache Storm* - 4 *Supervisor* - CPU e memória 53
5.7 *Apache Flink* - 1 *Task Manager* - Latências 54
5.8 *Apache Flink* - 1 *Task Manager* - CPU e memória 55
5.9 *Apache Flink* - 2 *Task Managers* - Latências 56
5.10 *Apache Flink* - 2 *Task Managers* - CPU e memória 57
5.11 *Apache Flink* - 4 *Task Manager* - Latências 58
5.12 *Apache Flink* - 4 *Task Manager* - CPU e memória 59
5.13 *Apache Kafka* - 1 *Kafka Streams* - Latências 60
5.14 *Apache Kafka* - 1 *Kafka Streams* - CPU e memória 61
5.15 *Apache Kafka* - 1 *Kafka Streams* - *Kafka Node* - CPU e memória 62
5.16 *Apache Kafka* - 2 *Kafka Streams* - Latências 63
5.17 *Apache Kafka* - 2 *Kafka Streams* - CPU e memória 64
5.18 *Apache Kafka* - 2 *Kafka Streams* - *Kafka Node* - CPU e memória 64
5.19 *Apache Kafka* - 4 *Kafka Streams* - Latências 65
5.20 *Apache Kafka* - 4 *Kafka Streams* - CPU e memória 66
5.21 *Apache Kafka* - 4 *Kafka Streams* - *Kafka Node* - CPU e memória 66

B.1 *Apache Storm* - 1 *Supervisor* - Tráfego - Transmissão e Recepção 25
B.2 *Apache Storm* - 2 *Supervisors* - Tráfego - Transmissão e Recepção 26
B.3 *Apache Storm* - 4 *Supervisors* - Tráfego - Transmissão e Recepção 27
B.4 *Apache Flink* - 1 *Task Manager* - Tráfego - Transmissão e Recepção 28
B.5 *Apache Flink* - 2 *Task Managerss* - Tráfego - Transmissão e Recepção 29
B.6 *Apache Flink* - 4 *Task Managers* - Tráfego *In & Out* 30
B.7 *Apache Kafka* - 1 *Kafka Streams* - Tráfego - Transmissão e Recepção 31
B.8 *Apache Kafka* - 1 *Kafka Streams* - *Kafka Node* - Transmissão e Recepção . . . 32
B.9 *Apache Kafka* - 2 *Kafka Streams* - Tráfego - Transmissão e Recepção 33

B.10 *Apache Kafka* - 2 *Kafka Streams* - Tráfego - Transmissão e Recepção 34

B.11 *Apache Kafka* - 4 *Kafka Streams* - Tráfego - Transmissão e Recepção 35

B.12 *Apache Kafka* - 4 *Kafka Streams* - *Kafka Node* - Tráfego - Transmissão e Recepção 36

Índice de Tabelas

2.1 Tabela de comparação dos vários operadores das *API* 17

5.1 Tabela de parametrização para *Apache Storm* 44
5.2 Tabela de parametrização para *Apache Flink* 45
5.3 Tabela de parametrização para *Apache Kafka* 46
5.4 Tabela de máximo *throughput* sustentável (eventos/s) 47
5.5 Tabela de avaliação das API 69
5.6 Tabela de resumo dos resultados obtidos no *Apache Storm* 71
5.7 Tabela de resumo dos resultados obtidos no *Apache Flink* 72
5.8 Tabela de resumo dos resultados obtidos no *Apache Kafka* 73

Índice de Listagens

4.1 *Apache Storm - Spouts & Bolts API* - Inicialização do *TopologyBuilder* . . . 36
4.2 *Apache Storm - Spouts & Bolts API* - Fase de *Ingestion* 36
4.3 *Apache Storm - Spouts & Bolts API* - Fase de *Parse* 37
4.4 *Apache Storm - Spouts & Bolts API* - Fase de *First Join* 37
4.5 *Apache Storm - Spouts & Bolts API* - Fase de *Static Join* 37
4.6 *Apache Storm - Spouts & Bolts API* - Fase de *Result* 38
4.7 *Apache Storm - Spouts & Bolts API* - Fase de *Output* 38
4.8 *Apache Flink - DataStream API* - Inicialização do *StreamExecutionEnvironment* 38
4.9 *Apache Flink - DataStream API* - Fase de *Ingestion* 39
4.10 *Apache Flink - DataStream API* - Fase de *Parse* 39
4.11 *Apache Flink - DataStream API* - Fase de *First Join* 39
4.12 *Apache Flink - DataStream API* - Fase de *Static Join* 40
4.13 *Apache Flink - DataStream API* - Fase de *Result* 40
4.14 *Apache Flink - DataStream API* - Fase de *Output* 40
4.15 *Kafka Streams - Streams DSL* - Inicialização do *StreamsBuilder* 41
4.16 *Kafka Streams - Streams DSL* - Fase de *Ingestion* 41
4.17 *Kafka Streams - Streams DSL* - Fase de *Parse* 41
4.18 *Kafka Streams - Streams DSL* - Fase de *First Join* 41
4.19 *Kafka Streams - Streams DSL* - Fase de *Static Join* 42

4.20 *Kafka Streams - Streams DSL* - Fase de *Result* 42
4.21 *Kafka Streams - Streams DSL* - Fase de *Output* 42
A.1 *Terraform EC2 Module* 3
A.2 Configuração das instâncias para o *Apache Storm* 4
A.3 Configuração das instâncias para o *Apache Flink* 6
A.4 Configuração das instâncias para o *Apache Flink* 8
A.5 *Ansible Role - Docker Compose* 11
A.6 *Ansible Role - Docker Compose - Jinja2 Template* 12
A.7 *Ansible Role - Apache Storm - Nimbus* 14
A.8 *Ansible Role - Apache Storm - Supervisor* 15
A.9 *Ansible Role - Apache Flink - Job Manager* 16
A.10 *Ansible Role - Apache Flink - Task Manager* 17
A.11 *Ansible Role - Apache Kafka - Kafka Node* 17
A.12 *Ansible Role - Apache Kafka - Kafka Connectors* 19
A.13 *Ansible Playbook - Apache Storm* 21
A.14 *Ansible Playbook - Apache Flink* 22
A.15 *Ansible Playbook - Apache Storm* 23

Abreviaturas e Siglas

API	*Application Programming Interface*.
AWS	*Amazon Web Services*.
CPU	*Central Processing Unit*.
CSV	*Comma-separated Values*.
DAG	*Directed Acyclic Graph*.
DSL	*Domain-Specific Language*.
EC2	*Elastic Compute Cloud*.
GB	Gigabyte.
GCP	*Google Cloud Plataform*.
InfluxQL	*Influx Query Language*.
IPMA	Instituto Português do Mar e da Atmosfera.
JMX	*Java Management Extensions*.
JSON	*JavaScript Object Notation*.
JVM	*Java Virtual Machine*.
KSQL	*Kafka Structured Query Language*.
ms	Milissegundo.
NVMe	*Non-Volatile Memory Express*.
OGC	*Open Geospatial Consortium*.
POJO	*Plain Old Java Object*.
SPDS	Sistemas de Processamento de Dados em *Stream*.
SQL	*Structured Query Language*.
SSD	*Solid State Drive*.
WKB	*Well-Known Binary*.

Glossário de Termos

Big Data	área do conhecimento que estuda como tratar, analisar e obter informações a partir de conjunto de dados grandes demais para serem analisados por sistemas tradicionais.
batch	um grupo de dados processados como uma única unidade.
latência	diferença entre o momento em que um evento é gerado na fonte e o momento em que este evento produz algum resultado.
throughput	quantidade de dados, por exemplo, processados em um determinado espaço de tempo.
bottleneck	ocorrência de uma limitação na capacidade de processamento de um componente.
cache	é um componente de hardware ou software que permite guardar dados para que futuros pedidos desses dados possam ser acedidos mais rapidamente.

1

Introdução

O processamento de dados em *batch* tem sido um paradigma no âmbito do *Big Data*, para a análise de grandes quantidades de dados de forma a permitir a transformação e a extracção de conhecimento nos mais variados casos de uso. Este paradigma, de uma forma geral, pode não ser o suficiente quando se pretende que os dados sejam processados o mais rapidamente possível de forma a se adaptarem e reagirem a alterações nos processos de análise dos dados e extracção de valor, uma vez que isto implica que sejam acumuladas grandes quantidades de dados ao longo do tempo, e só depois é que estes dados são enviados para serem processados, fazendo com que gerem enormes latências na obtenção de resultados. Dado o crescimento significativo no paradigma do processamento de dados em *stream*, têm vindo a surgido vários sistemas para este fim, tais como *Apache Storm* [1], *Apache Flink* [2], *Apache Kafka* [3], *Apache Heron* [4], *Apache Samza* [5] e *Hazelcast-Jet* [6].

1.1 Problema

Dado o conjunto alargado de Sistemas de Processamento de Dados em *Stream* (SPDS) existentes hoje em dia, escolher um deste sistemas pode tornar-se difícil dada a vasta popularidade que estes têm vindo a adquirir ao longo do tempo. Mas a popularidade pode não ser 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 partem, por um lado entender que API estes sistemas são disponibilizadas de forma a tornar mais ou menos complexo expressar as transformações a executar sobre os dados. Cada sistema apresentam vários níveis de abstracção das suas API 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 *Structured Query Language* (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 ou verticalmente estes sistemas.

1.2 Objectivos

Este trabalho tem os seguintes objectivos:

- Enquadrar os conceitos associados ao processamento de dados em *stream*;
- Comparar os SPDS *Apache Storm*, *Apache Flink* e *Apache Kafka* quanto às suas arquitecturas e modelos de programação;
- Apresentar a arquitectura desenvolvida para execução de uma topologia, juntamente com a natureza dos dados usados;
- Mostrar os desenvolvimentos realizados relativamente à gestão da infraestrutura e instalação dos SPDS em *clouds* públicas, a implementação da topologia e comparar e avaliar o desempenho de cada SPDS.

1.3 Trabalho Relacionado

Alguns dos trabalhos realizados no âmbito da avaliação e comparação entre os vários SPDS existentes têm como foco principal o estudo do desempenho destes sistemas, em que expõem a natureza dos dados e descrevem qual a topologia a ser executada, tais como os trabalhos de *Chintapalli* et al. [7], de *Shahverdi* et al. [8] e de *Dongen* e *Poel* [9]. Além disso, apresentam uma infraestrutura e configurações usadas durante a execução da topologia sendo possível medir valores de desempenho dos sistemas.

Um outro trabalho realizado por *Karimov* et al. [10] propõe uma nova abordagem a ser aplicado durante o estudo de desempenho dos sistemas, 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*.

Os trabalhos mencionados anteriormente, à excepção deste trabalho e do trabalho realizado por *Dongen* e *Poel*, não aplicam o conceito apresentado por *Karimov* et al. visto que a topologia usada para o estudo do desempenho dos sistemas, contém pelo menos uma operação com estado. 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.

1.4 Contribuições

Tendo em conta os trabalhos realizados no âmbito da avaliação e comparação entre vários SPDS, este trabalho contribui com:

1. Análise entre 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 SPDSs e outras topologias;
3. Disponibilização de uma arquitectura como referência a trabalhos futuros com a possibilidade de substituição por outros componentes;
4. Simplificação dos processos de instalação destes sistemas, bem como gestão de infraestrutura de modo a garantir a sua replicabilidade e extensibilidade;
5. Estudo do desempenho dos SPDSs, aplicando o conceito realizado no estudo por *Karimov* et al. [10], em tempo real;

Com base nestes contributos, e cumprindo com os objectivos deste trabalho, é implementada a topologia sobre padrões de uso das bicicletas Gira em que a fonte de dados é originado do desafio "Existem padrões de utilização das bicicletas partilhadas em Lisboa ?"pela LxDataLab [11], *Gira Travels Pattern*. Estes dados são compostos por viagens

realizadas nas bicicletas Gira, juntamente com informação sobre o trânsito e irregularidades do *Waze* e os valores médios dos sensores do Instituto Português do Mar e da Atmosfera (IPMA). Para além do desenvolvimento 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.

1.5 Organização do documento

Este documento está organizado da seguinte forma:

- Capítulo 2, apresenta alguns conceitos envolvidos no processamento de dados em *stream* dando a conhecer os SPDS usados neste trabalho;
- Capítulo 3, apresenta uma proposta de arquitectura implementada para este trabalho, com foco nos aspectos quantitativos de análise dos SPDS;
- Capítulo 4, demonstra os respectivos desenvolvimentos sobre a implementação da topologia *Gira Travels Pattern* nos SPDS e como é que os SPDS são instalados;
- Capítulo 5, apresenta e discute os resultados obtidos durante a execução da topologia nos SPDS;
- Por último, Capítulo 6, são apresentas as conclusões deste trabalho;

Todos os desenvolvimentos realizados no âmbito deste trabalho, estão disponíveis num repositório *Git* público denominado *Github* [https://github.com/RubenRibGarcia/infrastructure-and-programming-models-for-stream-data-analysis].

2

Estado da Arte e Tecnologias Relacionadas

Neste capítulo pretende-se contextualizar os conceitos envolvidos no processamento de dados em *stream*, realizando uma análise em alguns dos SPDS. São apresentadas também algumas ferramentas e sistemas que permitem simplificar o processo de instalação dos SPDS e outros sistemas, gestão da infraestrutura e monitorização. Depois, é feita uma análise dos vários estudos de *benchmarking* desenvolvidos, e por último a análise das características quantitativas e qualitativas de cada sistema.

2.1 Sistemas de processamento de dados em *stream*

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 [12]. Uma topologia é um grafo orientado sem ciclos (*Directed Acyclic Graph* (DAG)) de operações que processam e reencaminham os dados provenientes de uma *stream*, sejam recebidos ou transmitidos entre sistemas externos, ou entre operações.

Para que seja possível lidar com grandes quantidades de dados sem comprometer a qualidade do serviço, estes sistemas apresentam métodos de paralelização do processamento dos dados para evitar que com o aumento do *throughput* sobre estes sistemas, as latências do processamento dos dados não aumentem.

Os SPDS escolhidos para este trabalho foram decididos com base em documentação disponível *online*, contributos feitos pela comunidade *open-source*, clareza nas API fornecidas, maturidade e tendências, sendo estes o *Apache Storm*, *Apache Flink* e *Apache Kafka*.

2.1.1 *Apache Storm*

O *Apache Storm* é um sistema de computação distribuído em tempo real para computação sobre fluxos de dados limitados e ilimitados. Conceptualmente, o *Storm* é constituído por:

- ***Streams***, que representam a principal abstracção do *Storm*;
- ***Tuples***, ou tuplos, que representam a estrutura principal que é enviado entre *streams*;
- ***Tasks***, ou tarefas, que representam a execução da lógica implementada nos *Spouts* e *Bolts*. Existem tantas *tasks* a serem executadas quanto o número definido do paralelismo pretendido por *Spouts* e *Bolts* na topologia;
- ***Streams Groupings***, que representam a forma de enviar dos tuplos entre *tasks*, como por exemplo, aleatoriamente (*Shuffle Grouping*) ou particionado por um campo de um tuplo (*Fields Grouping*). A Figura 2.1 representa genericamente as ligações entre as várias *tasks* e respectivo paralelismo.

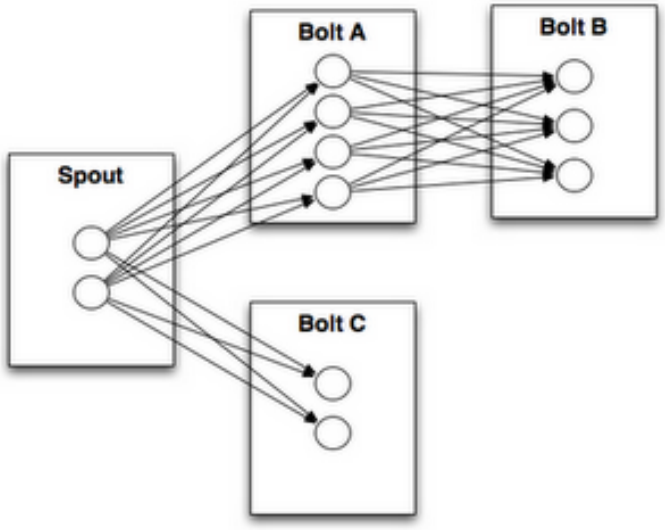


Figura 2.1: Conceito de paralelismo de uma topologia no *Apache Storm* [13]

- ***Workers***, que representam um processo que executa um sub-conjunto de todas as *tasks* de uma topologia.
- ***Spouts***, que representam a lógica implementada de consumo dos dados e que depois são emitidos para uma ou mais *streams*;
- ***Bolts***, que representam a lógica do processamento dos dados que têm como origem uma ou mais *streams*;
- ***Topologias***, que representam a lógica do processamento dos dados em *stream*. A topologia na sua essência é um grafo de *spouts* e *bolts* que estão agrupados por *streams*, representado na Figura 2.2;

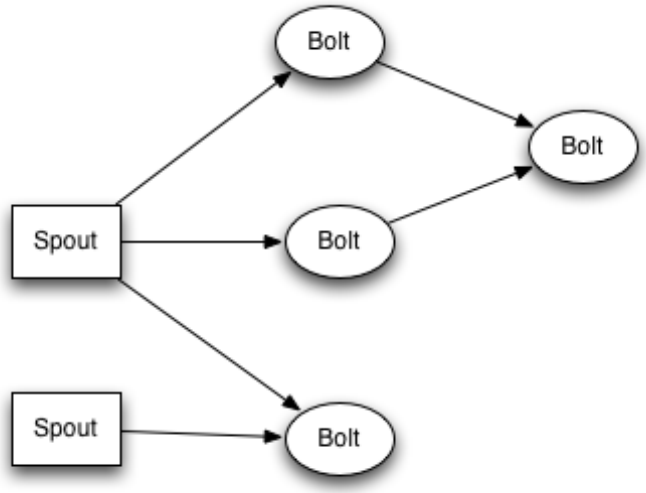


Figura 2.2: Conceito de topologia no *Apache Storm* [13]

O *Apache Storm* é composto pelos seguintes componentes:

- ***Nimbus***, que tem como responsabilidade distribuir a topologia desenvolvida e atribuir *tasks* aos *Supervisors*;
- ***Supervisor***, que tem como responsabilidade supervisionar a própria instância de execução da topologia submetida pelo *Nimbus* e executar os vários *workers* para submissão das *tasks*.
- ***Zookeeper***, que tem como responsabilidade coordenar a comunicação entre as instâncias *Nimbus* e *Supervisors*, mantendo todo o estado necessário de forma a garantir tolerância a falhas;

A Figura 2.3 representa a composição apresentada anteriormente, incluindo a comunicação entre os componentes:

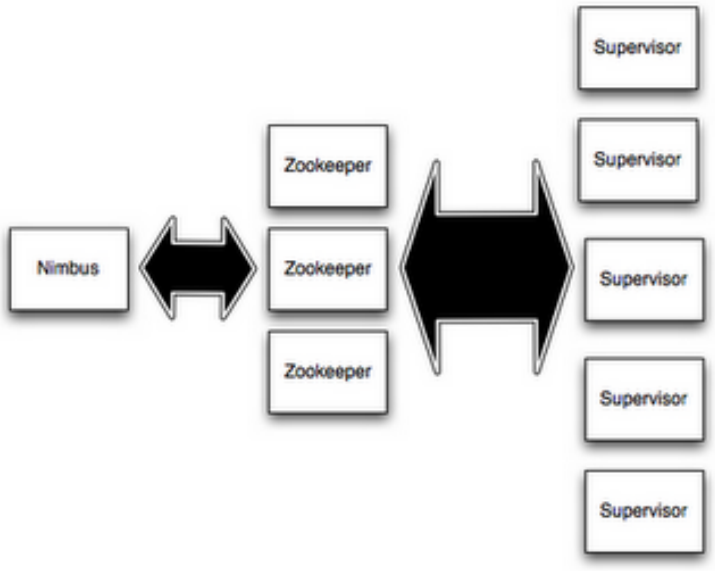


Figura 2.3: Componentes de um *Cluster* de *Apache Storm* [13]

2.1.2 *Apache Flink*

O *Apache Flink* é uma *framework* e motor de processamento distribuído (*distributed processing engine*) para computação com estado sobre fluxos de dados limitados e ilimitados. O *Flink* é composto por dois tipos de processos:

- ***Job Managers***, que coordenam a execução distribuída sobre as tarefas a realizar. Este gestor agenda tarefas, coordena *checkpoints* e coordena recuperação de falhas.
- ***Task Managers***, executam tarefas de um *dataflow*.

Para um melhor entendimento do funcionamento das aplicações desenvolvidas com o *Apache Flink*, representado na Figura 2.4, este encontra-se divido nos seguintes componentes:

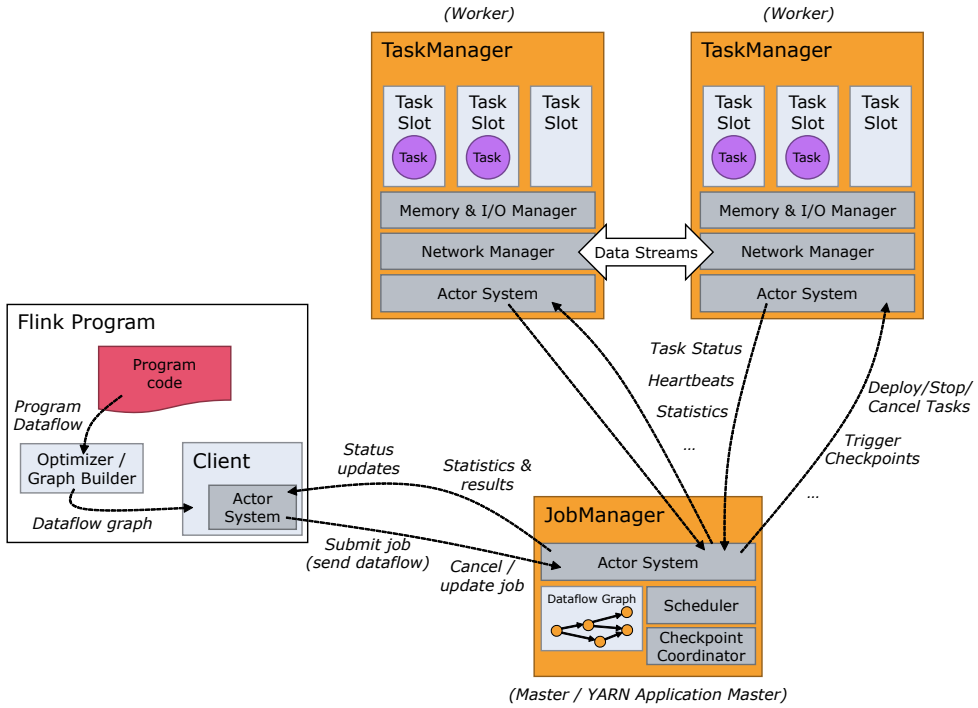


Figura 2.4: Arquitectura do *Apache Flink* [14]

- ***Job Manager***, que é responsável por analisar a topologia submetida e coordenar as *tasks*, com base nas operações definidas pela topologia, submetidas aos *Tasks Managers*;
- ***Task Manager***, que é responsável por executar as operações que lhe foram incumbidas pelo *Job Manager*;

- ***Flink Program***, que é a topologia definida no *Flink* e poderá ser submetida para o *Job Manager*;
- ***Data Streams***, que é a ligação entre *Task Managers* quando é pretendido transferir partes dos dados para outro nó que contém outro conjunto de operações a serem aplicadas com base na topologia definida.

A Figura 2.5 apresenta uma visão de como é que o *Apache Flink* realizar o fluxo entre operações definidas numa topologia e respectivo paralelismo:

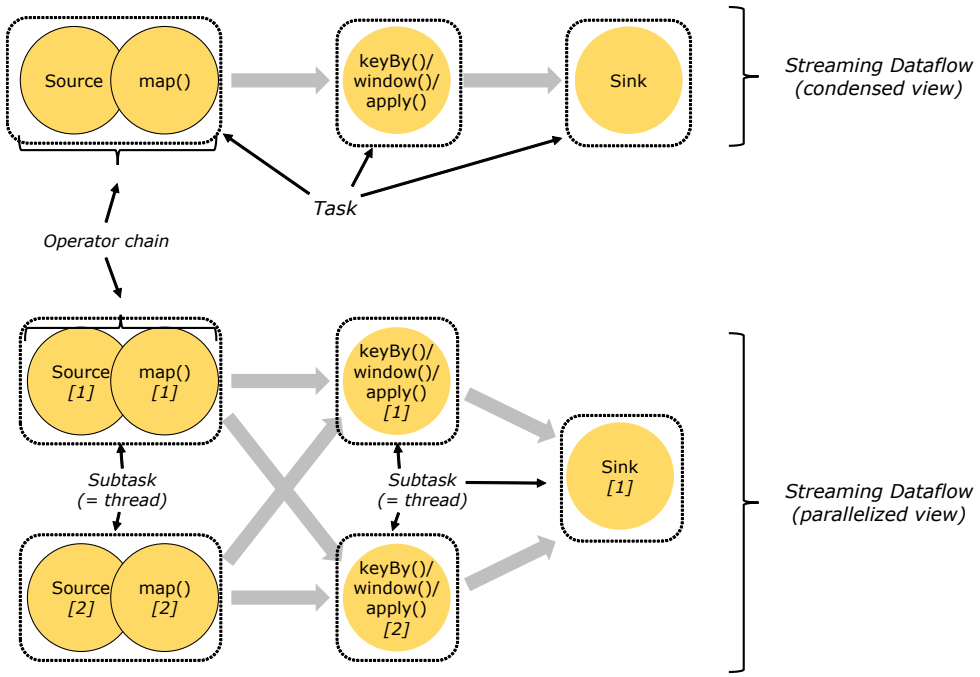


Figura 2.5: Tarefas e paralelismo no *Apache Flink* [15]

2.1.3 *Apache Kafka*

O *Apache Kafka* é uma plataforma de *streaming* distribuído (*distributed streaming plataform*) que pode desempenhar os seguintes papéis:

- **Sistema de Mensagens**, principalmente sobre o modelo de filas de espera (*queue*) ou modelo *publish-subscribe*;
- **Sistema de Armazenamento**;
- **Processamento em *Stream***.

Para que o *Kafka* possa desempenhar estes papéis, este disponibiliza 4 API fundamentais:

- ***Producer API***, que permite que qualquer aplicação possa publicar uma *stream* de dados a um ou mais tópicos no *Kafka*;
- ***Consumer API***, que permite que qualquer aplicação possa subscrever a um ou mais tópicos e processar uma *stream* de dados;
- ***Streams API***, permite que uma aplicação possa agir como um processador de *streams*, consumindo uma *stream* de um ou mais tópicos e produzindo os resultados desse mesmo processamento para um outro tópico qualquer;
- ***Connector API***, que permite construir produtores, ou consumidores, que conectam um tópico do *Kafka* a um sistema externo.

Para o caso de estudo em questão a API mais importante é a *Streams API*, que passaremos a designar como *Kafka Streams* [16].

O *Kafka Streams* é uma biblioteca que permite desenvolver aplicações que desempenham um papel de processadores de *streams*, sendo que todos os dados de entrada e saída são guardados no *cluster* do *Kafka*. As aplicações desenvolvidas com esta API definem uma topologia, que é a forma de descrever quais são as operações a realizar no processamento de dados em *stream*.

Para um melhor entendimento relativamente ao funcionamento das aplicações desenvolvidas com o *Kafka Streams*, representado na Figura 2.6, este encontra-se dividido nos seguintes componentes:

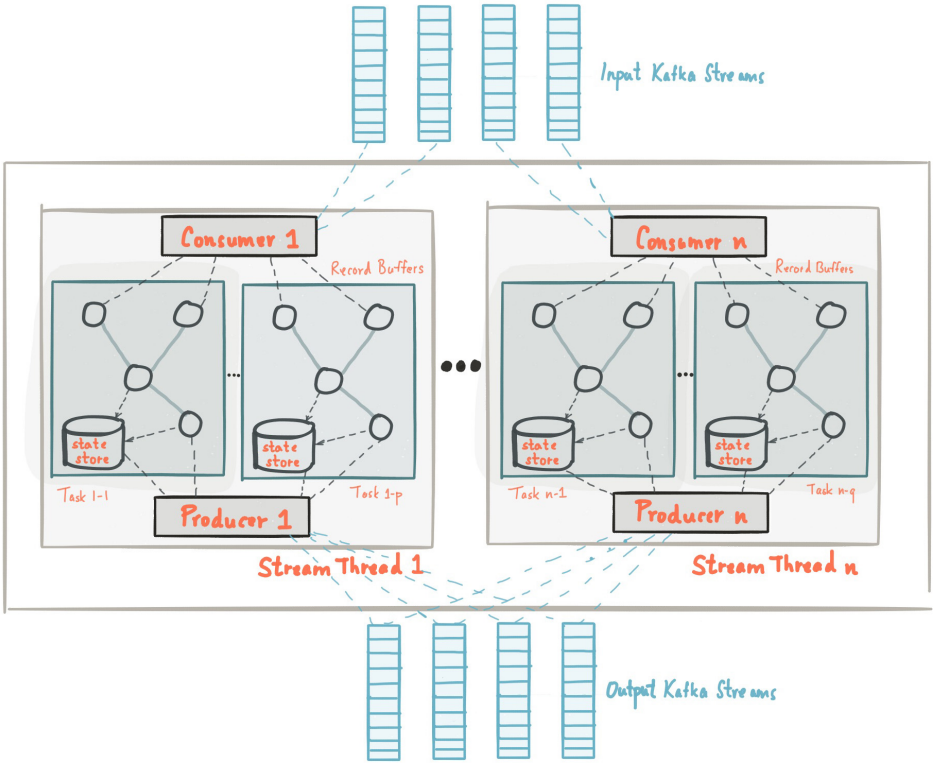


Figura 2.6: Arquitectura do *Apache Kafka Streams* [17]

- ***Stream thread***, que representa o conceito para a paralelização do processamento em *stream* das topologias desenvolvidas no *Kafka*, sendo que os dados vêm a partir dos tópicos dos quais pretendemos consumir;
- ***Tasks***, que representa o conceito de execução das operações definidas pela topologia e que são criadas conforme o número de partições definidas nos tópicos que este consuma. Cada *task* contém um *record buffer* associado dado o particionamento aplicado;
- ***State Store***, que representa o conceito de estado para quando existem operações na topologia que necessitam de manter estado, por exemplo, junções, agregações com *windowing*;
- ***Producer-Consumer***, que representa o conceito de entrada e saída dos dados processados para tópicos do *Kafka*.

2.2 Ferramentas

De forma a simplificar a instalação e gestão dos sistemas e infraestrutura beneficiando da possível replicabilidade e dos automatismos, existe um conjunto de ferramentas que cumpre estes objectivos, nomeadamente, *Docker* [18], *Terraform* [19] e *Ansible* [20].

O *Docker* é uma tecnologia de *containers* para a distribuição e execução de aplicações, que abstrai a camada aplicacional empacotando a aplicação e respectivas dependências, fazendo com que cada processo possa ser executado isoladamente numa mesma máquina. Em comparação às máquinas virtuais, os *containers* são muito mais leves e não sofrem das mesmas penalização que uma máquina virtual. O *Docker Compose* [21] é uma ferramenta para definir e executar *multi-containers Docker*.

O *Terraform* é uma ferramenta de infraestrutura como código para fornecer e gerir *clouds*, infraestruturas ou serviços, com base num conjunto de *providers*, tais como *Google Cloud Plataform*, *Amazon Web Services* e *Azure*. Cada *provider* fornece um conjunto de configurações que irão representar os vários recursos que estes possam vir a disponibilizar, por exemplo, máquinas virtuais, bases de dados, entre outros serviços.

O *Ansible* é uma ferramenta que permite automatizar processos de instalação, gestão de sistemas e aplicações em infraestruturas. Esta ferramenta apresenta uma sintaxe com base em *YAML* que permite definir os passos a executar de um processo (*role*) e em que instâncias (*playbook*).

2.3 Monitorização

Para que seja possível analisar, visualizar e reagir com base em alarmísticas sobre o estado de qualquer sistema, é estritamente necessário que estes sejam monitorizados com base em métricas que estes disponibilizem. Como tal, existe um conjunto de ferramentas que permite cumprir com estes requisitos, nomeadamente, *Telegraf* [22], *InfluxDB* [23] e *Grafana* [24].

O *Telegraf* é um agente que colecta métricas dos mais variados *inputs*, como por exemplo, métricas do sistemas (memória, CPU, disco), *Java Management Extensions* (*JMX*) [25], *StatsD* [26], 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.

2.4 Estudos comparativos

Dado o crescimento da popularidade que os SPDS têm vindo a adquirir cada vez mais, escolher um destes sistemas torna-se complexo no sentido de saber qual o sistema que oferece o melhor conjunto de características e qual o sistema que tem o melhor desempenho, dependendo de um caso em concreto.

Isto leva a que hajam alguns trabalhos de investigação que façam estas comparações de características e desempenho, e outros que propõem formas mais precisas de realizar estas mesmas medidas, independentemente do sistema em questão. Em todos estes trabalhos existem duas medidas que são avaliadas, nomeadamente a latência e o *throughput*.

Em relação às características de alguns SPDS, existe um estudo [27] que apresenta uma avaliação compreensiva sobre alguns aspectos mais relevantes, tais como, modelos de programação, tipo de infraestrutura e métodos de paralelização de operadores.

O *benchmark* desenvolvido na *Yahoo!* por *Chintapalli* et al. [7] desenvolveu e disponibilizou um *benchmark* [28] com o objectivo de simular um caso real. Os sistemas avaliados pela *Yahoo!* foram o *Apache Storm*, *Apache Flink* e *Apache Spark Streaming*, simulando uma análise de publicidade em que existe um número de campanhas e um número de publicidades por campanha. A *pipeline* para o processamento destes dados tem início no *Kafka*, em que são lidos os eventos no formato *JSON*. De seguida, são identificados os eventos relevantes e registada uma contagem de eventos por campanha no *Redis* [29], como está representado na Figura 2.7.

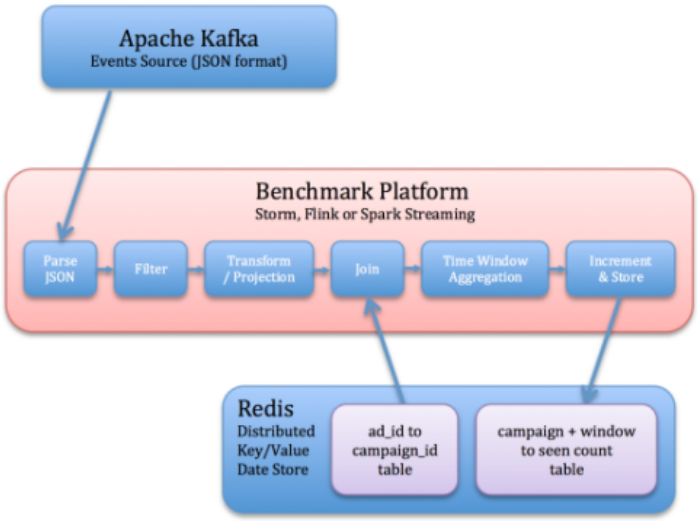


Figura 2.7: *Yahoo! Streaming Benchmark Architecture* [7]

O *benchmark* desenvolvido na Universidade de Taru por *Shahverdi* et al. [8] baseou-se no *benchmark* da *Yahoo!*, adicionando mais dois SPDS, nomeadamente *Apache Kafka*, usando a biblioteca *Kafka Streams* e *Hazelcast Jet* [6], desenvolvendo a mesma topologia e usando a mesma fonte de dados. Para além desta adição, os restantes SPDS foram actualizados para versões mais recentes como também a execução dos testes foram realizados em *clouds* públicas, nomeadamente a *DigitalOcean*. Um outro estudo realizado por *Marcu* et al. [30] apresenta uma análise entre *Flink* e *Spark*, correlacionando o plano de execução dos operadores com o uso de recursos computacionais. Apesar de na análise realizada sobre um conjunto de *workloads* terem sido usadas as API de processamento em *batch*, a obtenção das latências de execução é repartida pelas operações que estes sistemas decidem agregar dado a sequência de operações aplicadas.

Apesar destes *benchmarks* terem como objectivo principal a simulação de um caso real, o estudo realizado por *Karimov* et al. [10] refere que a maioria dos *benchmarks* realizam uma medição imprecisa da latência sobre operações com estado, como por exemplo operações de *join* e *windowing*, e não existe uma separação clara entre o sistema em teste e os vários componentes para a realização do *benchmark*, visto que são estes sistemas que medem e calculam os valores que irão servir como métricas. Posto isto, a proposta deste estudo cria um conjunto de definições de como medir a latência em tempo de evento e de processamento em operadores de *windowing joins* e operadores com *windowing*, nomeadamente agregações. Também é definido onde deve ser feita a medição do *throughput*.

Por último, *Dongen* e *Poel* [9] deram o seu contributo com um novo *benchmark open-source* [31] que mede a latência com precisão das *pipelines* com diferentes complexidades para alguns dos SPDS do estudo. O trabalho analisa a relação entre latência e o *throughput* com os recursos que cada SPDS usa, entre diferentes tipos de *workloads*, como determinação da menor latência, *sustainable throughput*, *burst* inicial de *throughput* e *bursts* periódicos. Também dedica um conjunto de parâmetros utilizado para cada *workload* e alterna o uso de operadores com estado nativos dos SPDS e customizados.

2.5 Elementos qualitativos de análise

Em cada SPDS, existe um conjunto de características e funcionalidades que podem levar a que seja possível decidir qual o sistema que mais se adequa a um determinado caso em que se pretenda realizar o processamento de dados em *stream*.

Os SPDS abordados neste trabalho são compostos por um conjunto de API de mais baixo nível, nomeadamente *Spouts & Bolts* [32], *Process Function* [33] e *Processor API*, que pertencem aos SPDS *Storm*, *Flink* e *Kafka* respectivamente. Estas API têm como objectivo oferecer um maior controlo sobre a construção da lógica de processamento dos dados em *stream*, por exemplo, gerir o estado nas funções de processamento e fazer as ligações "manualmente"de *streams* entre funções, e também servirem como base para as restantes API de mais alto nível. Em relação ás ligações realizadas de *streams* entre funções, a API *Spouts & Bolts* permite definir como é realizada a distribuição dos dados entre *Bolts*, nomeadamente, distribuição aleatória entre as várias tarefas de um *Bolt* e distribuição particionada por valores dos dados a processar. Esta API ainda disponibiliza uma interface, *CustomStreamGrouping*, que permite implementar outros tipos de distribuição dos dados entre as várias tarefas de um *Bolt*.

O próximo nível de API nos SPDS abstrai a camada anterior de modo a fornecer uma API mais fluente no desenvolvimentos das topologias, nomeadamente *Streams API* [34], *DataStream API* [35] e *Streams DSL* [36]. Esta camada apresenta um conjunto de métodos com base em operadores base de *streams*, tais como *Map*, *Filter*, *Aggregation*, *Window* e *Join*, de modo a descrever a topologia desenvolvida para o processamento de dados em específico. A Tabela 2.1 apresenta uma compilação das operações suportadas pelas API descritas anteriormente.

O último nível de API que estes SPDS apresentam é um nível de abstracção muito mais alto que pode ser representado com base em expressões *Structured Query Language* (SQL), nomeadamente *Storm SQL* [37], *Table API & SQL* [38], e *KSQL* [39].

Tabela 2.1: Tabela de comparação dos vários operadores das *API*

API / Operadores	*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		✓	

(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) - No *Flink*, as operações *Aggregate* e *Reduce* já têm ser uma *stream* particionada(*KeyBy*), e não necessita de ser em *window*.
(3) - No *Kafka Streams* 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*

Os SPDS em análise permitem configurar o processamento dos dados com base em tempo de evento ou de processamento. Esta configuração é bastante importante uma vez que características de tempos diferentes podem obter resultados diferentes. Um exemplo desta diferença está relacionado com os operadores como *windowing* e *windowing join* em que, no caso do processamento ser com base em tempo de processamento, dependendo do tamanho da janela, estes operadores são executados quando passam exactamente o tamanho definido. Quando o processamento estiver configurado com base em tempo de evento então os operadores anteriores só são executados quando estes receberem o evento que coincide com o tempo esperado naquela mesma janela. Estes SPDS também permitem realizar o processamento dos dados em *stream* com a característica de *exactly-once* ou *at-least-once*. A característica *exactly-once* permite que durante o processamento dos dados, exista a garantia de que aquele evento foi processado uma única vez, enquanto que em *at-least-once* o mesmo evento pode ser processado mais que uma vez, no entanto é garantido que foi pelo menos processado.

Por fim, estes SPDS disponibilizam um conjunto de integrações, sobre a forma de bibliotecas, com outros sistemas externos de forma a simplificar o processo de leitura ou de escrita para estes mesmos sistemas, por exemplo, integração com sistemas de bases de dados relacional e não relacional, integração com *message queues* ou *message brokers*.

2.6 Resumo

Os conceitos apresentados sobre sistemas de processamento de dados em *stream* , analisando os SPDS *Apache Storm*, *Apache Flink* e *Apache Kafka*, permitiram adquirir conhecimentos relativamente às características que cada SPDS apresenta. Ao mesmo tempo, foram apresentados alguns pontos relevantes para a análise qualitativa de cada SPDS. Por último, esta análise contribuiu para o desenho da arquitectura do sistema de benchmark.

3

Arquitectura

Neste capítulo pretende-se apresentar uma proposta de arquitectura de como os vários componentes interagem entre si, juntamente com elementos de análise quantitativa a realizar e explicar quais as razões das decisões tomadas durante a definição desta arquitectura. De seguida, são enquadrados os tipos de dados usados no processamento, nomeadamente *Gira Travels*, *Waze Jams*, *Waze Irregularities* e valores dos sensores do IPMA originado de um desafio "Existem padrões de utilização das bicicletas partilhadas em Lisboa ?"pela LxDataLab [11]. Por fim, é apresentada a topologia desenvolvida, *Gira Travels Pattern*, com base nos dados referidos anteriormente.

3.1 Elementos quantitativos de análise

Em todos os SPDS, existe uma noção quantitativa, nomeadamente de tempo e *throughput*. O tempo destes sistemas está dividido em três tipos:

- **Tempo do evento**, representa o tempo em que um evento é gerado/criado;
- **Tempo de ingestão**, representa o tempo quando o evento entra no fluxo do processamento em *streams*;
- **Tempo de processamento**, representa o tempo em que o evento, após entrar no fluxo do processamento em *streams*, acaba de ser processado.

O *throughput* representa a quantidade de dados processados, por unidade de tempo.

Tendo em conta as definições anteriores, para fins de análise de cada sistema, tipicamente existem duas métricas importantes, sendo estas a latência e o *throughput*. A latência, proposta realizada por *Karimov* et [10], está divida em duas categorias, latência de tempo de evento (*event-time latency*) e latência de tempo de processamento (*processing-time latency*). A latência de tempo de evento define-se como o intervalo de tempo entre a criação dum evento e o tempo de emissão no operador de saída dos dados. A latência de tempo de processamento define-se como o intervalo de tempo entre o tempo de ingestão e o tempo de emissão no operador de saída dos dados. A latência de tempo de processamento faz parte da latência de tempo de evento.

Ainda em relação ao estudo, os operadores de *windowing* precisam de ser analisados de outra forma, pois estes operadores acumulam estado dos eventos e devolvem quando tiver passado um número de eventos ao longo do tempo, logo a latência é afectada pelo tempo de espera até a janela estar completa. Como tal, para medir a latência de tempo de evento em operadores *windowing*, o tempo de evento a usar deve ser o máximo do tempo de evento de todos os eventos que contribuíram para a saída desses mesmo eventos. Para medir a latência de tempo de evento em operadores *windowing join* a medida é realizada com o mesmo intuito que os operadores *windowing*, mas no momento de juntar os eventos, é maximizado o tempo entre um evento e outro que resultaram nessa mesma junção. A latência de tempo de processamento nos operadores de *windowing join* e *windowing* é realizada da mesma forma, só que em vez de usar o tempo de evento dos eventos, é usado o tempo de processamento.

Por último, o mesmo estudo apresenta uma medida em relação ao *throughput* que é o máximo *throughput* sustentável. Este permite avaliar nos SPDS que apresentam mecanismos de *backpressure*, qual o *throughput* máximo que este permite sem que as latências de tempo de evento e de processamento aumentem linearmente.

3.2 Arquitectura proposta

A arquitectura proposta, presente na Figura 3.1, é 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* [40], 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* [29], que será responsável por manter os dados do IPMA e receber os dados processados pelos SPDS;

- **Monitorização**, composta pelo *Telegraf* [22], *InfluxDB* [23] e *Grafana* [24] que será responsável por recolher métricas dos SPDS, a nível das latência do tempo de evento e do tempo de processamento, utilização da memória e CPU e tráfego. Este também irá recolher métricas relativas ao *throughput* nas *queues*, e também do *GiraGen*.

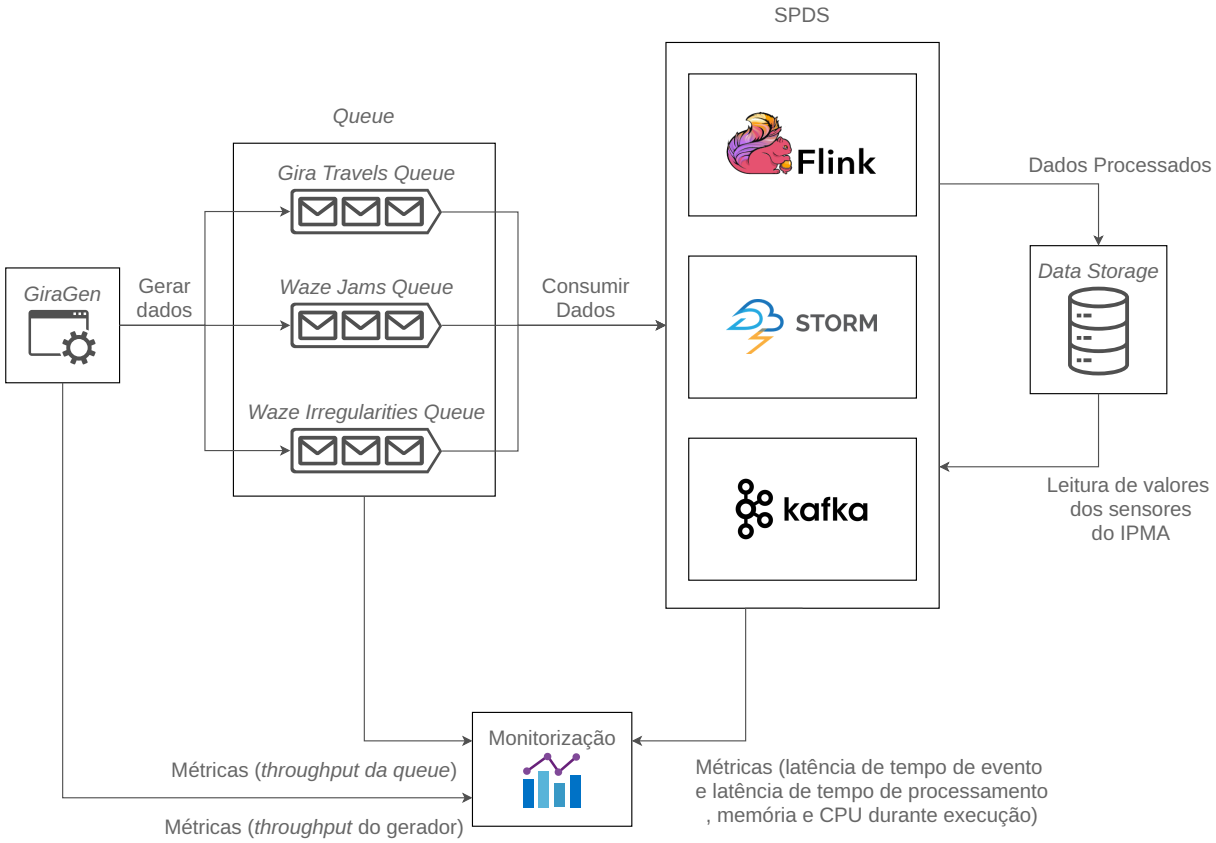


Figura 3.1: Arquitectura proposta

O *GiraGen* é uma aplicação de geração de dados em memória que necessita dos dados relativos ao *Gira Travels*, *Waze Jams* e *Waze Irregularities*, no formato *Comma-separated*

Values (CSV), para servir como fonte dos dados a realizar a geração. Este permite configurar o *throughput* do gerador, apesar de que o limite máximo do *throughput* estará limitado ao poder computacional onde é executado.

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 *bottleneck* nesta 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, ou seja, evitar a situação de *bottleneck* na respectiva arquitectura. Apesar deste componente servir o propósito tanto de escrita dos dados processados como de leitura de valores, é possível separar este componente em dois da seguinte forma:

1. Um componente para escrita dos dados processados com base em fila de mensagens ou *message broker*, ou um sistema de armazenamento de dados, relacional ou não relacional;

2. Um componente para leitura de valores com base num sistema de armazenamento de dados, relacional ou não relacional;

Tanto a componentes de *Queue* como a componente de *Data Storage* implicam que a topologia para cada SPDS seja adaptada para a(s) biblioteca(s) correspondentes caso estes dois componentes sejam substituidos por outros componentes que cumpram o mesmo fim.

Na Monitorização, o *Grafana* é responsável por apresentar painéis de instrumentação com base nos valores de métricas armazenadas no *InfluxDB*. O agente *Telegraf* é responsável por captar métricas, tais como o uso de memória, CPU, latências de tempo de evento e de processamento, *throughput* e tráfego. Ao captar estas métricas, as mesmas são enviadas e armazenadas no *InfluxDB*. Este agente é executado nas mesmas máquinas onde o *GiraGen* e a *Queue* são executados para captar o *throughput*. Para captar as latências, CPU, memória e tráfego são executados nas mesmas máquinas onde os componentes de processamento dos dados de cada SPDS são instalados. Em relação aos protocolos de comunicação para o agente, o *throughput* no *GiraGen* e latências é usado *StatsD*, enquanto que para o *throughput* da *Queue* é usado *RabbitMQ Management HTTP API* [41]. Para o CPU é usado *Docker Engine API* [42]. No uso de memória é utilizado *JMX*, visto que os SPDS são executados em *Java Virtual Machine* (*JVM*). Por último, o protocolo de comunicação entre o agente e o *InfluxDB* é realizado por *InfluxDB*

Line Protocol [43], enquanto que o protocolo entre o *Grafana* e o *InfluxDB* é realizado por *Influx Query Language* (*InfluxQL*) [44].

3.3 Topologia *Gira Travels Pattern*

3.3.1 Definição dos dados

Os dados usados na construção da topologia *Gira Travels Pattern* surgiram no âmbito do desafio "Existem padrões de utilização das bicicletas partilhadas em Lisboa ?"[45].

São disponibilizados dados acerca das viagens nas bicicletas Gira (*Gira Travels*), juntamente com dados de trânsito e irregularidades fornecidas pelo *Waze* (*Waze Jams* e *Waze Irregularities*), bem como valores dos vários sensores das estações meteorológicas do Instituto Português do Mar e da Atmosfera (IPMA), como por exemplo, temperatura média do ar, humidade média do ar, intensidade e direcção média do vento, média da radiação do sol e precipitação total.

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, 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.

A Figura 3.2 apresenta a estrutura dos dados usados no desenvolvimento da topologia. A estrutura dos dados do IPMA é exactamente a mesma para todos os valores dos vários sensores da estação meteorológica. Nos casos do *Gira Travels*, *Waze Jams* e *Waze Irregularites*, o valor "geom"representa uma geometria no formato *Well-Known Binary* (WKB) [46]. O WKB é um formato para representar geometrias, tais como pontos, linhas e polígonos, definidos pela *Open Geospatial Consortium* (OGC). Nos dados do Gira e do *Waze*, as geometrias presentes são do tipo *MultiLineString*.

3.3.2 *Topologia*

Com o objectivo de avaliar os diferentes SPDS, foi definida uma topologia que tem em conta os dados apresentados anteriormente. O resultado que se pretende obter durante

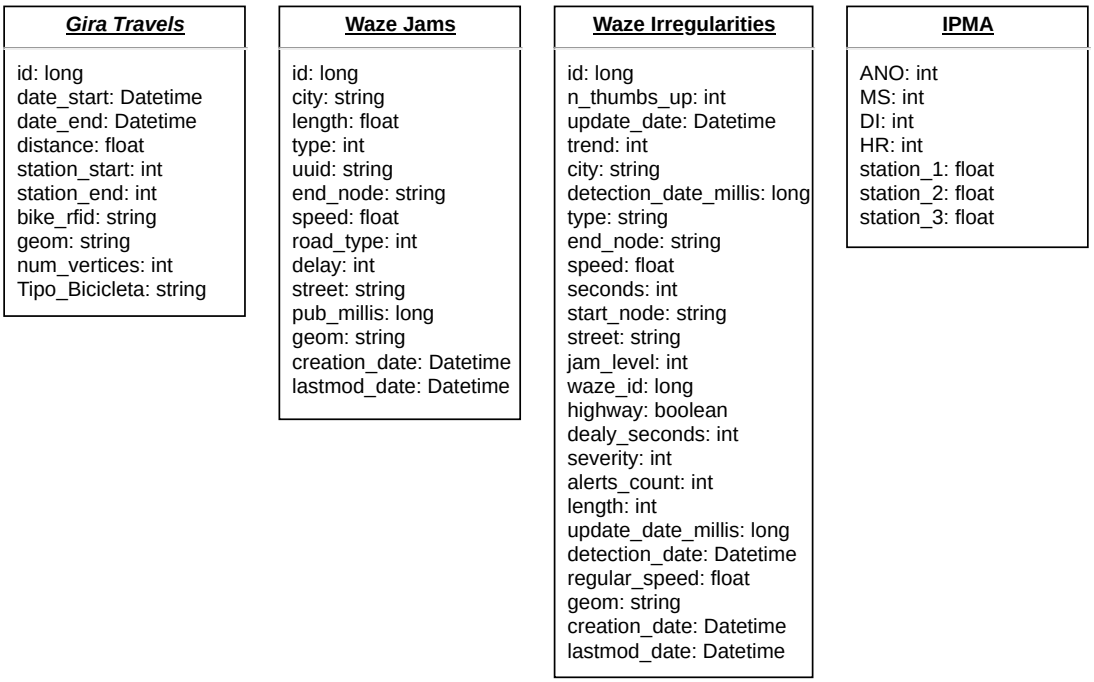


Figura 3.2: Esquema dos Dados

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.

Durante a análise dos dados fornecidos, detectou-se que não existe uma forma directa de poder correlacionar estes dados entre si, ou seja, não existe um identificador único entre eventos de viagens nas bicicletas Gira e os eventos do *Waze*, ou até mesmo com os valores dos sensores do IPMA. Por esta razão, optou-se por seguir outras abordagens de modo a correlacionar estes dados com base nas geometrias e no tempo de evento dos dados.

O uso de geometrias para correlação entre os dados do Gira e do *Waze* é especialmente interessante,uma vez que nos permite perceber se um certo evento ocorreu por estar "perto"de outro evento qualquer, ou seja, se o ponto de início de uma viagem nas bicicletas Gira intersectar com uma zona onde ocorreu um evento do *Waze*, então estes podem estar correlacionados. O uso do tempo de evento dos eventos do Gira e do *Waze* também permite que haja uma correlação temporal entre eles, isto é, se entre uma viagem nas bicicletas Gira e um evento do *Waze* aconteceram no mesmo minuto, então isto significa que estes eventos também podem estar correlacionados. Por último, de forma a enriquecer os dados para a produção final dos resultados deste processamento, é utilizado o tempo de evento do Gira de forma a adquirir os valores dos vários sensores do IPMA. A Figura 3.3 apresenta um esquema da topologia a ser implementada e

executada em cada SPDS.

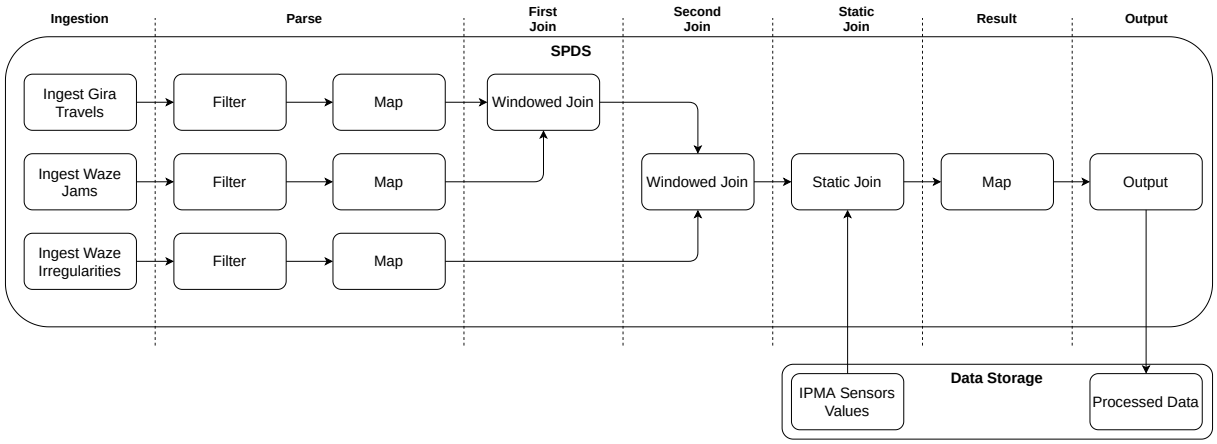


Figura 3.3: Topologia *Gira Travels Pattern*

A topologia está divida em sete fases funcionais, sendo estas *Ingestion*, *Parse*, *First Join*, *Second Join*, *Static Join*, *Result* e *Output*, caracterizado da seguinte forma:

1. O *Ingestion* é responsável por consumir os eventos das viagens Gira e do *Waze*, proveniente da fila de mensagens, no formato JSON;

2. O *Parse* filtra os eventos que não são legíveis para processamento e transforma estes eventos numa representação mais simplificada, contendo o identificador único, a geometria e o tempo de evento. A filtragem dos eventos é feita da seguinte forma: para os eventos das viagens Gira a condição é ter uma geometria associada, a distância estar presente e ser maior que zero e o número de vértices estar presente e ser maior que um; para os eventos do *Waze* a condição é ter uma geometria associada;

3. O *First Join* junta os eventos simplificados das viagens Gira com os eventos de trânsito do *Waze*, ao qual a chave de correlação é o tempo de evento, arredondado ao segundo, e que produz um tuplo com a combinação dos eventos das viagens Gira e trânsito do *Waze*;

4. O *Second Join* junta os eventos unidos anteriormente com os eventos de irregularidades do *Waze*, ao qual a chave de correlação é o tempo de evento, arredondado ao segundo, e que produz um tuplo com a combinação dos eventos das viagens Gira, trânsito e irregularidades do *Waze*;

5. O *Static Join* vai interrogar ao *data storage* os valores dos sensores do IPMA, com base numa chave composta por mês, dia e hora, a partir do tempo de evento das

viagens Gira, produzindo um tuplo com a combinação dos eventos das viagens Gira, trânsito e irregularidades do *Waze* e os valores dos sensores do IPMA. Como esta fase exige alguma carga para com este sistema externo e estes valores só se alteram hora a hora, é realizada a *cache* dos valores localmente ao componente que processa os dados;

6. O *Result* vai transformar o tuplo anterior num único dado em que contém a informação se o evento do trânsito e da irregularidade do *Waze* estão relacionados, com base na geometria. Além disso contém a informação se o evento da viagem Gira está relacionado com o evento do trânsito e irregularidade do *Waze*. Por fim, contém também toda a informação vinda do tuplo anterior.

7. O *Output* é responsável por enviar os dados processados na fase de *Result*, para o *data storage* no formato JSON;

3.4 Resumo

A arquitectura proposta para este trabalho segue uma arquitectura em que grande parte dos componentes envolventes no processamento de dados em *stream* são distribuídos e em que os elementos para análise quantitativa a serem aplicadas são as mais indicadas para a obtenção de resultados mais precisos dado o tipo de topologia em questão. Para além do processamento de dados, também é importante haver uma monitorização destes sistemas para a criação de alarmísticas para haver a percepção do comportamento dos mesmos. A separação por fases funcionais da topologia permitem pormenorizar os passos que são realizados durante a sua execução do processamento dos dados.

4

Desenvolvimento

Neste capítulo pretende-se apresentar todos os desenvolvimentos realizados neste trabalho de forma a enquadrar as várias tecnologias usadas e demonstrar como foi desenvolvida a topologia *Gira Travels Pattern* nos SPDS escolhidos.

4.1 Infraestrutura

4.1.1 Requisitos dos SPDS em *cluster*

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.

4.1.2 Implementação

Com base na arquitectura apresentada na Secção 3.2 e nos requisitos dos SPDS na Sub-Secção 4.1.1, as figuras seguintes representam a estruturação dos vários componentes e respectivas interacções. No anexo A.1 é apresentada a figura geral com todos os componentes. Toda a infraestrutura necessária para este trabalho será alojada em *clouds* públicas, nomeadamente na *Amazon Web Services* (AWS), recorrendo principalmente máquinas virtuais *Elastic Compute Cloud* (EC2) e a execução dos componentes realizado em *containers Docker*.

Dado que os componentes são executados em *containers Docker*, as imagens disponíveis de cada SPDS têm um desfasamento em relação à versão do *Java*, nomeadamente o *Apache Flink* tem disponível uma imagem com a versão *Java 11*, enquanto que o *Apache Storm* e *Apache Kafka* só têm disponível imagens com versão *Java 8*. De forma a estandardizar as imagens de todos os SPDS deste trabalho, foram desenvolvidas novas imagens *Docker* para o *Apache Storm* e *Apache Kafka* para *Java 11*. Para além do motivo de estandardizar as versões do *Java*, outra razões que impulsionaram este desenvolvimentos foram:

1. A versão da linguagem de programação *Java* a usar durante o desenvolvimento da topologia *Gira Travels Pattern* nos SPDS ser exactamente a mesma;

2. O uso padrão do *garbage collector G1GC* [47][48] durante a execução da topologia nos SPDS, dado os benefícios que este pode trazer em relação ao *ParallelGC* [49] durante a execução da topologia;

4.1.2.1 *GQSM*

O *GQSM* é um grupo constituído pelo *GiraGen*, *RabbitMQ* e *Redis*, que é responsável por gerar os dados para a execução do processamento da topologia *Gira Travels Pattern* e enviá-los para a fila de mensagens. Esta componente do sistema contém os dados do IPMA armazenados no *Redis* para pesquisa e também para armazenar os dados processados pelos SPDS. Tudo isto instalado numa única instância. Em resumo, este componente produz e recebe dados dos SPDS e produz valores de métricas de monitorização, nomeadamente *throughput*.

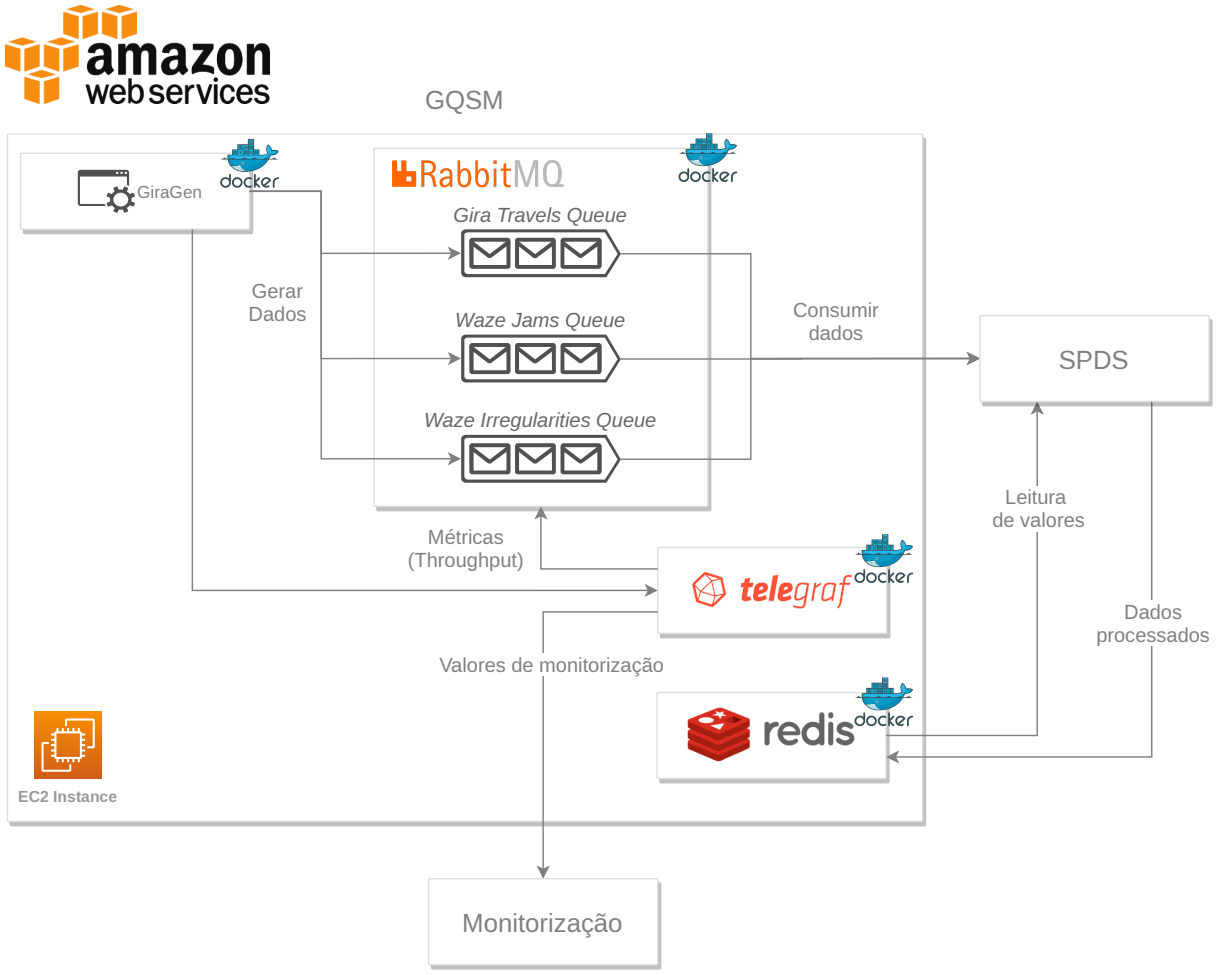


Figura 4.1: Infraestrutura GQSM

4.1.2.2 *Monitorização*

A *Monitorização* é instalada numa única instância, composta pelos *InfluxDB* e *Grafana*, com o objectivo de armazenar e visualizar as métricas enviadas pelos agentes *Telegraf*.

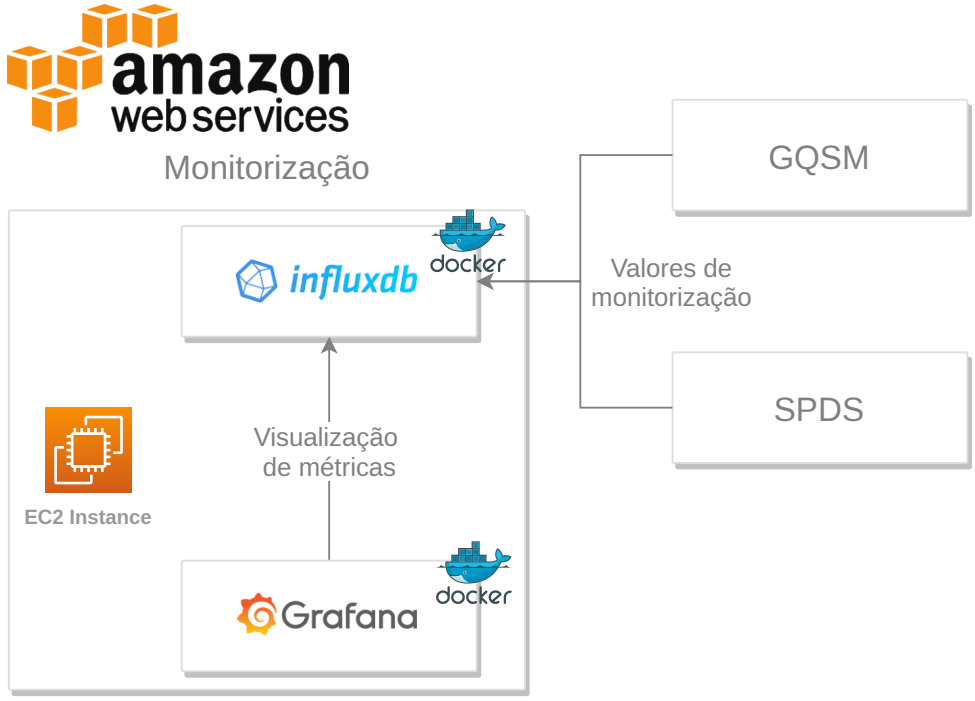


Figura 4.2: Infraestrutura Monitorização

4.1.2.3 *Apache Storm*

Para o *Apache Storm*, o *Nimbus* é instalado numa única instância juntamente com o *Apache Zookeeper* e o *Supervisor* será instalado numa ou mais instâncias. Para a execução das topologias, estes são submetidos no *Nimbus* que depois encarrega os *Supervisor*s de executar os vários operadores conforme a topologia desenvolvida. Durante a execução da topologia as métricas de latência de tempo de evento e de processamento, uso de memória e de CPU são captados pelo agente *Telegraf*, que por sua vez envia as métricas para o *InfluxDB*.

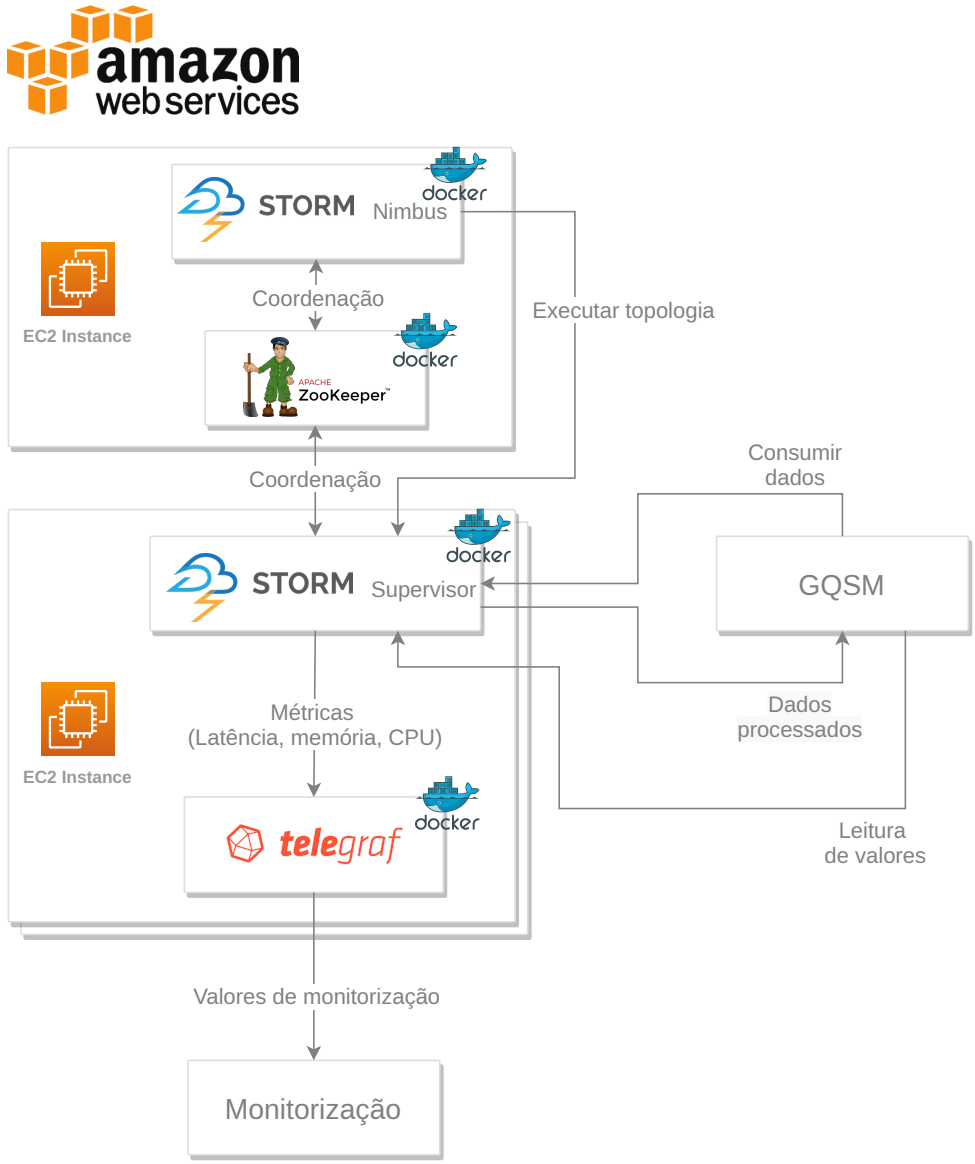


Figura 4.3: Infraestrutura para *Apache Storm*

4.1.2.4 *Apache Flink*

Para o *Apache Flink*, o *Job Manager* é instalado numa única instância e o *Task Manager* será instalado em uma ou mais instâncias. Para a execução das topologias, estes são submetidos no *Job Manager* que depois encarrega os *Task Manager*s de executar os vários operadores conforme a topologia desenvolvida. Durante a execução da topologia, o processo de captação das métricas é igual ao que foi apresentado na Secção anterior.

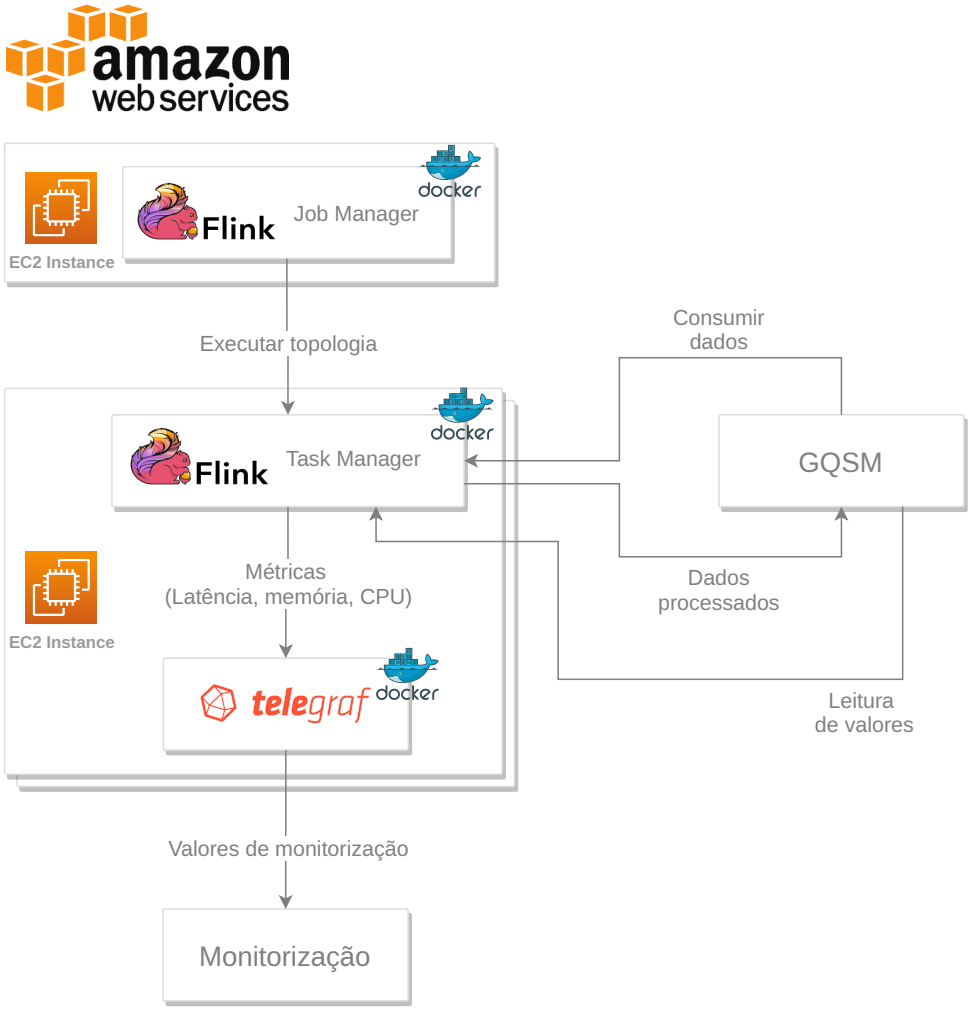


Figura 4.4: Infraestrutura para *Apache Flink*

4.1.2.5 *Apache Kafka*

Para o *Apache Kafka*, o único nó de *Kafka* é instalado numa única instância, juntamente com várias aplicações de *Kafka Connect*, para consumir os dados do *RabbitMQ* e colocá-los nos tópicos correspondentes, juntamente com o *Apache Zookeeper*. O *Kafka Stream Application* é instalado em uma ou mais instâncias. Para a execução das topologias, é a própria aplicação que usa a API de *Kafka Streams* que trata de executar o processamento dos dados, consumindo-os de tópicos do *Kafka*. Durante a execução da topologia, o processo de captação das métricas é igual ao que foi apresentado nas secções anteriores, à excepção de que o nó de *Kafka* são instrumentados métricas de uso de memória e CPU porque operadores com estado, por exemplo *joins*, necessitam de guardar o seu estado em tópicos intermédios no *Kafka* para depois serem consumidos num próximo operador.

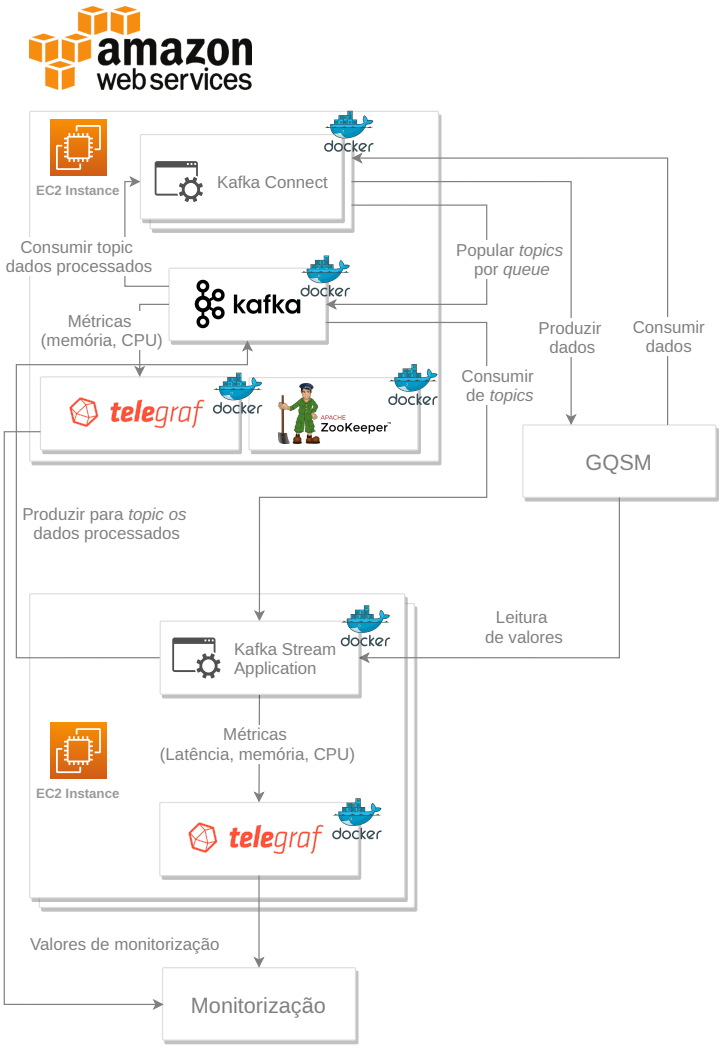


Figura 4.5: Infraestrutura para *Apache Kafka*

4.1.3 Instalação

Para simplificar a instalação e a replicabilidade da solução, os vários componentes correm em *containers Docker*, cujo início da execução é coordenada pela ferramenta *Docker Compose*. Para automatizar a instalação destes componentes, utilizaram-se duas ferramentas para gestão das instâncias EC2 e outra para automatizar a instalação, sendo estas o *Terraform* e o *Ansible*.

Relativamente à gestão desta infraestrutura, o *Terraform* foi configurado para usar o *provider* AWS, com ênfase na criação de máquinas virtuais EC2. Para tal, criou-se um módulo presente na Listagem A.2.1, que permite simplificar a caracterização de cada instância, incluindo configurações tais como o número de instâncias de um dado *namespace* e tipo de instâncias. Para definir a infraestrutura na AWS com o *Terraform* é necessário criar os *namespaces* para cada componente juntamente com o uso do módulo anteriormente definido e parametrizar. Em anexo, a Sub-Secção A.2.1 apresenta o módulo de criação das máquinas virtuais, enquanto que as Sub-Secções A.2.2, A.2.3 e A.2.4 apresentam as parametrizações para criar a infraestrutura para o *Apache Storm*, *Apache Flink* e *Apache Kafka*, de acordo com os diagramas apresentados nas Figuras 4.1, 4.2, 4.3, 4.4 e 4.5.

Para automatizar a instalação dos componentes nas máquinas virtuais EC2, no *Ansible* foram criados *roles* que definem os processos de instalação de cada componente, e foram criados *playbooks* para execução destes processos nas máquinas remotamente. Como os componentes são baseados em *containers Docker*, foi criado um *role* base que permite definir a criação de *containers* a instalar numa dada máquina, criando os ficheiros de *docker-compose* para depois serem executados. De seguida foram criados outros *roles*, estendendo do *role* anterior, para definir os serviços em si, como por exemplo para o caso do *Apache Flink* a criação do *Job Manager* e dos *Task Managers*. Por último foram criados os *playbooks* onde é definida a máquina onde executar o respectivo *role* e as variáveis de configurações. Em anexo, a Sub-Secção A.3.1, apresenta com maior detalhe a definição do *role* para criação de *docker-compose* e respectivo processo. As Sub-Secções A.3.2, A.3.3 e A.3.4 demonstram o uso dos *roles* anterior para definir o *docker-compose* para execução do vários componentes de cada SPDS enquanto que nas Sub-Secções A.3.5, A.3.6 e A.3.7 demonstram o uso dos *playbooks* para instalação dos componentes de cada SPDS nas máquinas virtuais EC2.

Estas ferramentas para além de servirem como suporte à instalação dos *clusters*, também vão permitir escalar verticalmente e horizontalmente, as instâncias que estiverem a ser usadas, havendo assim a possibilidade de obter resultados que possivelmente serão diferentes para configurações diferentes.

4.2 Implementação da topologia

Dado a definição da topologia na Sub-Secção 3.3.2, a sua implementação em cada SPDS deste trabalho, tem dois objectivos:

1. Transpor a lógica de processamento dos dados similar em cada SPDS para que seja possível a sua comparação, nomeadamente sobre os custos de desenvolvimento e a análise das latências do tempo de evento e de processamento;

2. Uniformalizar a extracção das latências do tempo de evento e de processamento de uma topologia.

Relativamente à uniformalização na extracção das latências do tempo de evento e de processamento de uma topologia, durante os desenvolvimentos das topologias, foi desenvolvida uma classe que permite abordar o conceito apresentado por um estudo [10] e obter estes valores da forma mais correcta. Esta classe, designada de *Observable*, permite encapsular um evento com três instantes: o instante da ocorrência do evento, o instante de quanto iniciou o processamento do evento e o instante de quando terminou o processamento. A Figura 4.6 apresenta a estrutura da classe *Observable*.

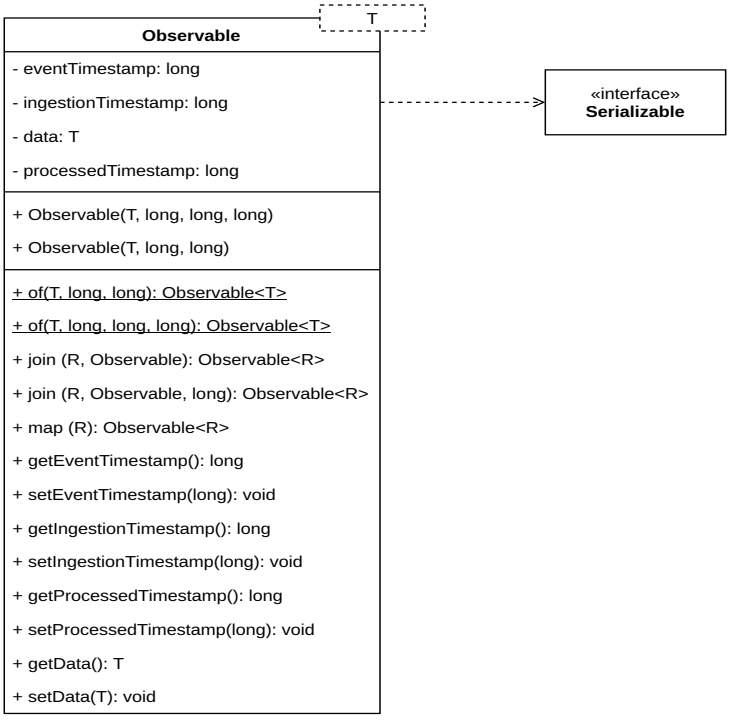


Figura 4.6: Classe *Observable*

As topologias foram desenvolvidos tendo em conta a separação por fases funcionais por uma questão de simplificação no desenvolvimento da topologia e para poder comparar mais elaboradamente o esforço e a simplicidade que cada um destes SPDS fornecem com as suas API.

Durante o desenvolvimento das topologias foram implementadas duas classes, independentes das API usadas em cada SPDS. Uma classe que permitem fazer *cache* dos valores obtidos na invocação de comandos de leitura no *Redis*, por exemplo, *HGET*. Outra classe que permite invocar comandos de escrita no *Redis*, tais como *LPUSH* ou *RPUSH*. Estas classes, denominadas *RedisHashCacheableMapFunction* e *RedisWriterFunction* respectivamente, são usadas em classes mais especificas para adaptar às API de cada SPDS de forma a não replicar exactamente o mesmo código que realiza a interacção com o *Redis*.

4.2.1 *Apache Storm*

No *Apache Storm* foi utilizada a API base (*Spouts & Bolts*) [32] para implementação da topologia do *Gira Travels Pattern*. Este disponibiliza uma classe, *TopologyBuilder*, que permite criar os *Spouts* e *Bolts*, inicializado da seguinte forma:

Listagem 4.1: *Apache Storm - Spouts & Bolts API* - Inicialização do *TopologyBuilder*

```
TopologyBuilder topologyBuilder = new TopologyBuilder();
```

Na fase de *Ingestion* são inicializados três *Spouts* que consomem os eventos das *queues* do *RabbitMQ*, desserializam os eventos no formato JSON e encapsulam os evento na estrutura *Observable*. A Listagem 4.2 demonstra como é inicializado o *Spout* para a *queue* dos eventos das viagens Gira.

Listagem 4.2: *Apache Storm - Spouts & Bolts API* - Fase de *Ingestion*

```
topologyBuilder.setSpout(GIRA_TRAVELS_SOURCE,
    RMQSpout.newRabbitMQSpout([...]))
```

Na fase de *Parse* são inicializados três *Bolts* que filtram os eventos e transformam estes eventos numa representação mais simples. No *Storm*, para poder definir uma *stream* de processamento de dados é necessário definir com qual nó de processamento é feito a ligação, e qual a estratégia de distribuição dos eventos. Na maioria dos casos queremos que os eventos sejam distribuídos uniformemente pelas *tasks* resultantes da criação do *Bolt* para maximização do paralelismo. Como apresentado na Secção 2.5 sobre a API *Spouts & Bolts*, este não permite uma distribuição uniforme, por exemplo *round-robin*, sendo que o tipo de distribuição usada foi uma distribuição aleatória através do

método *shuffleGrouping*. A Listagem 4.3 demonstra como é inicializado o *Bolt* que filtra e transforma os eventos das viagens Gira.

Listagem 4.3: *Apache Storm - Spouts & Bolts API* - Fase de *Parse*

```
topologyBuilder.setBolt(SIMPLIFIED_GIRA_TRAVELS_STREAM,
    ParserBolt.parse([...]))
    .shuffleGrouping(giraTravelsSource);
```

Nas fases de *First Join* e *Second Join* são efectuados, primeiro, a junção das *streams* geradas na fase de *Parse* dos eventos das viagens Gira e eventos do trânsito do *Waze*, e segundo, a junção da *stream* gerada na junção anterior com os eventos das irregularidades do *Waze*. Ao contrário da fase anterior, as operações de *join* necessitam que a distribuição dos eventos sejam particionados, por um ou mais campos como definição de chave única, método *fieldsGrouping*. Como os operadores de *join* usam o processo de *windowing*, o tipo de janela escolhido foi *Sliding Window* com o uso do método *withWindow*, usando os argumentos de tamanho da janela e intervalo. Para garantir que a topologia usa o tempo de evento como característica temporal, é definido o campo que representa o respectivo tempo, método *withTimestampField*. A Listagem 4.4 demonstra como inicializar o *Bolt* para realizar a fase de *First Join*, sendo que a inicialização para a fase de *Second Join* é semelhante.

Listagem 4.4: *Apache Storm - Spouts & Bolts API* - Fase de *First Join*

```
topologyBuilder.setBolt(JOINED_GIRA_TRAVELS_WITH_WAZE_JAMS_STREAM,
   joinedGiraTravelsWithWazeJamsBolt
    .withWindow([...], [...])
    .withTimestampField("event_timestamp")
    .withWatermarkInterval([...])), [...])
    .fieldsGrouping(simplifiedGiraTravelsStream, new Fields("key"))
    .fieldsGrouping(simplifiedWazeJamsStream, new Fields("key"));
```

Na fase de *Static Join* é inicializado um *Bolt* que comunica com o *Redis* para adquirir os valores dos sensores do IPMA e faz *cache* desses mesmos valores para não sobrecarregar o *Redis* com múltiplos pedidos ao qual o resultado só difere de hora em hora. A Listagem 4.5 demonstra a inicialização da função que realiza os pedidos ao *Redis* e realização de *cache*, e criação do respectivo *Bolt*.

Listagem 4.5: *Apache Storm - Spouts & Bolts API* - Fase de *Static Join*

```
RedisHashCacheableMapFunction<[...]> mapFunction =
    RedisHashCacheableMapFunction.newMapperMonitored([...]);

IpmaValuesCacheableMapBolt bolt = new
```

```
    IpmaValuesCacheableMapBolt(mapFunction, [...]);

topologyBuilder.setBolt(JOINED_GIRA_TRAVELS_WITH_WAZE_AND_IPMA_STREAM, bolt
    , [...])
    .shuffleGrouping(joinedGiraTravelsWithWazeStream);
```

Por último, as fases de *Result* e *Output* são criados mais dois *Bolts*, um para transformar o conjunto dos eventos agrupados nas fases anteriores no resultado final, e outro para escrever este resultado para o *Redis*.

Listagem 4.6: *Apache Storm - Spouts & Bolts API* - Fase de *Result*

```
topologyBuilder.setBolt(RESULT_STREAM
    , new ResultMapBolt(geoFactory), [...])
    .shuffleGrouping(joinedGiraTravelsWithWazeAndIpmaStream);
```

Listagem 4.7: *Apache Storm - Spouts & Bolts API* - Fase de *Output*

```
RedisWriterFunction<[...]]> redisSinkFunction = RedisWriterFunction
    .newWriter([...]);

ResultSinkBolt bolt = new ResultSinkBolt([...]
    , redisSinkFunction);

topologyBuilder.setBolt("output", ObservableBolt.observe([...], bolt,
    [...])
    .shuffleGrouping(resultStream);
```

4.2.2 *Apache Flink*

No *Apache Flink* foi usado a *DataStream API* [35] para implementação da topologia do *Gira Travels Pattern*. Este disponibiliza uma classe, *StreamExecutionEnvironment*, que permite criar o fluxo de dados, inicializado da seguinte forma:

Listagem 4.8: *Apache Flink - DataStream API* - Inicialização do *StreamExecutionEnvironment*

```
final StreamExecutionEnvironment env = StreamExecutionEnvironment.
    getExecutionEnvironment();
```

Na fase de *Ingestion* são adicionados três *sources*, usando o método *addSource*, que consomem os eventos das *queues* do *RabbitMQ*, desserializa os eventos no formato JSON e encapsula o evento na estrutura *Observable*. Depois, para cada evento é assignado um *timestamp* para que a execução da topologia esteja configurado para a característica

temporal com base em tempo de evento através do método *assignTimestampsAndWatermarks*. A Listagem 4.9 mostra como é inicializado a *source* para a *queue* dos eventos das viagens Gira.

Listagem 4.9: *Apache Flink - DataStream API* - Fase de *Ingestion*

```
streamExecutionEnvironment.addSource(DataStreamRMQSource
    .newRabbitMQSource([...])
    .assignTimestampsAndWatermarks([...])
    .name("gira_travels_source");
```

Na fase de *Parse* são definidas as sequências de operações, *filter* e *map*, sobre os três *sources* anteriormente criadas, que filtram os eventos e transformam estes eventos numa representação mais simples. O uso do método *rebalance* permite que seja aplicada o algoritmo de *round-roubin* para distribuição uniforme dos eventos pelas operações subsequentes. A Listagem 4.10 demonstra como é definido estas operações sobre os eventos das viagens Gira.

Listagem 4.10: *Apache Flink - DataStream API* - Fase de *Parse*

```
giraTravelsSource
    .rebalance()
    .filter([...])
    .map([...])
```

Nas fases de *First Join* e *Second Join* é feita uma junção através do método *join*, em que é definido como juntar as duas *streams* usando os métodos *where* e *equalTo*. Depois é definido o tipo de janela, método *window*. Para a junção foi usada uma janela deslizante, *SlidingEventTimeWindows*. Por último, a lógica de processamento da junção das duas *streams* é realizada no método *apply*. A Listagem 4.11 demonstra como juntar duas *streams* com um operador de *join*, e definido um certo tipo de janela. A aplicação deste operador na fase *Second Join* é semelhante.

Listagem 4.11: *Apache Flink - DataStream API* - Fase de *First Join*

```
simplifiedGiraTravelsStream
    .join(simplifiedWazeJamsStream)
    .where([...])
    .equalTo([...])
    .window(SlidingEventTimeWindows.of([...]))
    .apply([...]);
```

Na fase de *Static Join* é inicializada uma função que comunica com o *Redis* para adquirir os valores dos sensores do IPMA e faz *cache* desses mesmos valores, pelas mesmas

razões apresentadas na mesma fase presente na Sub-Secção 4.2.1. Ao adquirir os respectivos valores, estes são mapeados, com o uso do método *map*. O método *rebalance* é utilizado pela mesma razão apresentado na fase *Parse*, relativamente à distribuição uniforme dos eventos entre operações subsequentes, neste caso em concreto, entre o *intervalJoin* da fase *Second Join* e *map* desta fase. A Listagem 4.12 demonstra o uso do *Redis* para adquirir os valores do IPMA e mapeá-los.

Listagem 4.12: *Apache Flink - DataStream API* - Fase de *Static Join*

```
RedisHashCacheableMapFunction<S[...]> mapper =
    RedisHashCacheableMapFunction.newMapperMonitored([...]);

joinedGiraTravelsWithWazeStream
    .rebalance()
    .map(IpmaValuesCacheableMapFunction.function([...]));
```

Por fim, as fases de *Result* e *Output* são definidas. Primeiro, o método *map* para transformar o conjunto dos eventos agrupados nas fases anteriores no resultado final, e segundo, o método *addSink* para escrever este resultado para o *Redis*.

Listagem 4.13: *Apache Flink - DataStream API* - Fase de *Result*

```
enrichedJoinedGiraTravelsWithWazeAndIpma.map(new ResultMapFunction(
   geoFactory));
```

Listagem 4.14: *Apache Flink - DataStream API* - Fase de *Output*

```
RedisWriterFunction<[...]> redisSinkFunction = RedisWriterFunction
        .newWriter([...]);

resultStream.addSink(ObservableSinkFunction.observe([...]
    , ResultSink.sink(redisSinkFunction)));
```

4.2.3 *Apache Kafka*

No *Apache Kafka* utilizou-se a *Streams DSL* [36] para implementação da topologia do *Gira Travels Pattern*. Ao contrário dos SPDS anteriores, o *Kafka Streams* só consegue consumir e produzir os eventos com base em tópicos do *Kafka*, e como tal, são criados três conectores onde cada um consome de uma *queue* do *RabbitMQ* e produzem para os respectivos tópicos, *gira_travels*, *waze_jams* e *waze_irregularities*. Para criar o fluxo de dados, este disponibiliza uma classe, *StreamsBuilder*, que é inicializado da seguinte forma:

Listagem 4.15: *Kafka Streams - Streams DSL* - Inicialização do *StreamsBuilder*

```
final StreamsBuilder streamsBuilder = new StreamsBuilder();
```

Na fase de *Ingestion* são adicionados três *streams*, denominados *KStream*, que consomem os eventos dos tópicos do *Kafka*, desserializam os eventos no formato JSON e encapsulam o evento na estrutura *Observable*. A Listagem 4.16 demonstra como é inicializado a *stream* para o tópico *gira_travels* dos eventos das viagens Gira.

Listagem 4.16: *Kafka Streams - Streams DSL* - Fase de *Ingestion*

```
streamsBuilder
    .stream("gira_travels", [...])
    .mapValues([...]);
```

Na fase de *Parse* são definidas as sequências de operações, *filter* e *map*, sobre os três *streams* anteriormente criados, que filtram os eventos, transformam estes eventos numa representação mais simples e é gerada uma nova *KStream* em que é particionada os eventos ao usar o método *pair*. A Listagem 4.17 demonstra como são definidas estas operações sobre os eventos das viagens Gira.

Listagem 4.17: *Kafka Streams - Streams DSL* - Fase de *Parse*

```
giraTravelsSource
    .filter([...])
    .map((k, v) -> KeyValue.pair([...]));
```

Nas fases de *First Join* e *Second Join* define-se como é realizada a junção entre os eventos das viagens Gira com os eventos do trânsito do *Waze*, usando o método *join* e definindo a junção como *Sliding Window*, *JoinWindows*. De seguida, é realizada a junção entre os eventos das irregularidades do *Waze* e os eventos gerados pela junção anterior, usando o mesmo método e tipo de *window*. A Listagem 4.18 demonstra como juntar duas *streams* com um operador de *join*. A aplicação deste operador é semelhante para a fase *Second Join*.

Listagem 4.18: *Kafka Streams - Streams DSL* - Fase de *First Join*

```
simplifiedGiraTravelsStream
    .join(simplifiedWazeJamsStream
            , (left, right) -> [...]
            , JoinWindows.of([...])
            , StreamJoined.with([...]);
```

Na fase de *Static Join* é definido um mapeamento, *mapValues*, que realiza a comunicação com o *Redis* para adquirir os valores dos sensores do IPMA, presente na Listagem 4.19.

Listagem 4.19: *Kafka Streams - Streams DSL* - Fase de *Static Join*

```
joinedGiraTravelsWithWazeStream
        .mapValues([...]);
```

Por último, as fases de *Result* e *Output* são definidas, primeiro, o método *map* para transformar o conjunto dos eventos agrupados nas fases anteriores no resultado final, e segundo, o método *to* para escrever este resultado para o tópico *kafka_result*. Para que os resultados do processamento dos dados no *Kafka Streams* sejam escritos para o *Redis*, criou-se um conector para consumir os eventos do tópico do *Kafka* e escrever para no *Redis*.

Listagem 4.20: *Kafka Streams - Streams DSL* - Fase de *Result*

```
enrichedJoinedGiraTravelsWithWazeAndIpma
    .map((k, v) ->[...]);
```

Listagem 4.21: *Kafka Streams - Streams DSL* - Fase de *Output*

```
resultStream.to("kafka_result",[...]);
```

4.3 Resumo

Os desenvolvimentos realizados no âmbito deste trabalho trazem algumas ideias acerca da automatização dos processos de instalação dos componentes necessários para o correcto funcionamento de cada SPDS, e restantes componentes que complementam o envio dos eventos para estes sistemas e monitorização, mas também como automatizar o aprovisionamento de infraestrutura em *clouds* públicas. Implementar uma topologia com um caso de estudo em concreto permite demonstrar o uso das API que os SPDS disponibilizam, e também servir de base para obter resultados relevantes para o estudo de desempenho.

5

Execução e Resultados

Neste capítulo pretende-se demonstrar o processo de execução da topologia *Gira Travels Pattern* nos SPDS e apresentar o conjunto de parâmetros usado durante a execução do mesmo. Por último, pretende-se apresentar os resultados obtidos, sendo estes o máximo *throughput* sustentável, latência de tempo de evento e de processamento, uso de memória, de CPU e tráfego.

5.1 Execução da topologia *Gira Travels Pattern*

Para poder realizar a comparação de desempenho entre os vários SPDS escolhidos para este trabalho, é necessário que uma mesma topologia seja executada no SPDS. O processo de execução de uma topologia em cada SPDS é a seguinte:

1. Inicializar as instâncias EC2 e instalar os componentes obrigatórios de cada SPDS testar, com base no que foi apresentado nas Sub-Secções 4.3, 4.4 e 4.5
2. Inicializar a instância EC2 e instalar os componentes para *GQSM*, apresentado na Sub-Secção 4.1
3. Inicializar a instância EC2 e instalar os componentes para monitorização, apresentado na Sub-Secção 4.2
4. Migrar os dados dos valores dos sensores do IPMA para o *Redis*

5. 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*;

6. Instalar o *GiraGen*;

Para executar a topologia *Gira Travels Pattern* foi definido um conjunto de parâmetros para cada SPDS, presentes nas tabelas 5.1, 5.2 e 5.3. O tipo de instância EC2 usada durante a execução da topologia é do tipo *C5* [50]. Este tipo de instâncias beneficiam de CPUs de alto desempenho fazendo com que sejam instâncias optimizadas para computação. No total o sistema usa actualmente 126 vCPU e 252 GB de memória.

Tabela 5.1: Tabela de parametrização para *Apache Storm*
Apache Strom - versão 2.2.0

Parâmetro	Valor padrão	Valor definido
Sistema Operativo	-	Debian 10.5 (Buster)
Número de instâncias - *Nimbus*	-	1
Número de instâncias - *Supervisor*	-	1, 2 e 4
Tipo de instância - *Nimbus*	-	*c5a.large*
Tipo de instância - *Supervisor*	-	*c5a.2xlarge*
Nimbus CPUs	-	*2*
Supervisor CPUs	-	*8*
Nimbus memória / *heap*	-	8 GB / 6 GB
Supervisor memória / *heap*	-	16 GB / 10 GB
Garbage Collector	*ParallelGC*	*G1GC*
Paralelismo	1	8, 16 e 32
Número de *Workers*	-	8, 16 e 32
Característica temporal	tempo de processamento	tempo de evento
Fase *First Join - Sliding Window joins*	-	L: 5 ms; I: 5 ms (1)
Fase *Second Join - Sliding Window joins*	-	L: 30 ms; I: 30 ms (1)

(1) - Nos *joins* o L representa tamanho da janela(*Length*) e I representa o intervalo da janela(*Interval*);

Tabela 5.2: Tabela de parametrização para *Apache Flink*
Apache Flink - versão 1.11.1

Parâmetro	Valor padrão	Valor definido
Sistema Operativo	-	Debian 10.5 (Buster)
Número de instâncias - *Job Manager*	-	1
Número de instâncias - *Task Manager*	-	1, 2 e 4
Tipo de instância - *Job Manager*	-	c5a.large
Tipo de instância - *Task Manager*	-	c5a.2xlarge
Job Manager vCPUs	-	2
Task Manager vCPUs	-	8
Job Manager memória / *heap*	-	4 GB / 3 GB
Task Manager memória / *heap*	-	16 GB / 10 GB
Task Manager memória *off-heap*	-	4 GB
Garbage Collector	*ParallelGC*	*G1GC*
Task slots	1 por *Task Manager*	8 por *Task Manager*
Paralelismo	1	8, 16, 32
Característica temporal	tempo de processamento	tempo de evento
Object reuse	desactivado	activado (1)
Sliding Window Joins	-	L: -5 ms; L: 5 ms (2)

(1) - Esta configuração permite que objectos definidos na topologia possam ser reaproveitados, deixando assim de serem sempre inicializados e melhorando o desempenho;
(2) - Nos *joins* o L representa tamanho da janela(*Length*) e I representa o intervalo da janela(*Interval*).

Tabela 5.3: Tabela de parametrização para *Apache Kafka*
Apache Kafka - versão 2.7.0

Parâmetro	Valor padrão	Valor definido
Sistema Operativo	-	Debian 10.5 (Buster)
Número de instâncias - *Kafka*	-	1
Número de instâncias - *Kafka Streams*	-	1, 2 e 4
Tipo de instância - *Kafka*	-	*c5ad.2xlarge*
Tipo de instância - *Kafka Streams*	-	*c5a.2xlarge*
Tipo de disco - *Kafka*	-	SSD NVMe
Tamanho de disco - *Kafka*	-	100 GB
Kafka CPUs	-	*8*
Kafka Streams CPUs	-	*8*
Kafka memória / *heap*	-	16 GB / 16 GB
Kafka Streams memória / *heap*	-	16 GB / 10 GB
Garbage Collector	*ParallelGC*	*G1GC*
Característica temporal	tempo de processamento	tempo de evento
Sliding Window Joins	-	S: 5 ms; I: 5 ms (1)
Número de *threads* para execução do processamento em *stream*	-	8, 16, 32

(1) - Nos *joins* o L representa tamanho da janela(*Length*) e I representa o intervalo da janela(*Interval*);

5.2 Resultados

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. Os resultados obtidos foram:

- Máximo *throughput* sustentável;
- Latências de tempo de eventos e de processamento (média, 90 percentil e 99.9 percentil);
- Percentagem de utilização de CPU por nó;
- Percentagem de utilização de memória por nó;
- Tráfego;

Em relação à infraestrutura, esta era utilizada logo após terminar o aprovisionamento das instâncias para instalar e executar a topologia nos SPDS.

O máximo *throughput* sustentável obteve-se ao ajustar o *throughput* dos eventos que o *GiraGen* produzia para as *queue*s e monitorizando as latências de tempo de evento e de processamento. Caso as latências aumentassem constantemente, isto seria indicativo de que para as condições presentes durante a execução da topologia, estava a causar atrasos, logo seria necessário reajustar novamente o *throughput* do *GiraGen* até que as latências apresentassem valores estáveis. A Tabela 5.4 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.

Tabela 5.4: 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

5.2.1 *Apache Storm*

As próximas Sub-Secções irão apresentar os resultados relativamente às latências de tempo de evento e de processamento, uso de memória e de CPU e tráfego, obtidos durante a execução da topologia no *Apache Storm*, com um, dois e quatro nós de *Supervisors*.

1 *Supervisor*

Na primeira execução da topologia *Gira Travels Pattern*, com um *throughput* de 3000 eventos por segundo, na Figura 5.1 destaca-se o facto das latências de tempo de evento e de processamento serem semelhantes.

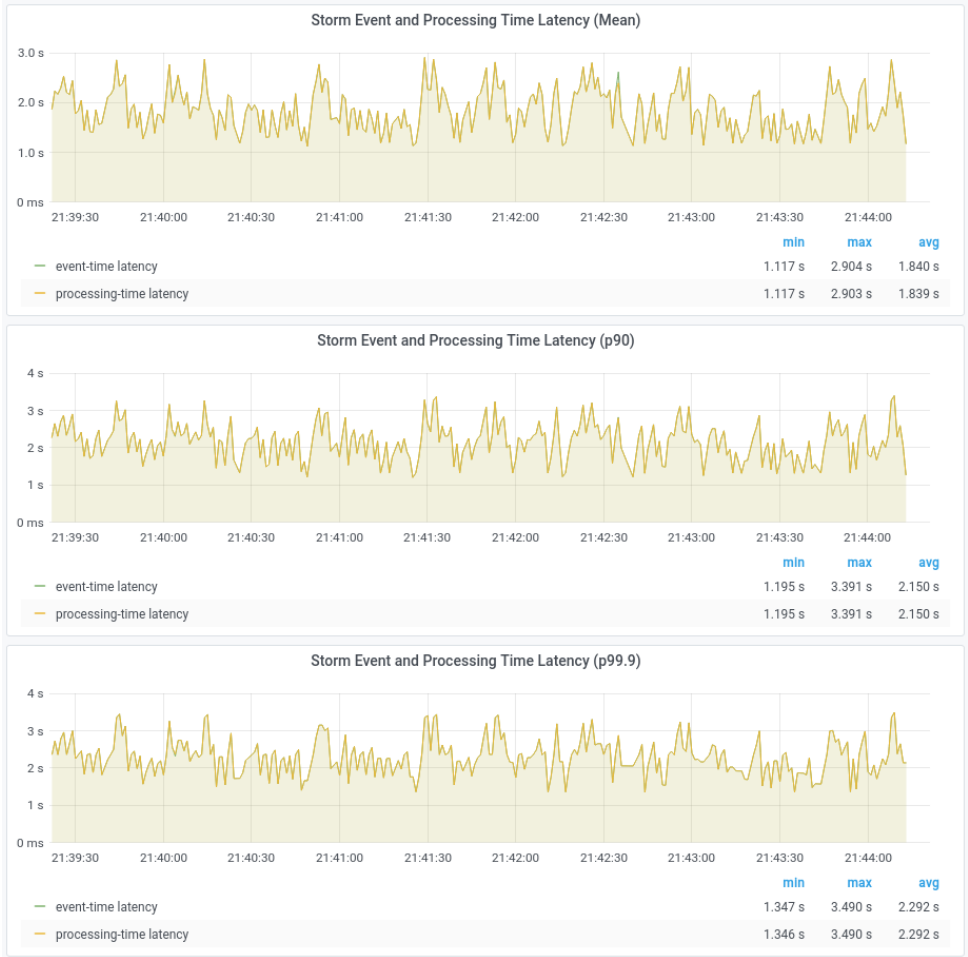


Figura 5.1: *Apache Storm* - 1 *Supervisor* - Latências

Dado a Figura 5.2, no uso de memória observa-se algumas oscilações, e com uma tendência de aumento dado um espaço de tempo de cerca de 5 minutos. No entanto, a

certa altura o *garbage collector* é executado, o que faz com que o valores sejam semelhantes ao minuto 39.

Figura 5.2: *Apache Storm* - 1 *Supervisor* - CPU e memória

Por fim, o tráfego detectado é somente da comunicação dos eventos da *queue* para a topologia, e a escrita dos resultados da topologia para o *Redis*, como indica a Figura B.1 em anexo.

2 *Supervisor*

Na segunda execução da topologia *Gira Travels Pattern*, com um *throughput* de 3500 eventos por segundo, na Figura 5.3 destaca-se semelhanças nas latências de tempo de evento e de processamento, sendo que, ao aumentar-mos o *throughput* e o número de nós de *Supervisor*, as latências também aumentaram.

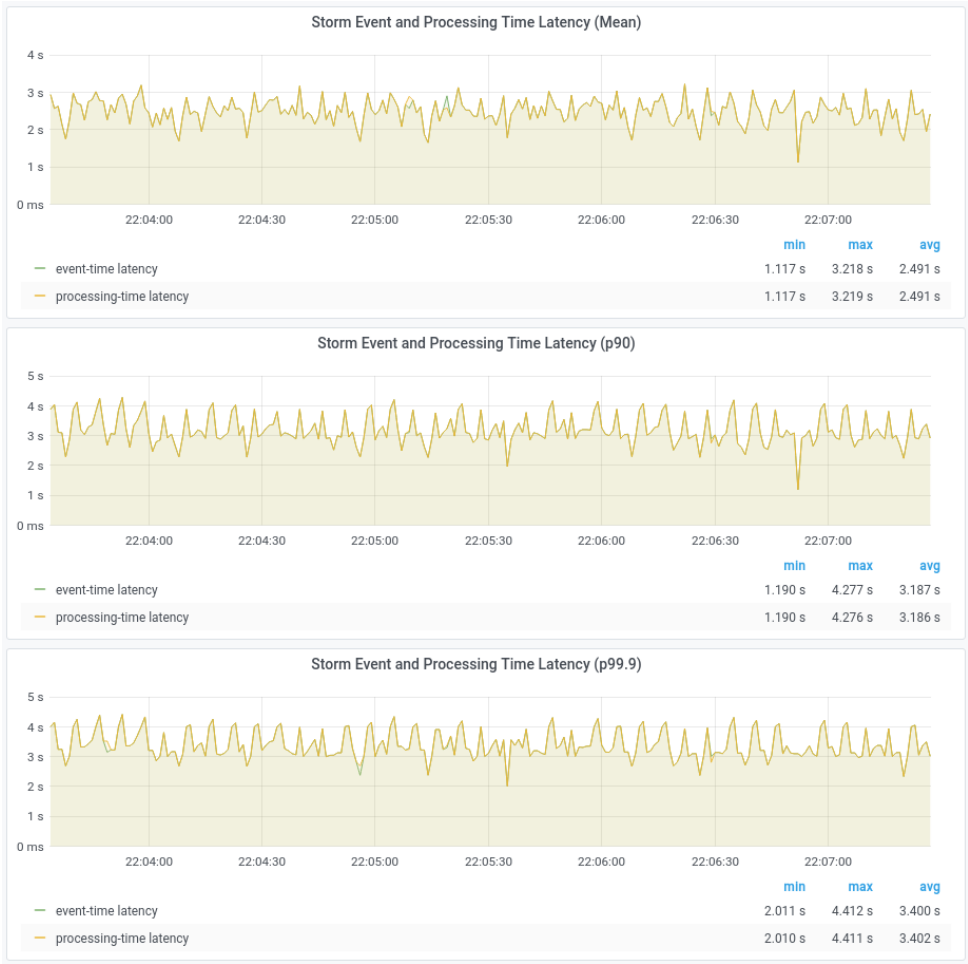
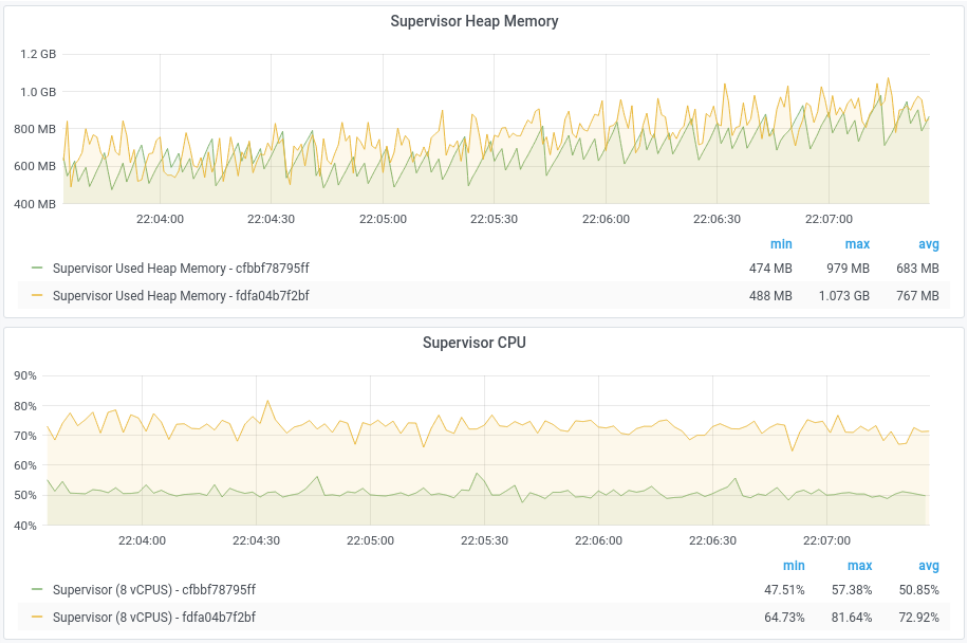


Figura 5.3: *Apache Storm* - 2 *Supervisor* - Latências

Relativamente ao uso de memória, em comparação aos resultados que foram apresentados na Figura 5.2, estes são semelhantes para ambos os nós de *Supervisor*. Ao contrário do uso de CPU ao qual se destaca na Figura 5.4 que um dos nós apresenta valores relativamente próximos ao anterior, enquanto que o outro nó só está com uma carga média de cerca de 50.85%.

Por fim, o tráfego entre os dois nós parecem ser proporcionais à carga que os nós estavam submetidos durante a execução da topologia, como indica a Figura B.2 em anexo. Um factor que pode levar a este acontecimento é o facto de durante a submissão da



Figura

Figura 5.4: *Apache Storm* - 2 *Supervisor* - CPU e memória

topologia é o *Nimbus* que decide sobre os *Supervisor* que operadores, *Spouts* e *Bolts*, devem executar, tendo em conta o paralelismo pretendido. Ou seja, o nó com mais carga pode estar a executar *Bolts* que tenham algum custo significativo na execução.

4 *Supervisor*

Na terceira execução da topologia *Gira Travels Pattern*, com um *throughput* de 4000 eventos por segundo, dados as Figuras 5.5, 5.6 e Figura B.3 em anexo, apresentam conclusões semelhantes às enunciadas na Sub-Secção 5.2.1 sobre as latências, uso de CPU, de memória e tráfego.

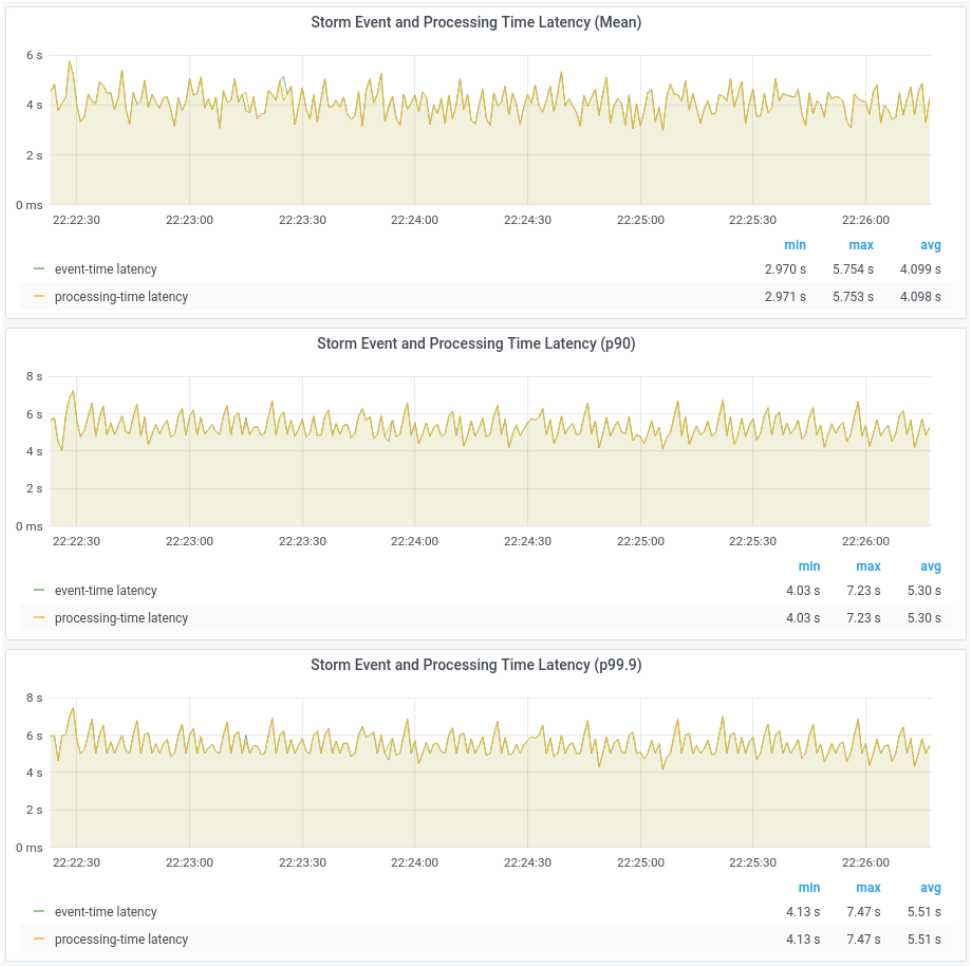


Figura 5.5: *Apache Storm* - 4 *Supervisor* - Latências

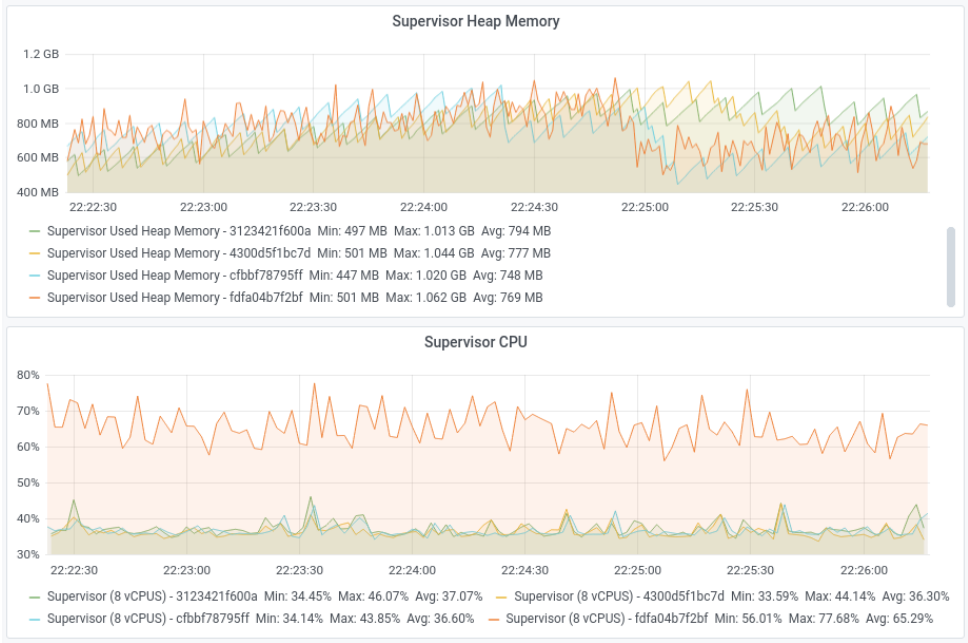


Figura 5.6: *Apache Storm* - 4 *Supervisor* - CPU e memória

5.2.2 *Apache Flink*

As próximas Sub-Secções irão apresentar os resultados relativamente às latências de tempo de evento e de processamento, uso de memória e de CPU e tráfego, obtidos durante a execução da topologia no *Apache Flink*, com um, dois e quatro nós de *Task Managers*.

1 *Task Manager*

Na primeira execução da topologia *Gira Travels Pattern*, com um *throughput* de 2850 eventos por segundo, na Figura 5.7 destaca-se o facto de as latências de tempo de evento e de tempo de processamento serem semelhantes.

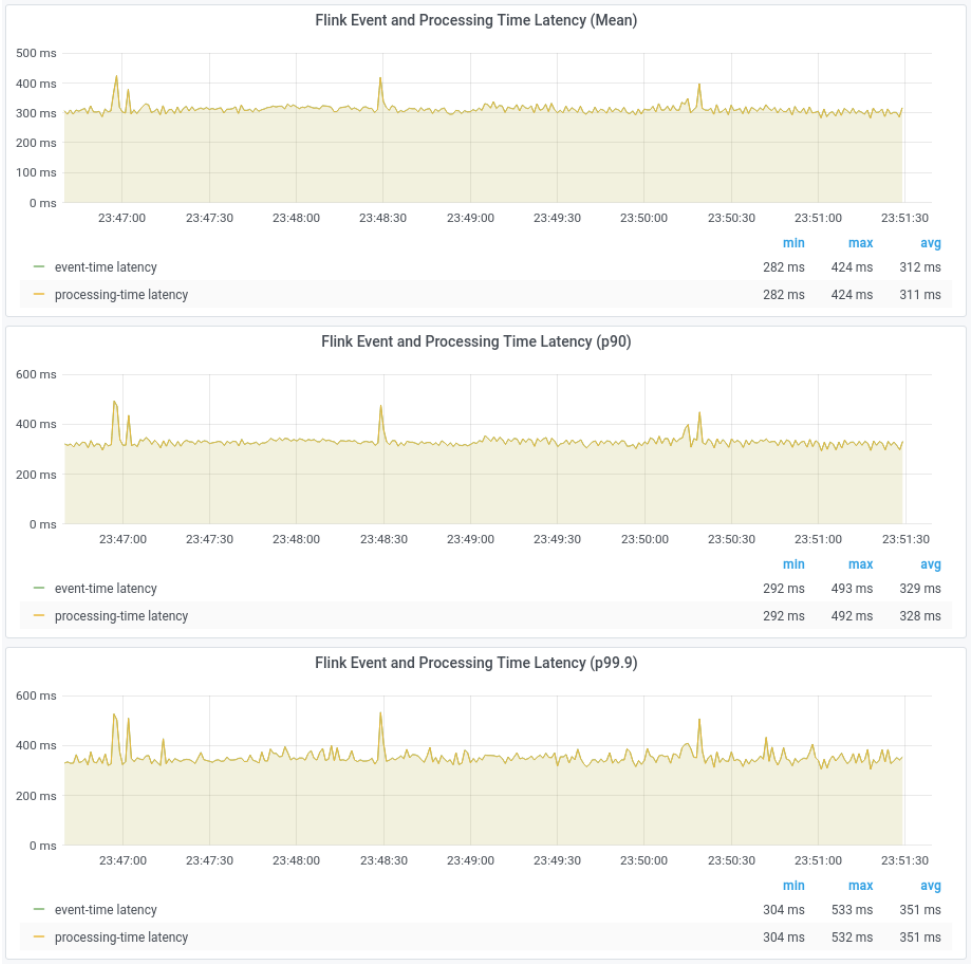


Figura 5.7: *Apache Flink* - 1 *Task Manager* - Latências

Dado a Figura 5.8, o uso de memória destaca-se algumas oscilações bastante acentuadas, rondando entre os 100 MB e 6 GB. A frequência destas oscilações é indicativo

que o *garbage collector* foi executado, e esta execução não teve qualquer impacto no desempenho. Em relação ao uso de CPU, é bastante estável, excepto nos momentos que houve uma redução súbita da carga.

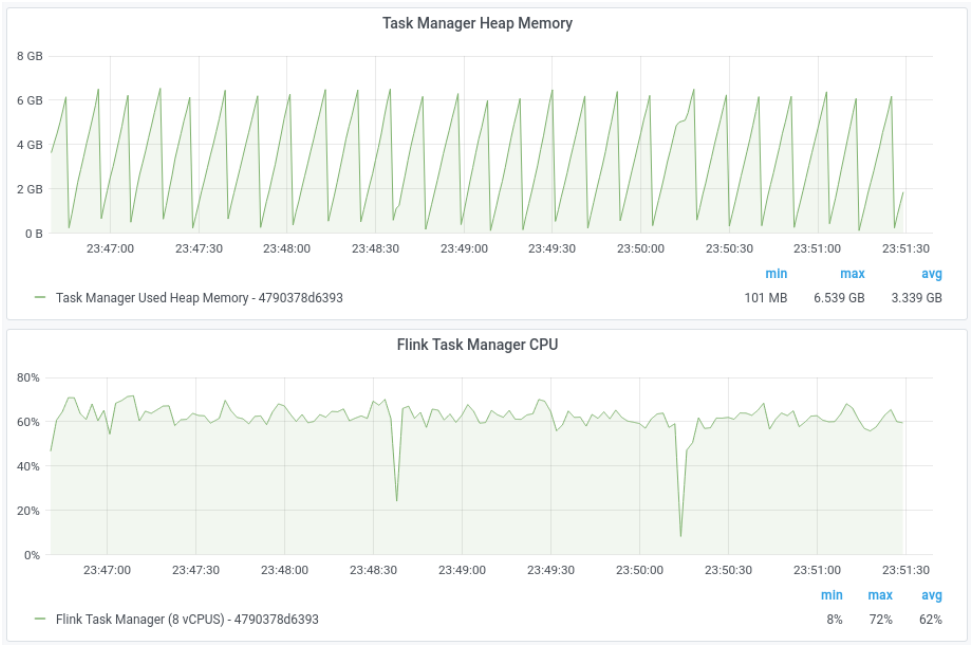


Figura 5.8: *Apache Flink* - 1 *Task Manager* - CPU e memória

Por fim, o tráfego em relação à transmissão de pacotes é bastante acentuado, isto porque, como as latências rondam os milissegundos, é indicativo que a topologia esteja a produzir bastantes valores de escrita para o *Redis*, como indica a Figura B.4 em anexo.

2 *Task Managers*

Na segunda execução da topologia *Gira Travels Pattern*, com um *throughput* de 3500 eventos por segundo, na Figura 5.9 destaca-se que as latências de tempo de evento e de tempo de processamento são semelhantes à execução anterior.

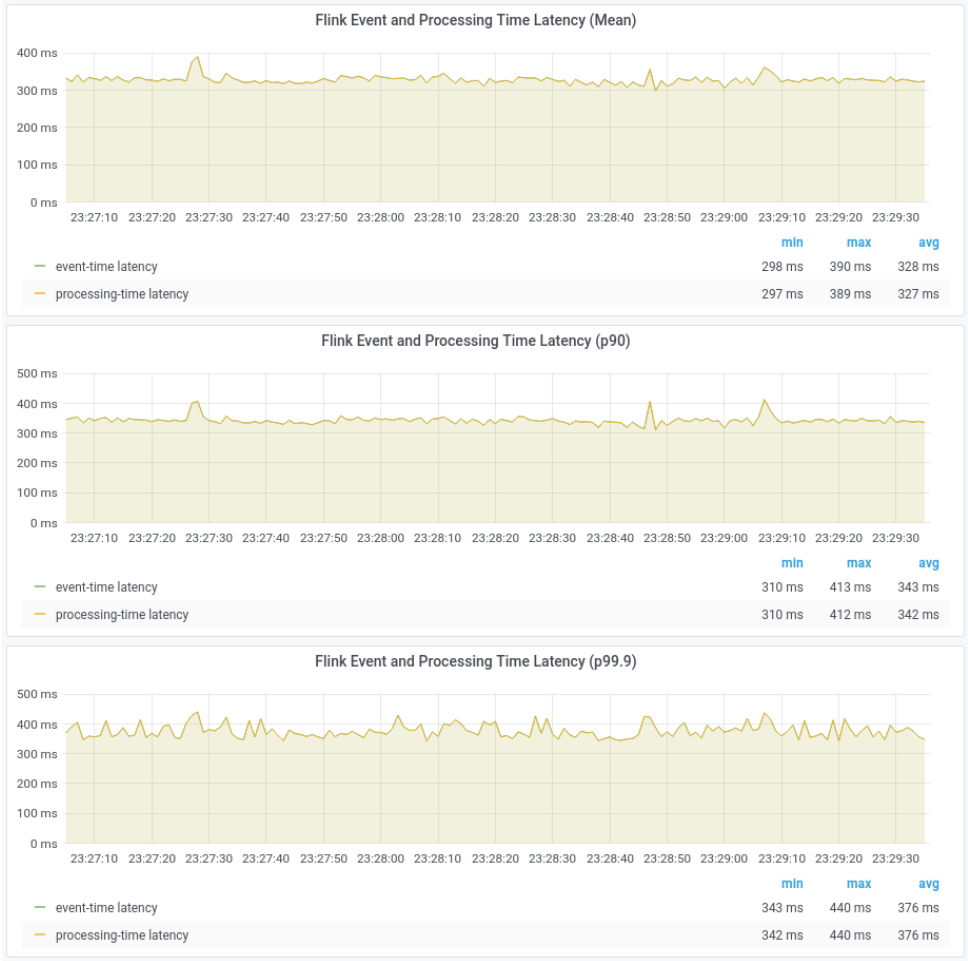


Figura 5.9: *Apache Flink* - 2 *Task Managers* - Latências

Relativamente ao uso de memória, a Figura 5.10 destaca exactamente as mesmas oscilações presentes na Figura 5.8. Em relação do uso do CPU, é visível que ambos os nós têm uma carga semelhante e bem distribuída.

Por último, em comparação com a análise feita na Secção 5.2.2 o tráfego em relação à transmissão e recepção de pacotes aumenta substancialmente, visível na Figura B.5 em anexo. A razão deste acontecimento parte de que a submissão da topologia no *Job Manager* implica que este delegue os operados entre os vários nós de *Task Manager*, e isto pode fazer com que haja uma maior comunicação entre estes nós para executar um operador que esteja delegado num *Task Manager*.

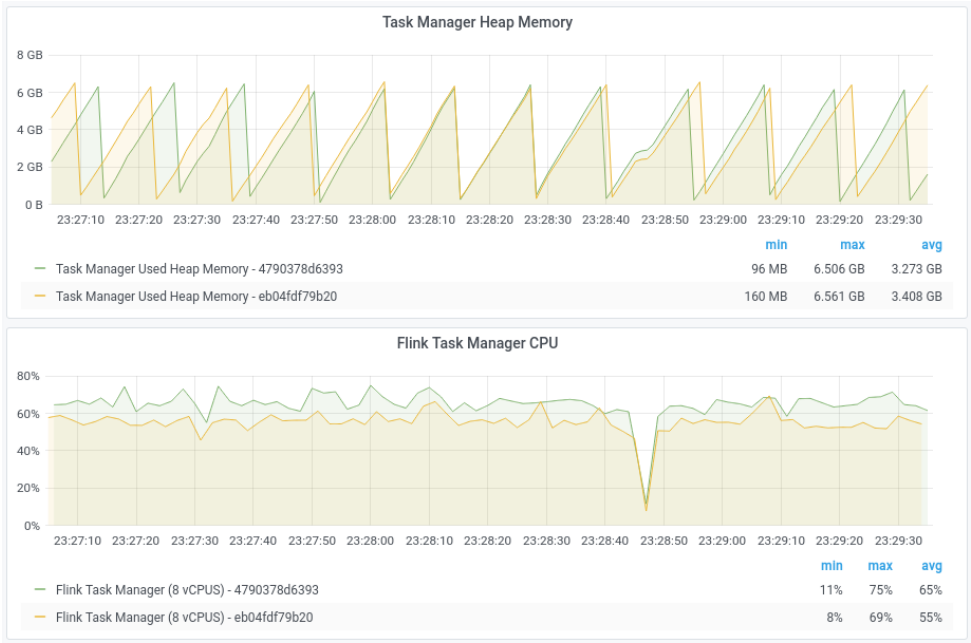


Figura 5.10: *Apache Flink* - 2 *Task Managers* - CPU e memória

4 *Task Managers*

Na terceira execução da topologia *Gira Travels Pattern*, com um *throughput* de 4000 eventos por segundo, as conclusões são semelhantes à que foi apresentada na Sub-Secção 5.2.2 sobre as latências, uso de CPU, de memória e tráfego, dado as Figuras 5.11, 5.12 e Figura B.6 em anexo.

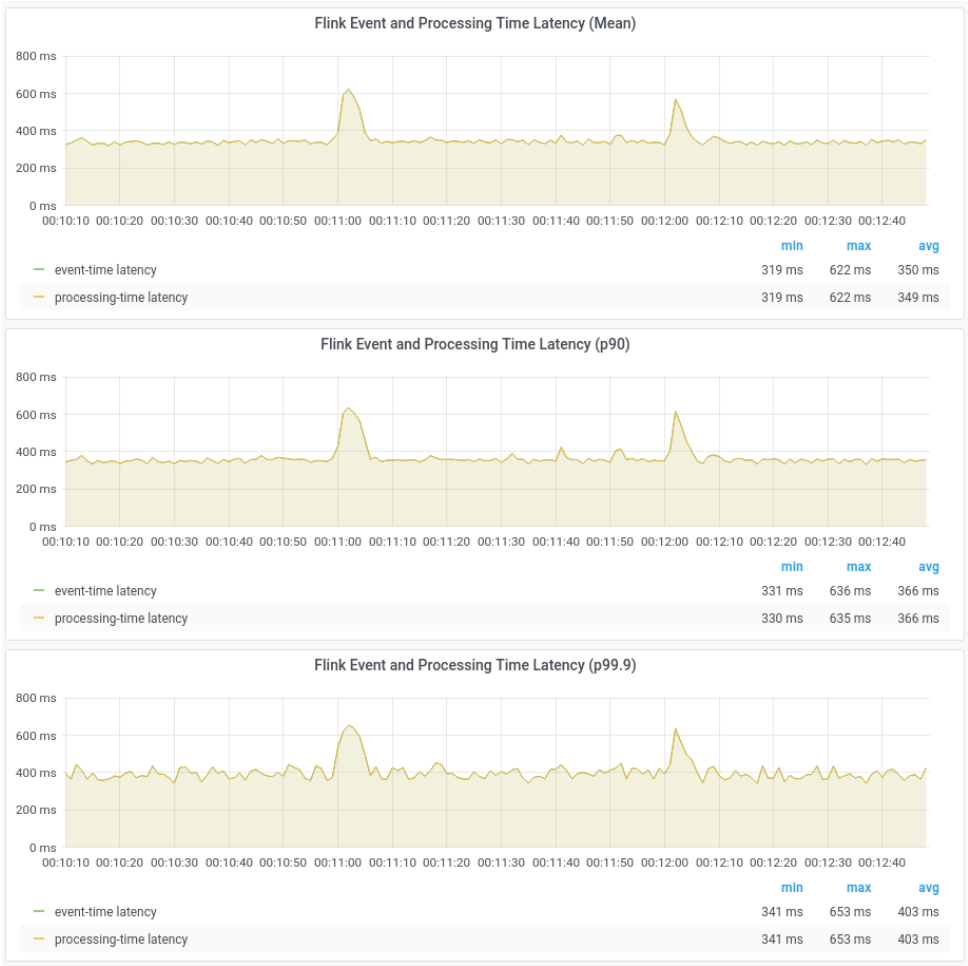


Figura 5.11: *Apache Flink* - 4 *Task Manager* - Latências

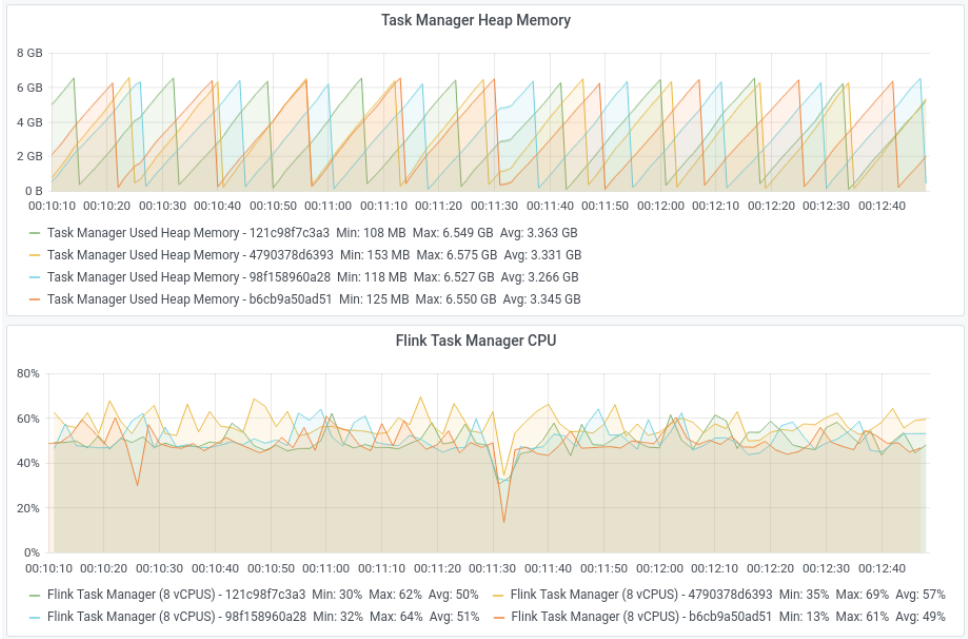


Figura 5.12: *Apache Flink* - 4 *Task Manager* - CPU e memória

5.2.3 *Apache Kafka*

As próximas Sub-Secções irão apresentar os resultados relativamente às latências de tempo de evento e de processamento, uso de memória e de CPU do nó de *Kafka* e das aplicações de *Kafka Streams*, e tráfego, obtidos durante a execução da topologia no *Apache Kafka*, com um, dois e quatro nós de *Kafka Streams*.

1 *Kafka Streams*

Na primeira execução da topologia *Gira Travels Pattern*, com um *throughput* de 1400 eventos por segundo, na Figura 5.13 observa-se que as latências de tempo de evento e de tempo de processamento têm oscilações significativas e a latência de tempo de evento em comparação à latência de tempo de processamento têm uma diferença acentuada.

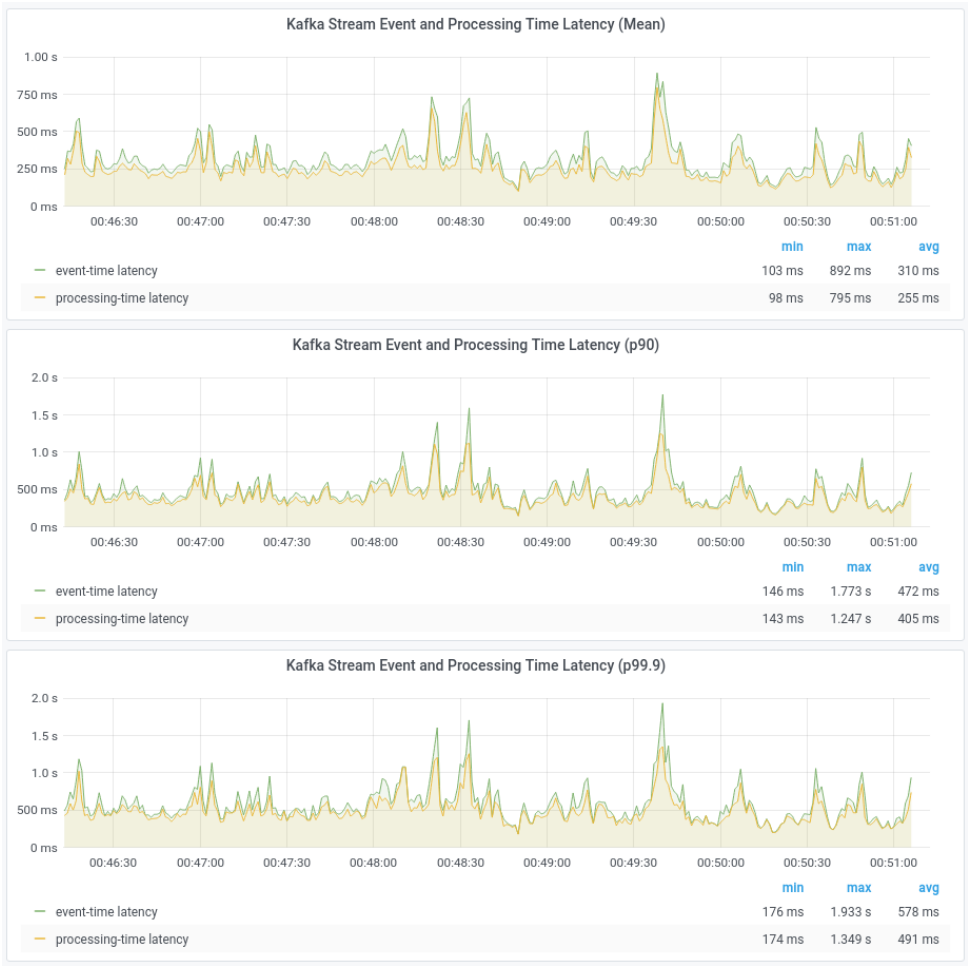


Figura 5.13: *Apache Kafka* - 1 *Kafka Streams* - Latências

Relativamente ao uso de CPU e de memória, são apresentadas as Figuras 5.14 e 5.15

onde uma é medido o nó de *Kafka* e outra em que é medido o nó onde a aplicação de *Kafka Streams* é executada. Para o *Apache Kafka* esta distinção é relevante porque o nó de Kafka desempenha um papel importante sobre alguns operadores, tais como *join* ou escrita de resultados do processamento para um tópico do *Kafka*, em que implica que as *streams*, com origem nos tópicos *Kafka* os eventos necessitam de serem materializados, ou seja, existe uma interacção, possivelmente, grande entre a aplicação de *Kafka Streams* e o nó de *Kafka*.

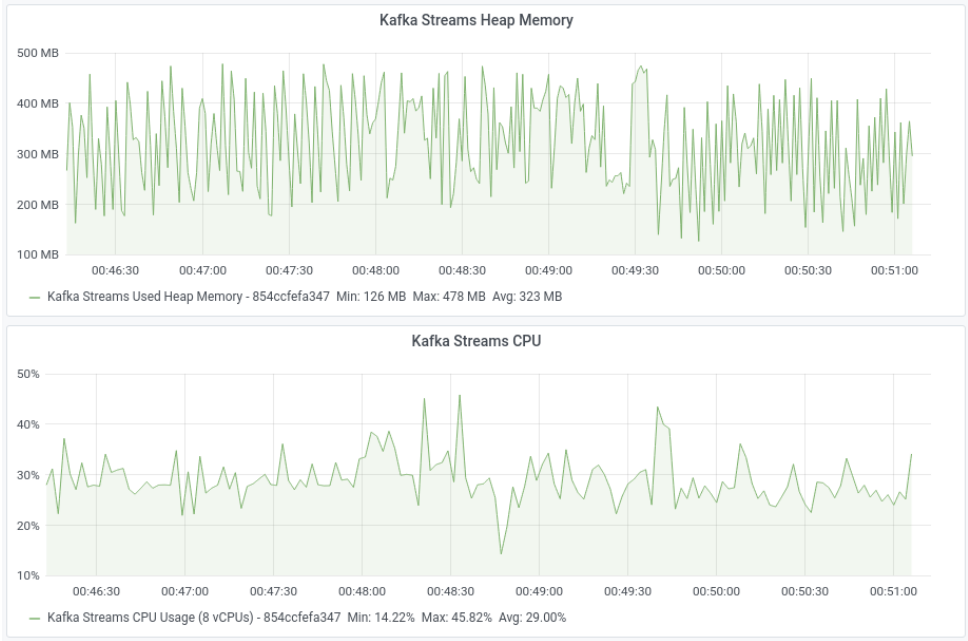


Figura 5.14: *Apache Kafka* - 1 *Kafka Streams* - CPU e memória

Por último, e pelas mesmas razões apresentadas anteriormente entre o nó de *Kafka* e o nó onde a aplicação de *Kafka Streams* é executada, o tráfego tem alguns padrões semelhantes entre a transmissão de pacotes da aplicação e a recepção de pacotes pelo nó de *Kafka*, como indicam as figuras B.7 e B.8 em anexo.

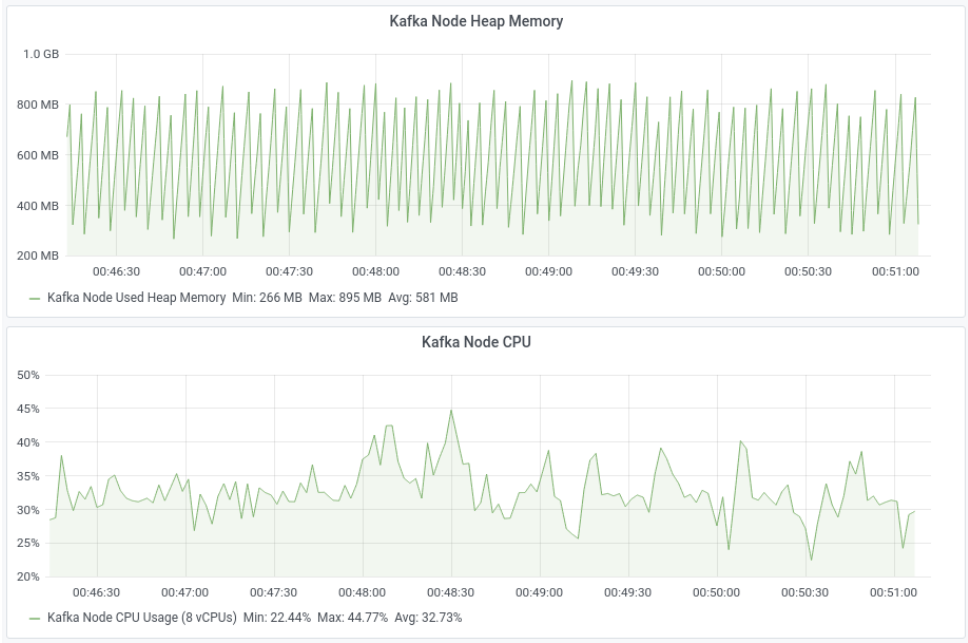


Figura 5.15: *Apache Kafka* - 1 *Kafka Streams* - *Kafka Node* - CPU e memória

2 *Kafka Streams*

Na segunda execução da topologia *Gira Travels Pattern*, com um *throughput* de 1600 eventos por segundo, na Figura 5.16 destaca-se um aumento nos tempos de latência do tempo de evento e de processamento.

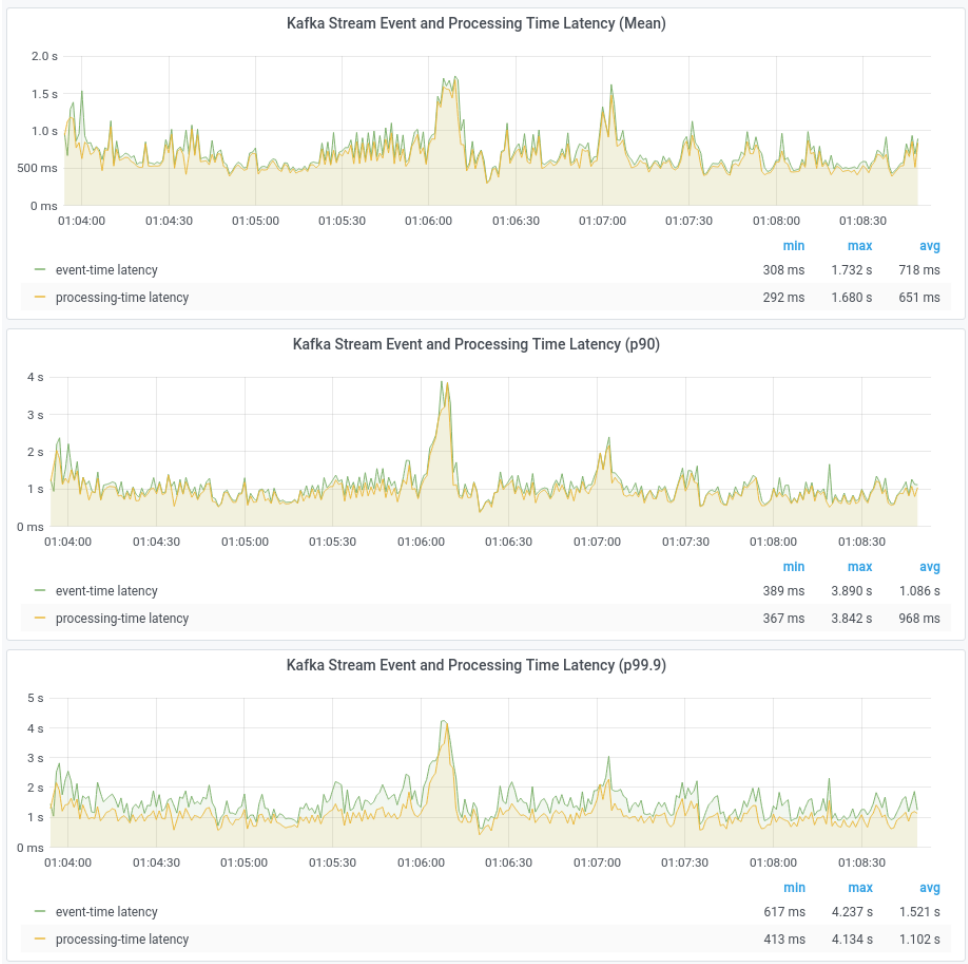


Figura 5.16: *Apache Kafka* - 2 *Kafka Streams* - Latências

Em relação ao uso de CPU e de memória, as Figuras 5.17 e 5.18 indicam que no nó de *Kafka* houve um aumento no nó de *Kafka* enquanto nas aplicações de *Kafka Streams*, em ambos os nós, os valores foram ligeiramente mais baixos em comparação com a execução anterior.

Nas Figuras B.9 e B.10 em anexo, destaca-se também que o tráfego para o nó de *Kafka* aumentou e nas aplicações de *Kafka Streams* a transmissão de pacotes obtiveram valores semelhantes mas com maior oscilação em relação à execução anterior.

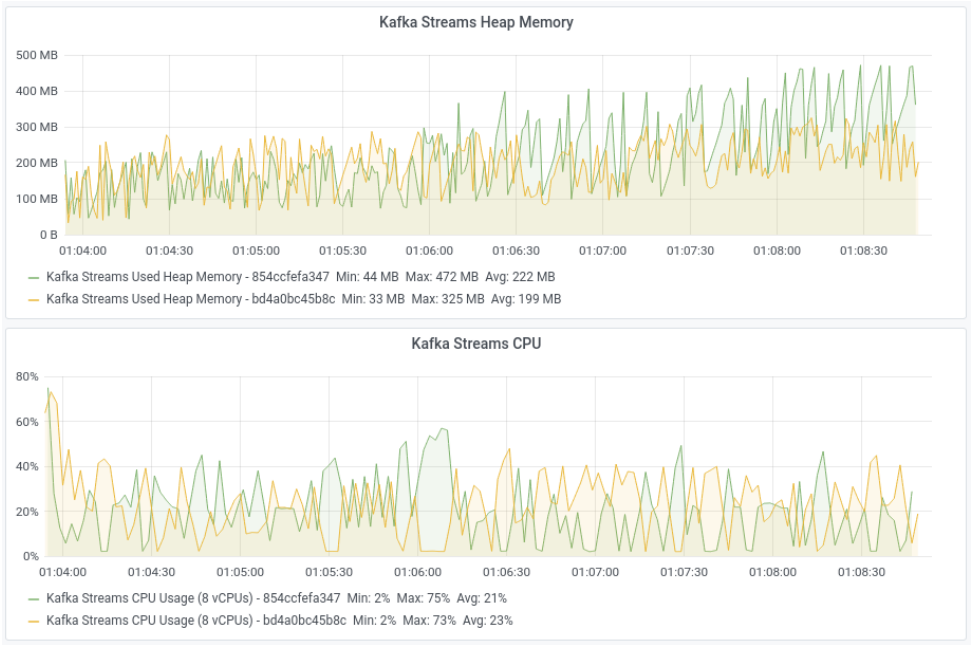


Figura 5.17: *Apache Kafka* - 2 *Kafka Streams* - CPU e memória

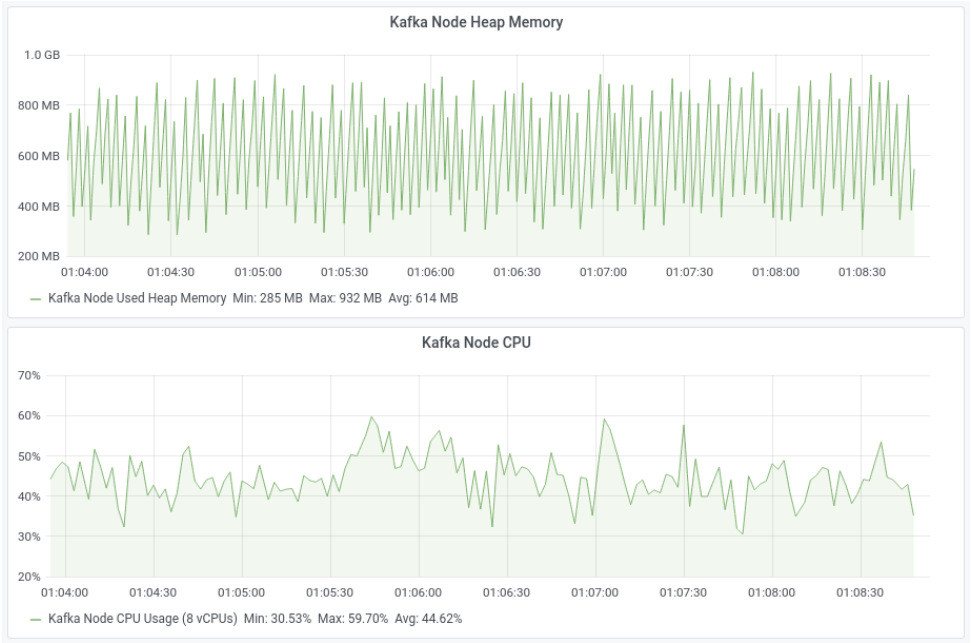


Figura 5.18: *Apache Kafka* - 2 *Kafka Streams* - *Kafka Node* - CPU e memória

4 *Kafka Streams*

Na terceira execução da topologia *Gira Travels Pattern*, com um *throughput* de 1900 eventos por segundo, a Figura 5.19 apresenta novamente um aumento das latências de tempo de evento e de processamento, no entanto o desfasamento entre o tempo de evento e de processamento diminui em comparação às execuções anteriores.

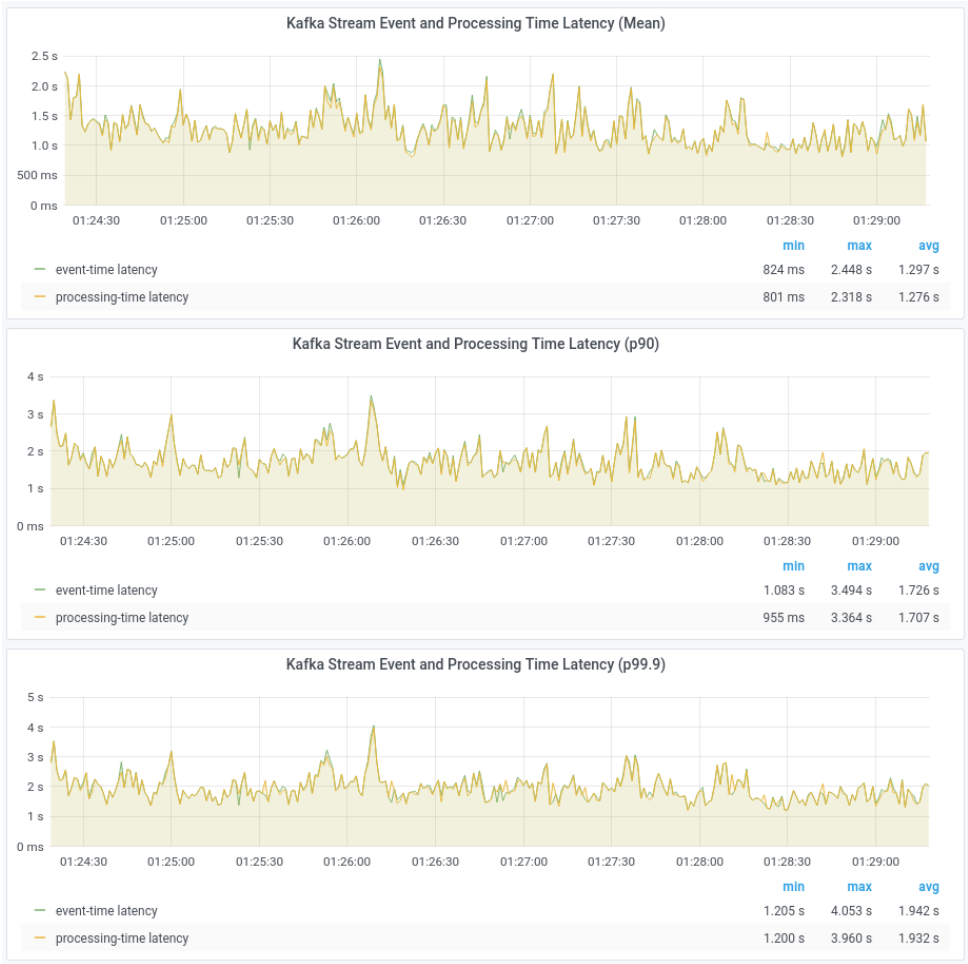


Figura 5.19: *Apache Kafka* - 4 *Kafka Streams* - Latências

Em relação ao uso de CPU e de memória, as Figuras 5.20 e 5.21 observam-se novamente um aumento no nó de *Kafka*, enquanto que nas aplicações de *Kafka Streams* o uso de CPU manteve-se perante o mesmo padrão que as execuções anterior, mas em comparação à execução anterior o uso de memória por parte das aplicações *Kafka Streams* aumentou.

Por fim, tanto a recepção de pacotes por parte do nó de *Kafka* como de transmissão de pacotes por parte das aplicações de *Kafka Streams* aumentaram, como indicam as Figuras B.11 e B.12 em anexo.

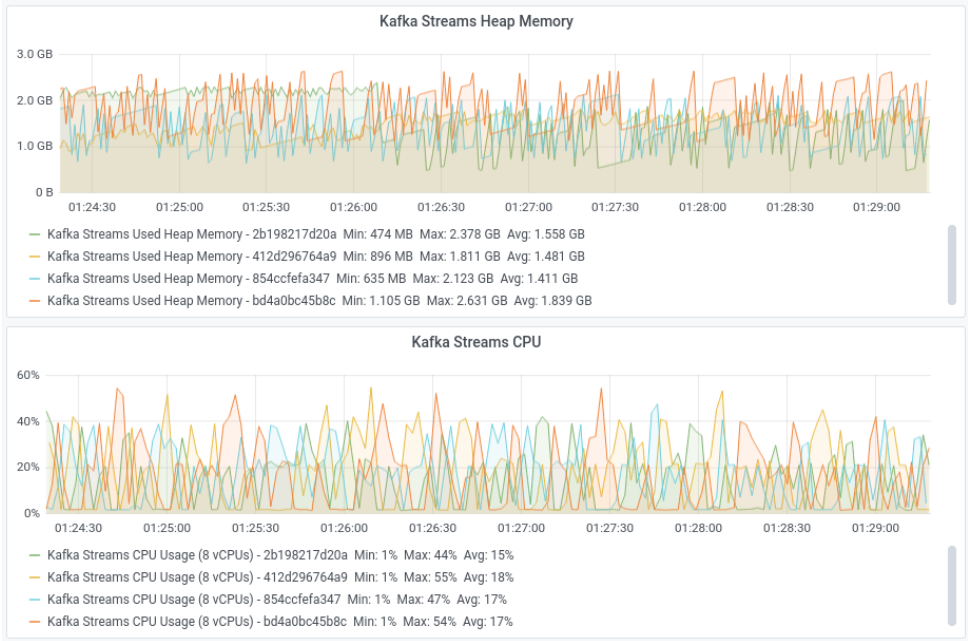


Figura 5.20: *Apache Kafka* - 4 *Kafka Streams* - CPU e memória

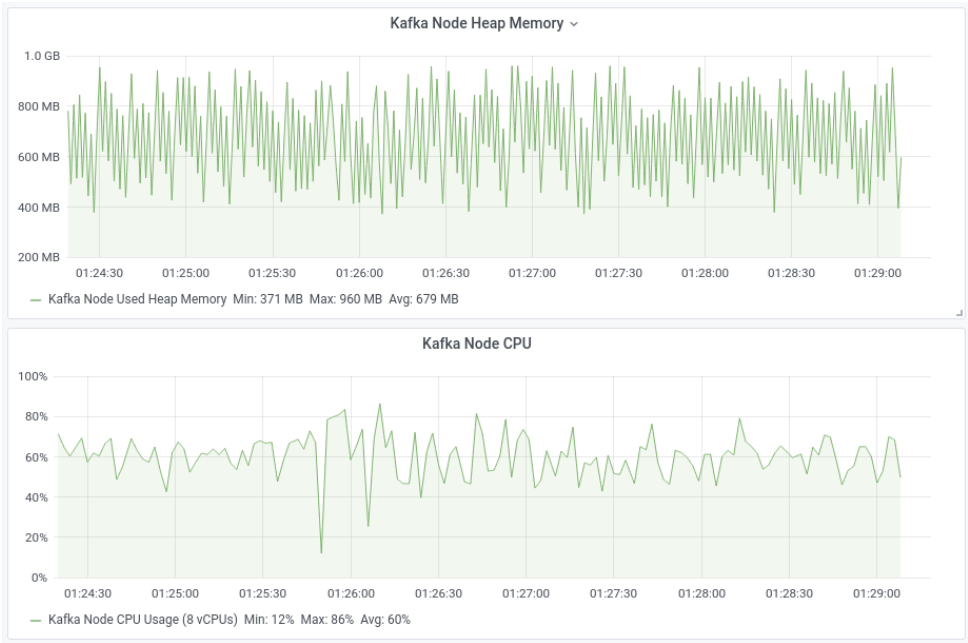


Figura 5.21: *Apache Kafka* - 4 *Kafka Streams* - *Kafka Node* - CPU e memória

5.3 Análise qualitativa das API

Durante os desenvolvimentos da topologia nos vários SPDS, foi possível realizar uma análise que permitisse categorizar o modelo de programação aplicada e à usabilidade das API, com base nas seguintes características:

1. **Aprendizagem** - quão simples é entender o que a API disponibiliza e o quão bem documentada está;
2. **Produtividade** - quão produtivo é o desenvolvimento ao utilizar a API;
3. **Simplicidade** - quão claro é o uso da API relativamente à sua aplicação em tempo de desenvolvimento;
4. **Extensibilidade** - como a API permite criar componentes específicos para casos de usos específicos;
5. **Prevenção de erros** - como a API previne o menor número de erros possíveis durante o desenvolvimento;

O uso da linguagem de programação *Java* nos desenvolvimentos da topologia em cada SPDS faz com que seja visível a aplicação de várias técnicas de alguns modelos de programação, tais como a programação imperativa e funcional. Para além dos modelos de programação, existem dois conceitos presentes nas API, nomeadamente o *Dataflow* [51][52] e grafos orientados sem ciclos (em inglês: *Directed Acyclic Graph* (DAG)).

O *Dataflow* é um paradigma que permite modelar um programa como um grafo orientado do fluxo de dados entre as operações definidas. Um DAG descreve um conjunto de nós e de vértices ligados directamente entre si, num só sentido e sem a existência de ciclos.

Enquadrando as API aos modelos de programação e conceitos apresentados acima, a API base do *Apache Storm*, *Spouts & Bolts* é categorizada como uma API do qual o seu desenvolvimento tem semelhança ao desenvolvimento de um DAG, seguindo um modelo de programação imperativa. Em relação às API *DataStream API* e *Streams DSL*, estas são categorizadas como API do qual o seu desenvolvimento têm semelhanças ao desenvolvimento de um *Dataflow*, seguindo um modelo de programação funcional. Sendo que a linguagem base destas API é o *Java*, logo um outro modelo de programação presente é a programação orientada aos objectos.

Tendo em conta o que foi apresentado nas secções 4.2.1, 4.2.2 e 4.2.3, no *Storm* é necessário definir como ligar os vários *Spouts* e *Bolts* entre si, enquanto que no *Flink* e *Kafka*

as operações definidas apresentam um fluxo bastante claro da sequência do processamento de dados, deixando a responsabilidade aos SPDS de construir o grafo mais optimizado possível. O *Storm* apresenta também outras API que se enquadra no conceito de *dataflow*, denominada de *Streams API*, no entanto esta não foi considerada para este trabalho pois apresenta alguns problemas de construção do grafo quando se pretende transpor as fases *First Join* e *Second Join*.

Relativamente à usabilidade das API, a *DataStream API* e *Streams DSL* são bastante semelhantes em relação aos operadores disponíveis e à forma como são usadas para definir o encadeamento do processamento dos dados, seguindo um estilo padrão *Builder*. Existem algumas particularidades, nomeadamente no operador *join*, em que na *Streams DSL* é necessário passar um conjunto de argumentos, enquanto que na *DataStream API* a sua definição é mais fluente. A API de *Spouts & Bolts* não tem uma comparação directa por se tratar de uma API de mais baixo nível em que é necessário criar as operações podendo levar à não reutilização dos operadores definidos e definir como é que estes operadores estão ligados entre si. Em relação à prevenção de erros, a API de *Spouts & Bolts* tem uma maior probabilidade de surgimento de erros uma vez que, primeiro, não é uma API tipificada, ou seja, a única estrutura que o *Storm* conhece é o *Tuple*. Segundo, como é necessário definir as ligações entres os vários *Spouts* e *Bolts*, facilmente podemos-nos enganar na *stream* pretendida para a respectiva fase, e isto só poderá ser detectado em tempo de execução da topologia. As restantes API ao serem tipificadas e ao existir um encadeamento lógico entre as operações usadas, torna-se menos provável o surgimento de erros em tempo de desenvolvimento. Ainda em relação à prevenção de erros, estes SPDS apresentam um momento de validação da topologia desenvolvida que permite detectar outros problemas que não são detectáveis em tempo de compilação.

Estes SPDS usam mecanismos de serialização para poderem trocar dados entre operadores ao qual a sua execução pode estar em processos diferentes. A API *DataStreams API* e a de *Spouts & Bolts*, apresentam mecanismos que permitem a não definição de serializadores, ou desserializadores, nomeadamente em *Plain Old Java Object* (POJO), reduzindo assim o custo de implementação. Ao contrário do *Streams DSL*, é necessário criar estas classes por cada classe de dados presente.

A *DataStreams API* e a API de *Spouts & Bolts* definem um conjunto de interfaces que permitem estender facilmente a sua utilização dependendo da especificidade do que se pretende implementar, por exemplo implementar um operador que comunique com sistemas externos. Para a *Streams DSL* apesar não apresentar interfaces semelhantes para desenvolver estes componentes, esta também é facilmente extensível uma vez que a própria aplicação de *Kafka Streams* ao ter estes componentes inicializados durante a

execução da aplicação, estes são passados por referência. As API de *DataStreams API* e *Spouts & Bolts* definem um conjunto de métodos que permite realizar este tipo de inicializações de outra forma, visto que o código desenvolvido por cada operador é depois transferido e inicializado pelos nós responsáveis de criar *tasks* para o processamento dos dados.

Com base na análise anterior entre as API usadas no desenvolvimentos da topologia nos SPDS, a tabela 5.5 apresenta o modelo de programação e a avaliação relativamente à usabilidade destas API.

Tabela 5.5: Tabela de avaliação das API

API / Características	*Storm - Spouts & Bolts*	*Flink - DataStream API*	*Kafka - Streams DSL*
Modelos de programação	Imperativa, Orientado aos Objectos	Imperativa, Orientado aos Objectos, Funcional	Imperativa, Orientado aos Objectos, Funcional
Conceitos de implementação	DAG	*Dataflow*	*Dataflow*
Aprendizagem	●●◐○○	●●●●●	●●●◐○
Produtividade	●●○○○	●●●●●	●●●◐○
Simplicidade	●●◐○○	●●●●○	●●●●○
Extensibilidade	●●●●○	●●●●○	●●●○○
Prevenção de erros	●●○○○	●●●●●	●●●●●

5.4 Análise dos resultados de execução

Durante a execução da topologia *Gira Travels Pattern* nos vários SPDS, foi possível realizar uma análise que permitisse comparar quantitativamente os resultados obtidos com ênfase em escalar horizontalmente o número de nós responsáveis pela execução de topologias de processamento de dados em *stream*, obtendo assim resultados distintos. Não é possível comparar estes resultados com outros estudos, pois a topologia contém duas junções de *streams*, uma entre os eventos das viagens Gira e eventos de trânsito do *Waze*, outra entre os eventos gerados da junção anterior e eventos de irregularidades do *Waze*. Outros estudos contêm somente uma única junção entre duas *streams*, ou então operações de agregações com *window*.

Relativamente aos resultados obtidos em cada SPDS, o *Apache Flink* é o que apresenta melhores resultados em comparação com o *Apache Storm* e *Apache Kafka*.

Comparando o *Apache Flink* com o *Apache Storm*, apesar da execução da topologia com

um nó de *Supervisor* e *Task Manager* ter uma diferença de cerca de 150 eventos por segundo, em relação às latências de tempo de evento e de processamento, o *Flink* atinge, no máximo, valores na ordem de grandeza dos milissegundos, enquanto que o *Storm* atinge, no mínimo, valores na ordem de grandeza dos segundos.

O *Flink* em relação ao uso de memória apresenta valores bastante mais altos que o *Storm*, no entanto quando pretendemos escalar os nós de *Supervisor* ou *Task Manager*, o *Storm* apresenta uma desproporcionalidade no uso de CPU entre nós, enquanto que o *Flink* apresenta uma distribuição mais uniforme. Para além de um uso mais distribuídos dos recursos, o *Flink* continua a garantir valores na ordem de grandeza dos milissegundos, com aumentos ligeiros, enquanto que o *Storm* mantém valores na ordem de grandeza dos segundos, e com aumentos significativos.

Em relação ao tráfego, o *Flink* apresenta valores mais altos, mas isto deve-se ao facto de poder existir comunicação entre operadores dado a definição da topologia submetida, porque os operadores são delegados pelos *Task Managers* que o *Job Manager* tenha registado.

Comparando o *Apache Flink* com o *Apache Kafka*, os resultados obtidos sobre as latências, durante a execução para 1 nó de *Task Manager* e 1 nó para a aplicação de *Kafka Streams*, são mais ou menos semelhantes em relação à ordem de grandeza dos valores, ou seja milissegundos, mas o *Kafka* apresenta oscilações substanciais nas latências. Da mesma forma que aconteceu com o *Storm*, o *Kafka* ao escalar os nós de *Kafka Streams*, as latências começam a aumentar, passando assim para uma ordem de grandeza dos segundos, mas a nível do uso dos recurso que os nós apresentam, o *Kafka* apresenta semelhanças ao *Flink*.

A nível de infraestrutura necessária para executar uma topologia no *Kafka*, versus no *Flink*, para o *Kafka* é necessário dar alguma relevância aos nós de *Kafka*, porque são estes os responsáveis por ter os eventos a serem consumidos pelas aplicações *Kafka Streams*, ou seja, por muito potente que os nós de aplicações de *Kafka Streams* sejam, o nó de *Kafka* pode-se tornar num *bottleneck*, logo pode trazer piores resultados. Para além do tipo de nó que o *Kafka* esteja instalado, os tópicos são persistentes, logo é necessário usar discos com velocidade de escrita e de leitura bastante rápidos, por exemplo, NVMe [53].

Com base na análise anterior sobre os resultados obtidos entre os SPDS, as tabelas 5.6, 5.7 e 5.8 apresentam um resumo dos resultados.

Tabela 5.6: Tabela de resumo dos resultados obtidos no *Apache Storm*
Apache Storm

		1 Nó	2 Nós	4 Nós
Latência (Média) (ms)	min	(1117, 1117)	(1117, 1117)	(2970, 2971)
	max	(2904, 2903)	(3218, 3219)	(5754, 5753)
	avg	(1840, 1839)	(2491, 2491)	(4099, 4098)
Latência (P90) (ms)	min	(1195, 1195)	(1190, 1190)	(4030, 4030)
	max	(3391, 3391)	(4277, 4276)	(7230, 7230)
	avg	(2150, 2150)	(3187, 3186)	(5300, 5300)
Latência (P99.9) (ms)	min	(1347, 1346)	(2011, 2010)	(4130, 4130)
	max	(3490, 3490)	(4412, 4411)	(7470, 7470)
	avg	(2292, 2292)	(3400, 3402)	(5510, 5510)
Memória (MB)	min	551	(474, 488)	(497, 501, 447, 501)
	max	1185	(979, 1073)	(1013, 1044, 1020, 1062)
	avg	840	(683, 767)	(794, 777, 748, 769)
CPU (%)	min	81.31	(47.51, 64.73)	(34.45, 33.59, 34.14, 56.01)
	max	92.71	(57.38, 81.64)	(46.07, 44.14, 43.85, 77.68)
	avg	85.43	(50.85, 72.92)	(37.07, 36.30, 36.60, 65.29)

Nos resultados das latências, os valores são expressos por latência do tempo de evento (LE) e latência do tempo te processamento (LP), da seguinte forma *(LE, LP)*, nas unidades do milissegundo;
Nos resultados do uso de memória e uso de CPU, os valores expressos são consoante o número de nós usados durante o teste;

Tabela 5.7: Tabela de resumo dos resultados obtidos no *Apache Flink*
Apache Flink

		1 Nó	2 Nós	4 Nós
Latência (Média) (ms)	min	(282, 282)	(298, 297)	(319, 319)
	max	(424, 424)	(390, 389)	(622, 622)
	avg	(312, 311)	(328, 327)	(350, 349)
Latência (P90) (ms)	min	(292, 292)	(310, 310)	(331, 330)
	max	(493, 492)	(413, 412)	(636,635)
	avg	(329, 328)	(343, 342)	(366, 366)
Latência (P99.9) (ms)	min	(304, 304)	(343, 342)	(341, 341)
	max	(533, 532)	(440, 440)	(653, 653)
	avg	(351, 351)	(376, 376)	(403, 403)
Memória (MB)	min	101	(96, 160)	(108, 153, 118, 125)
	max	6539	(6506, 6561)	(6549, 6575, 6527, 6550)
	avg	3339	(3273, 3408)	(3363, 3331, 3226, 3345)
CPU (%)	min	8	(11.00, 8.00)	(30, 35, 32, 13)
	max	72	(75, 69)	(62, 69, 64, 61)
	avg	62	(65, 55)	(50, 57, 51, 49)

Nos resultados das latências, os valores são expressos por latência do tempo de evento (LE) e latência do tempo te processamento (LP), nno seguinte formato *(LE, LP)*, em milissegundos;
Nos resultados do uso de memória e uso de CPU, os valores expressos são consoante o número de nós usados durante o teste;

Tabela 5.8: Tabela de resumo dos resultados obtidos no *Apache Kafka*
Apache Kafka

		1 Nó	2 Nós	4 Nós
Latência (Média) (ms)	min	(103, 98)	(308, 292)	(824, 801)
	max	(892, 795)	(1732, 1680)	(2448, 2318)
	avg	(310, 255)	(718, 651)	(1297, 1276)
Latência (P90) (ms)	min	(146, 143)	(389, 367)	(1083, 955)
	max	(1773, 1247)	(3890, 3842)	(3494, 3364)
	avg	(472, 405)	(1086, 968)	(1726, 1707)
Latência (P99.9) (ms)	min	(176, 174)	(617, 413)	(1205, 1200)
	max	(1933, 1349)	(4237, 4134)	(4053, 3960)
	avg	(578, 491)	(1521, 1102)	(1942, 1932)
Memória (MB)	min	126	(44, 33)	(474, 896, 635, 1105)
	max	478	(472, 325)	(2378, 1811, 2123, 2631)
	avg	323	(222, 199)	(1558, 1481, 1411, 1839)
CPU (%)	min	14.22	(2, 2)	(1, 1, 1, 1)
	max	45.82	(75, 73)	(44, 55, 47, 54)
	avg	29.00	(21, 23)	(15, 18, 17, 17)
Kafka Node Memória (MB)	min	266	285	371
	max	895	932	960
	avg	581	614	679
Kafka Node CPU (%)	min	22.44	30.53	12
	max	44.77	59.70	86
	avg	32.73	44.62	60

Nos resultados das latências, os valores são expressos por latência do tempo de evento (LE) e latência do tempo te processamento (LP), da seguinte forma *(LE, LP)*, nas unidades do milissegundo;
Nos resultados do uso de memória e uso de CPU, os valores expressos são consoante o número de nós usados durante o teste;

5.5 Resumo

A execução da topologia *Gira Travels Pattern* e obtenção de resultados permitiu que se quantificasse a avaliação dos SPDS em três vertentes, uma em que a instalação dos vários componentes de cada SPDS e arquitectura proposta são realizadas em *clouds* públicas. Outra em que permite escalar horizontalmente o número de nós de cada SPDS e obtendo resultados distintos. Por último, dar a entender as diferenças sobre o funcionamento de cada SPDS na execução de topologias de processamento de dados em *stream*.

6

Conclusões e Trabalhos Futuros

Neste trabalho apresentaram-se conceitos relacionados com o processamento de dados em *stream* e realizou-se uma comparação entre três SPDS, nomeadamente *Apache Storm*, *Apache Flink* e *Apache Kafka*. Foram caracterizados os elementos de análise quantitativa e qualitativa com base na realização da análise comparativa entre os SPDS.

Desenvolveu-se uma arquitectura com ênfase na substituibilidade e extensibilidade dos componentes envolventes e dos SPDS. Implementou-se a topologia *Gira Travels Pattern* para realizar o estudo sobre as API utilizadas e para extrair os resultados sobre o seu desempenho nos SPDS. Foram também desenvolvidos processos para gestão de infraestrutura em *clouds* públicas, nomeadamente na AWS, e processos de instalação dos SPDS e dos restantes componentes presentes na arquitectura definida, permitindo assim a automatização, simplificação e replicabilidade do trabalho.

Os resultados obtidos durante a execução da topologia *Gira Travels Pattern* em cada SPDS demonstram que o *Apache Flink* apresenta latências de tempo de evento e de processamento inferiores aos restantes SPDS, independentemente do número de nós utilizados durante os testes. No *Apache Storm* e no *Apache Kafka* à medida que é escalado o número de nós, as latências apresentaram uma tendência a aumentar.

A distribuição dos recursos computacionais revela-se mais uniforme no *Apache Flink* em comparação ao *Apache Storm*, uma vez que que este apresenta uma desproporcionalidade significativa quando se aumenta o número de nós. Em comparação ao *Apache Kafka*, este apresenta também uma distribuição dos recursos computacionais uniforme,

mas com alguma instabilidade. Além disso, o aumento do número de nós para execução da aplicação de *Kafka Streams* faz com que crie uma maior carga no nó de *Kafka* por existir uma maior concorrência.

Em relação ao tráfego, tanto o *Apache Flink* como o *Apache Kafka* apresentam valores significativos quanto ao seu uso, em especial no nó de *Kafka*. Este apresenta valores de recepção de pacotes superiores dado que as aplicações de *Kafka Streams* na operação de *join* implicam escrever as *streams* a juntar para dois tópicos intermédios. O *Apache Storm* apresenta valores igualmente desproporcionais em comparação com os restantes SPDS, visto que só um nó parece estar a realizar grande parte do processamento.

O máximo *throughput* sustentável indica que tanto o *Apache Flink* como o *Apache Storm* são semelhantes, mas o *Apache Kafka* está bastante abaixo do expectável. A razão disto prende-se com o facto de que no *Apache Kafka* o nó de *Kafka* poder ser considerado como um *bottleneck* dado que ao aumentar o número de nós de aplicações de *Kafka Streams* implicam uma maior quantidade de operações de leitura e de escrita em disco.

Em termos de trabalhos futuros, existe um conjunto de tópicos que seriam interessantes concretizar, sendo estes: Estender o *benchmark* desenvolvido para conter outro conjunto de SPDS, tais como, *Apache Samza*, *Hazelcast-Jet* e *Apache Heron*, de forma a complementar o *benchmark*; Estender o *benchmark* de forma a conter outro tipo de *workloads*, tais como *burst* inicial e periódico no envio de eventos para os SPDS; 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; Expandir a gestão de infraestrutura para outros fornecedores de *clouds* públicas, nomeadamente, *Google Cloud Plataform* e *Azure*; Criar novas topologias tendo como base o *benchmark* actual de forma a existir uma maior variedade de casos de processamento de dados em *stream*.

Referências Bibliográficas

[1] A. Batyuk & V. Voityshyn, "Apache storm based on topology for real-time processing of streaming data from social networks", em *2016 IEEE First International Conference on Data Stream Mining Processing (DSMP)*, 2016, páginas 345–349. DOI: 10.1109/DSMP.2016.7583573.

[2] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi & Kostas Tzoumas, "Apache flink™: Stream and batch processing in a single engine", *IEEE Data Eng. Bull.*, páginas 28–38, 2015.

[3] B. R. Hiraman, C. Viresh M. & K. Abhijeet C., "A study of apache kafka in big data stream processing", em *2018 International Conference on Information , Communication, Engineering and Technology (ICICET)*, 2018, páginas 1–3. DOI: 10.1109/ICICET.2018.8533771.

[4] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy & Siddarth Taneja, "Twitter heron: Stream processing at scale", em *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, sér. SIGMOD '15, Melbourne, Victoria, Australia: Association for Computing Machinery, 2015, 239–250, ISBN: 9781450327589. DOI: 10.1145/2723372.2742788. URL: https://doi.org/10.1145/2723372.2742788.

[5] Shadi A. Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta & Roy H. Campbell, "Samza: Stateful scalable stream processing at linkedin", *Proc. VLDB Endow.*, vol. 10, n.º 12, 1634–1645, ago. de 2017, ISSN: 2150-8097. DOI: 10.14778/3137765.3137770. URL: https://doi.org/10.14778/3137765.3137770.

[6] Hazelcast. (dez. de 2019). "Hazelcast Jet", URL: https://jet-start.sh/.

[7] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. J. Peng & P. Poulosky, "Benchmarking streaming computation engines: Storm, flink and spark streaming", em *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2016, páginas 1789–1792. DOI: 10.1109/IPDPSW.2016.138.

[8] E. Shahverdi, A. Awad & S. Sakr, "Big stream processing systems: An experimental evaluation", em *2019 IEEE 35th International Conference on Data Engineering Workshops (ICDEW)*, 2019, páginas 53–60. DOI: 10.1109/ICDEW.2019.00-35.

[9] G. van Dongen & D. Van den Poel, "Evaluation of stream processing frameworks", *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, n.º 8, páginas 1845–1858, 2020. DOI: 10.1109/TPDS.2020.2978480.

[10] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen & V. Markl, "Benchmarking distributed stream data processing systems", em *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, 2018, páginas 1507–1518. DOI: 10.1109/ICDE.2018.00169.

[11] Câmara Municipal de Lisboa. (jan. de 2021). "*LXDATALAB*", URL: https://lisboainteligente.cm-lisboa.pt/lxdatalabcs/.

[12] Confluent. (dez. de 2019). "Streams Concepts", URL: https://docs.confluent.io/current/streams/concepts.html#stream.

[13] Apache Software Foundation. (2019). "Apache Storm - Tutorial", URL: https://storm.apache.org/releases/2.2.0/Tutorial.html.

[14] Apache Software Foundation. (jan. de 2021). "Apache Flink - Anatomy of a Flink Cluster", URL: https://ci.apache.org/projects/flink/flink-docs-release-1.11/concepts/flink-architecture.html#anatomy-of-a-flink-cluster.

[15] Apache Software Foundation. (jan. de 2021). "Apache Flink - Tasks and Operator Chains", URL: https://ci.apache.org/projects/flink/flink-docs-release-1.11/concepts/flink-architecture.html#anatomy-of-a-flink-cluster.

[16] Apache Software Foundation. (dez. de 2019). "Kafka Streams", URL: https://kafka.apache.org/documentation/streams/.

[17] Apache Software Foundation. (dez. de 2019). "Kafka Streams Architecture", URL: https://kafka.apache.org/24/documentation/streams/architecture.

[18] Docker. (dez. de 2019). "Docker", URL: https://www.docker.com/.

[19] HashiCorp. (set. de 2020). "Terraform", URL: https://www.terraform.io/.

[20] Red Hat. (set. de 2020). "Ansible", URL: https://www.ansible.com/.

[21] Docker. (dez. de 2019). "Docker Compose", URL: https://docs.docker.com/compose/.

[22] Influxdata. (set. de 2020). "Telegraf", URL: https://www.influxdata.com/time-series-platform/telegraf/.

[23] Influxdata. (set. de 2020). "InfluxDB", URL: https://www.influxdata.com/products/influxdb-overview/.

[24] Grafana Labs. (set. de 2020). "Grafana", URL: https://grafana.com/.

[25] Oracle. (set. de 2020). "Java Management Extensions (JMX)", URL: https://www.oracle.com/technical-resources/articles/javase/jmx.html.

[26] Etsy. (set. de 2020). "StatsD", URL: https://github.com/statsd/statsd.

[27] Henriette Röger & Ruben Mayer, "A comprehensive survey on parallelization and elasticity in stream processing", *ACM Comput. Surv.*, vol. 52, n.º 2, abr. de 2019, ISSN: 0360-0300. DOI: 10.1145/3303849. URL: https://doi.org/10.1145/3303849.

[28] Yahoo. (dez. de 2019). "Yahoo Streaming Benchmark", URL: https://github.com/yahoo/streaming-benchmarks.

[29] Redislabs. (dez. de 2019). "Redis", URL: https://redis.io/.

[30] O. Marcu, A. Costan, G. Antoniu & M. S. Pérez-Hernández, "Spark versus flink: Understanding performance in big data analytics frameworks", em *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, 2016, páginas 433–442. DOI: 10.1109/CLUSTER.2016.22.

[31] IEEE, Giselle van Dongen, Dirk Van den Poeal. (set. de 2020). "Open Stream Processing benchmark", URL: https://github.com/Klarrio/open-stream-processing-benchmark.

[32] Apache Software Foundation. (set. de 2020). "Storm - Concepts", URL: https://storm.apache.org/releases/2.2.0/Concepts.html.

[33] Apache Software Foundation. (set. de 2020). "Flink - Process Function", URL: https://ci.apache.org/projects/flink/flink-docs-release-1.11/dev/stream/operators/process_function.html.

[34] Apache Software Foundation. (set. de 2020). "Storm - Streams API - Overview", URL: https://storm.apache.org/releases/2.2.0/Stream-API.html.

[35] Apache Software Foundation. (set. de 2020). "Flink - DataStream API - Overview", URL: https://ci.apache.org/projects/flink/flink-docs-release-1.11/dev/datastream_api.html.

[36] Apache Software Foundation. (set. de 2020). "Kafka Streams - Streams DSL", URL: https://kafka.apache.org/26/documentation/streams/developer-guide/dsl-api.html.

[37] Apache Software Foundation. (set. de 2020). "Storm SQL", URL: https://storm.apache.org/releases/2.2.0/storm-sql.html.

[38] Apache Software Foundation. (set. de 2020). "Flink - Table API & SQL - Overview", URL: https://ci.apache.org/projects/flink/flink-docs-release-1.11/dev/datastream_api.html.

[39] Confluent Inc. (set. de 2020). "KSQL", URL: https://www.confluent.io/blog/ksql-streaming-sql-for-apache-kafka/.

[40] Pivotal Software. (set. de 2020). "RabbitMQ", URL: https://www.rabbitmq.com/.

[41] Pivotal Software. (jan. de 2021). "RabbitMQ Management HTTP API", URL: https://www.rabbitmq.com/management.html#http-api-monitoring.

[42] Docker, *Docker Engine API*, jan. de 2021. URL: https://docs.docker.com/engine/api/.

[43] Influxdata. (jan. de 2021). "Influx DB Line Protocol", URL: https://docs.influxdata.com/influxdb/v1.8/write_protocols/line_protocol_reference/.

[44] Influxdata. (jan. de 2021). "Influx Query Language", URL: https://docs.influxdata.com/influxdb/v1.8/query_language/.

[45] Câmara Municipal de Lisboa. (dez. de 2019). "Lisboa Inteligente - Existem padrões de utilização das bicicletas partilhadas em Lisboa?", URL: https://lisboainteligente.cm-lisboa.pt/lxdatalab/desafios/existem-padroes-de-utilizacao-das-bicicletas-partilhadas-em-lisboa/.

[46] Wikipedia. (set. de 2020). "Well-known text representation of geometry", URL: https://en.wikipedia.org/wiki/Well-known_text_representation_of_geometry.

[47] Rodrigo Bruno, Duarte Patricio, José Simão, Luis Veiga & Paulo Ferreira, "Runtime object lifetime profiler for latency sensitive big data applications", em *Proceedings of the Fourteenth EuroSys Conference 2019*, sér. EuroSys '19, Dresden, Germany: Association for Computing Machinery, 2019, ISBN: 9781450362818. DOI: 10.1145/3302424.3303988. URL: https://doi.org/10.1145/3302424.3303988.

[48] Oracle. (jan. de 2021). "The garbage first garbage collector", URL: https://www.oracle.com/java/technologies/javase/hotspot-garbage-collection.html.

[49] Oracle. (jan. de 2021). "The parallel collector", URL: https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/parallel.html.

[50] Amazon Web Services. (jan. de 2021). "Amazon ec2 c5 instances", URL: https://aws.amazon.com/ec2/instance-types/c5/.

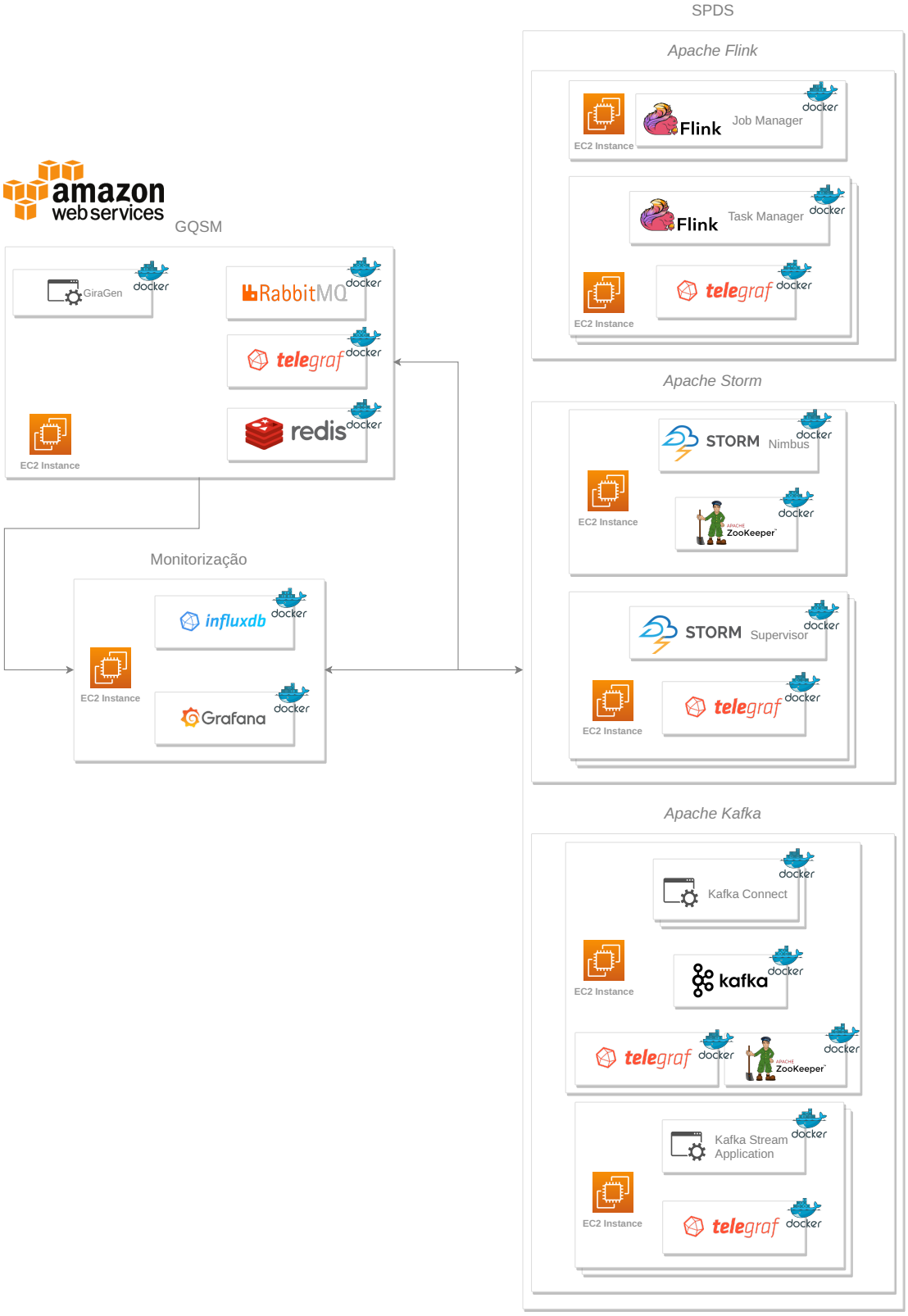
[51] Tiago Boldt Sousa, "Dataflow programming concept, languages and applications", 2012. URL: https://paginas.fe.up.pt/~prodei/dsie12/papers/proceedings.pdf.

[52] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt & Sam Whittle, "The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing", *Proceedings of the VLDB Endowment*, vol. 8, páginas 1792–1803, 2015.

[53] Qiumin Xu, Huzefa Siyamwala, Mrinmoy Ghosh, Tameesh Suri, Manu Awasthi, Zvika Guz, Anahita Shayesteh & Vijay Balakrishnan, "Performance analysis of nvme ssds and their implication on real world databases", em *Proceedings of the 8th ACM International Systems and Storage Conference*, sér. SYSTOR '15, Haifa, Israel: Association for Computing Machinery, 2015, ISBN: 9781450336079. DOI: 10.1145/2757667.2757684. URL: https://doi.org/10.1145/2757667.2757684.

Desenvolvimento da infraestrutura para os SPDS

A.1 Diagrama geral da infraestrutura



A.2 Implementação e gestão da infraestrutura dos SPDS com *Terraform*

A.2.1 Definicão de Terraform Module para *EC2*

```
locals {
    instance_name_prefix = "${var.name}-"
    instance_dns_prefix  = "${var.name}-"
}

data "template_file" "cloud_init" {
    count = var.instance_count

    template = file("${path.module}/cloud_config.yaml")

    vars = {
        tpl_hostname            = "${local.instance_name_prefix}${count.index}"
        tpl_ssh_authorized_keys = join("\n      - ", var.ssh_authorized_keys)
    }
}

resource "aws_instance" "default" {
    count         = var.instance_count
    instance_type = var.instance_type
    ami           = "ami-093185e1a0acee74b" ## Debian Buster

    monitoring                  = false
    ebs_optimized               = true
    associate_public_ip_address = true

    availability_zone = var.zone

    vpc_security_group_ids = [
        var.aws_security_group_id
    ]

    key_name = var.key_pair_name
```

```
33
34      user_data = element(data.template_file.cloud_init.*.rendered,
        count.index)
35
36      root_block_device {
37          volume_type = "gp2"
38          volume_size = 20
39          encrypted   = false
40      }
41
42      tags = {
43          Name        = "${local.instance_name_prefix}${count.index}"
44          Type        = var.name
45          Provisioner = "terraform"
46      }
47  }
```

Listagem A.1: *Terraform EC2 Module*

A.2.2 Definição de instâncias para o SPDS *Apache Storm*

```
1  /*==============================
2    INFRASTRUCTURE
3  ==============================*/
4
5  module "access" {
6    source = "../modules/access"
7  }
8
9  module "networking" {
10   source = "../modules/networking"
11 }
12
13 /*==============================
14   Misc Infrastrutrure
15 ==============================*/
16
17 module "misc-infrastructure" {
18   source = "../modules/ec2"
19
```

```
  name = "misc-infrastructure"

  instance_type = "c5a.4xlarge"
  instance_count = 1

  zone = "eu-west-1b"
  ssh_authorized_keys = local.ssh_authorized_keys

  aws_security_group_id = module.networking.aws_security_group_id
  key_pair_name = module.access.key_pair_name
}
/*==============================
  Apache Storm
==============================*/

module "storm-nimbus" {
  source = "../modules/ec2"

  name = "storm-nimbus"

  instance_type = "c5a.large"
  instance_count = 1

  zone = "eu-west-1b"
  ssh_authorized_keys = local.ssh_authorized_keys

  aws_security_group_id = module.networking.aws_security_group_id
  key_pair_name = module.access.key_pair_name
}

module "storm-supervisor" {
  source = "../modules/ec2"

  name = "storm-supervisor"

  instance_type = "c5a.2xlarge"
  instance_count = 4

  zone = "eu-west-1b"
  ssh_authorized_keys = local.ssh_authorized_keys
```

```
60
61   aws_security_group_id = module.networking.aws_security_group_id
62   key_pair_name = module.access.key_pair_name
63 }
64
65 /*==============================
66   Metrics Dashboard
67 ==============================*/
68
69 module "metrics-dashboard" {
70   source = "../modules/ec2"
71
72   name = "metrics-dashboard"
73
74   instance_type = "c5a.large"
75   instance_count = 1
76
77   zone = "eu-west-1b"
78   ssh_authorized_keys = local.ssh_authorized_keys
79
80   aws_security_group_id = module.networking.aws_security_group_id
81   key_pair_name = module.access.key_pair_name
82 }
```

Listagem A.2: Configuração das instâncias para o *Apache Storm*

A.2.3 Definição de instâncias para o SPDS *Apache Flink*

```
1 /*==============================
2   INFRASTRUCTURE
3 ==============================*/
4
5 module "access" {
6   source = "../modules/access"
7 }
8
9 module "networking" {
10   source = "../modules/networking"
11 }
12
```

```
/*==============================
  Misc Infrastrutrure
==============================*/

module "misc-infrastructure" {
  source = "../modules/ec2"

  name = "misc-infrastructure"

  instance_type = "c5a.4xlarge"
  instance_count = 1

  zone = "eu-west-1b"
  ssh_authorized_keys = local.ssh_authorized_keys

  aws_security_group_id = module.networking.aws_security_group_id
  key_pair_name = module.access.key_pair_name
}

/*==============================
  Apache Flink
==============================*/

module "flink-job-manager" {
  source = "../modules/ec2"

  name = "flink-job-manager"

  instance_type = "c5a.large"
  instance_count = 1

  zone = "eu-west-1b"
  ssh_authorized_keys = local.ssh_authorized_keys

  aws_security_group_id = module.networking.aws_security_group_id
  key_pair_name = module.access.key_pair_name
}

module "flink-task-manager" {
  source = "../modules/ec2"
```

```
53
54   name = "flink-task-manager"
55
56   instance_type = "c5a.2xlarge"
57   instance_count = 4
58
59   zone = "eu-west-1b"
60   ssh_authorized_keys = local.ssh_authorized_keys
61
62   aws_security_group_id = module.networking.aws_security_group_id
63   key_pair_name = module.access.key_pair_name
64 }
65
66 /*==============================
67   Metrics Dashboard
68 ==============================*/
69
70 module "metrics-dashboard" {
71   source = "../modules/ec2"
72
73   name = "metrics-dashboard"
74
75   instance_type = "c5a.large"
76   instance_count = 1
77
78   zone = "eu-west-1b"
79   ssh_authorized_keys = local.ssh_authorized_keys
80
81   aws_security_group_id = module.networking.aws_security_group_id
82   key_pair_name = module.access.key_pair_name
83 }
```

Listagem A.3: Configuração das instâncias para o *Apache Flink*

A.2.4 Definição de instâncias para o SPDS *Apache Kafka*

```
1 /*==============================
2   INFRASTRUCTURE
3 ==============================*/
4
```

```
5  module "access" {
6    source = "../modules/access"
7  }
8
9  module "networking" {
10   source = "../modules/networking"
11 }
12
13 /*==============================
14   Misc Infrastrutrure
15 ==============================*/
16
17 module "misc-infrastructure" {
18   source = "../modules/ec2"
19
20   name = "misc-infrastructure"
21
22   instance_type = "c5a.4xlarge"
23   instance_count = 1
24
25   zone = "eu-west-1b"
26   ssh_authorized_keys = local.ssh_authorized_keys
27
28   aws_security_group_id = module.networking.aws_security_group_id
29   key_pair_name = module.access.key_pair_name
30 }
31
32
33 /*==============================
34   Apache Kafka
35 ==============================*/
36
37 module "kafka-node" {
38   source = "../modules/ec2"
39
40   name = "kafka-node"
41
42   instance_type = "c5ad.2xlarge"
43   instance_count = 1
44
```

```
45   volume_size = 100
46
47   zone = "eu-west-1b"
48   ssh_authorized_keys = local.ssh_authorized_keys
49
50   aws_security_group_id = module.networking.aws_security_group_id
51   key_pair_name = module.access.key_pair_name
52 }
53
54 module "kafka-stream" {
55   source = "../modules/ec2"
56
57   name = "kafka-stream"
58
59   instance_type = "c5a.2xlarge"
60   instance_count = 4
61
62   zone = "eu-west-1b"
63   ssh_authorized_keys = local.ssh_authorized_keys
64
65   aws_security_group_id = module.networking.aws_security_group_id
66   key_pair_name = module.access.key_pair_name
67 }
68
69 /*============================
70   Metrics Dashboard
71 ============================*/
72
73 module "metrics-dashboard" {
74   source = "../modules/ec2"
75
76   name = "metrics-dashboard"
77
78   instance_type = "c5a.large"
79   instance_count = 1
80
81   zone = "eu-west-1b"
82   ssh_authorized_keys = local.ssh_authorized_keys
83
84   aws_security_group_id = module.networking.aws_security_group_id
```

```
85   key_pair_name = module.access.key_pair_name
86 }
```

Listagem A.4: Configuração das instâncias para o *Apache Flink*

A.3 Implementação da instalação dos SPDS com *Ansible*

A.3.1 Definição de *Ansible Role* para instalação de *Docker containers* com *docker-compose*

```
1  ---
2  - name: Adding user 'impads' to group 'docker'
3    user:
4      name: 'admin'
5      groups: docker
6      append: yes
7
8  - name: Directory creation
9    file:
10     recurse: true
11     state: directory
12     path: "{{ compose.src }}/{{ compose.name }}"
13     group: "admin"
14     owner: "admin"
15
16 - name: Put docker compose file
17   template:
18     src: "{{ item.src }}"
19     dest: "{{ compose.src }}/{{ compose.name }}/{{ item.dest }}"
20     group: "admin"
21     owner: "admin"
22   with_items:
23     - { src: "docker-compose.yml.j2", dest: "docker-compose.yml" }
24
25 - name: Put properties
26   template:
27     src: "{{ item.src }}"
28     dest: "{{ compose.src }}/{{ compose.name }}/{{ item.from }}"
```

```
29      group: "admin"
30      owner: "admin"
31    with_items:
32      - "{{ compose.services | selectattr('properties', 'defined') |
       map(attribute='properties') | list | flatten }}"
33    when: compose.services | selectattr("properties", "defined") | list
        | length > 0
34
35  - name: Put files
36    synchronize:
37      src: "{{ item.src }}"
38      dest: "{{ compose.src }}/{{ compose.name }}/{{ item.from }}"
39    with_items:
40      - "{{ compose.services | selectattr('files', 'defined') | map(
       attribute='files') | list | flatten }}"
41    when: compose.services | selectattr("files", "defined") | list |
       length > 0
42
43  - name : Stop services
44    docker_compose:
45      project_src: "{{ compose.src }}/{{ compose.name }}"
46      build: no
47      pull: yes
48      state: present
49      stopped: true
50    register: output
51
52  - name: Start services
53    docker_compose:
54      project_src: "{{ compose.src }}/{{ compose.name }}"
55      build: no
56      pull: yes
57      state: present
58      restarted: true
59    register: output
```

Listagem A.5: *Ansible Role - Docker Compose*

```
1 version: "3.7"
2
3 services:
```

```
4
5  {% for service in compose.services %}
6      {{ service.name }}:
7          image: {{ service.image }}
8          restart: {{ service.restart | default('always') }}
9  {% if service.user is defined %}
10         user: {{ service.user }}
11 {% endif %}
12 {% if service.container_name is defined %}
13         container_name: {{ service.container_name }}
14 {% endif %}
15 {% if service.command is defined %}
16         command: {{ service.command }}
17 {% endif %}
18 {% if service.environment is defined %}
19         environment:
20 {% for env in service.environment %}
21             - {{ env }}
22 {% endfor %}
23 {% if service.conditional_environment is defined %}
24 {% for condition_env in service.conditional_environment %}
25 {% if condition_env.predicate %}
26             - {{ condition_env.value }}
27 {% endif %}
28 {% endfor %}
29 {% endif %}
30 {% endif %}
31 {% if service.ports is defined %}
32         ports:
33 {% for port in service.ports  %}
34             - {{ port }}
35 {% endfor %}
36 {% endif %}
37 {% if service.properties is defined or service.files is defined or
   service.volumes is defined %}
38         volumes:
39 {% if service.properties is defined %}
40 {% for property in service.properties %}
41             - ./{{ property.from }}:{{ property.to }}
42 {% endfor %}
```

```
{% endif %}
{% if service.files is defined %}
{% for file in service.files %}
            - ./{{ file.from }}:{{ file.to }}
{% endfor %}
{% endif %}
{% if service.volumes is defined %}
{% for volume in service.volumes %}
            - {{ volume }}
{% endfor %}
{% endif %}
{% endif %}
{% endfor %}

networks:
  default:
    external:
```

Listagem A.6: *Ansible Role - Docker Compose - Jinja2 Template*

A.3.2 Definicão de *Ansible Role* para instalação do *Storm*

```
---
- name: Deploy Strom Nimbus
  include_role:
    name: container/docker/compose
  vars:
    compose:
      src: /opt/impads/containers
      name: storm
      services:
        - name: zookeeper
          image: bitnami/zookeeper:3.6.1
          container_name: zookeeper
          environment:
            - ALLOW_ANONYMOUS_LOGIN=yes
          ports:
            - "2181:2181"
        - name: nimbus
          image: impads/storm:2.2.0
```

```
19          container_name: nimbus
20          command: storm nimbus
21          ports:
22            - "6627:6627"
23            - "8000:8000"
24          properties:
25            - { src: storm.yaml.j2, from: storm.yaml, to: /conf/storm
    .yaml }
26        - name: nimbus-ui
27          image: impads/storm:2.2.0
28          container_name: nimbus-ui
29          command: storm ui
30          ports:
31            - "8080:8080"
32          properties:
33            - { src: storm.yaml.j2, from: storm.yaml, to: /conf/storm
    .yaml }
```

Listagem A.7: *Ansible Role - Apache Storm - Nimbus*

```
1 ---
2 - name: Deploy Strom Nimbus
3   include_role:
4     name: container/docker/compose
5   vars:
6     compose:
7       src: /opt/impads/containers
8       name: storm
9       services:
10        - name: supervisor
11          image: impads/storm:2.2.0
12          container_name: supervisor
13          command: storm supervisor
14          ports:
15            - "8000:8000"
16            - "6700:6700"
17            - "6701:6701"
18            - "6702:6702"
19            - "6703:6703"
20            - "6704:6704"
21            - "6705:6705"
```

```
22            - "6706:6706"
23            - "6707:6707"
24          properties:
25            - { src: storm.yaml.j2, from: storm.yaml, to: /conf/storm
    .yaml }
26          files:
27            - { src: "{{ playbook_dir }}/../roles/metrics-monitor/
    agent/files/jolokia-jvm-1.6.2-agent.jar", from: jolokia-jvm-1.6.2-
    agent.jar, to: "/opt/storm/jolokia-jvm-1.6.2-agent.jar" }
```

Listagem A.8: *Ansible Role - Apache Storm - Supervisor*

A.3.3 Definição de *Ansible Role* para instalação do *Flink*

```
1  ---
2  - name: Deploy Flink Job Manager
3    include_role:
4      name: container/docker/compose
5    vars:
6      compose:
7        src: /opt/impads/containers
8        name: flink-job-manager
9        services:
10         - name: job-manager
11           image: flink:1.11.1-scala_2.12-java11
12           container_name: job-manager
13           command: jobmanager
14           ports:
15             - "8081:8081"
16             - "6123:6123"
17             - "6124:6124"
18             - "6125:6125"
19             - "50200:50200"
20           environment:
21             - JOB_MANAGER_RPC_ADDRESS={{ flink_services.flink_job_
    manager_address }}
22           properties:
```

Listagem A.9: *Ansible Role - Apache Flink - Job Manager*

```
---
- name: Deploy Flink Task Manager
  include_role:
    name: container/docker/compose
  vars:
    compose:
      src: /opt/impads/containers
      name: flink-task-manager
      services:
        - name: task-manager
          image: flink:1.11.1-scala_2.12-java11
          container_name: task-manager
          command: taskmanager
          ports:
            - "6121:6121"
            - "6122:6122"
            - "6124:6124"
            - "6125:6125"
            - "40100:40100"
            - "50200:50200"
          environment:
            - JOB_MANAGER_RPC_ADDRESS={{ flink_services.flink_job_manager_address }}
            - JAVA_TOOL_OPTIONS=-javaagent:/opt/flink/jolokia-jvm-1.6.2-agent.jar=port=8778,host=0.0.0.0
          properties:
            - { src: flink-conf.yaml.j2, from: flink-conf.yaml, to: /opt/flink/conf/flink-conf.yaml }
          files:
            - { src: "{{ playbook_dir }}/../roles/metrics-monitor/agent/files/jolokia-jvm-1.6.2-agent.jar", from: jolokia-jvm-1.6.2-agent.jar, to: "/opt/flink/jolokia-jvm-1.6.2-agent.jar" }
```

Listagem A.10: *Ansible Role - Apache Flink - Task Manager*

A.3.4 Definicão de *Ansible Role* para instalação do *Kafka*

```
---
- name: Deploy Kafka node and Web UI
```

```
3   include_role:
4     name: container/docker/compose
5   vars:
6     compose:
7       src: /opt/impads/containers
8       name: kafka
9       services:
10        - name: zookeeper
11          image: bitnami/zookeeper:3.6.1
12          container_name: zookeeper
13          environment:
14            - ALLOW_ANONYMOUS_LOGIN=yes
15        - name: kafka
16          image: bitnami/kafka:2.7.0
17          container_name: kafka
18          ports:
19            - "9092:9092"
20          environment:
21            - KAFKA_CFG_ZOOKEEPER_CONNECT=zookeeper:2181
22            - ALLOW_PLAINTEXT_LISTENER=yes
23            - KAFKA_CFG_LISTENER_SECURITY_PROTOCOL_MAP=INSIDE:
    PLAINTEXT,OUTSIDE:PLAINTEXT
24            - KAFKA_CFG_ADVERTISED_LISTENERS=INSIDE://kafka:29092,
    OUTSIDE://{{ kafka_services.kafka_external_address }}:9092
25            - KAFKA_CFG_LISTENERS=INSIDE://:29092,OUTSIDE://:9092
26            - KAFKA_INTER_BROKER_LISTENER_NAME=INSIDE
27            - JAVA_TOOL_OPTIONS=-javaagent:/opt/kafka/jolokia-jvm
    -1.6.2-agent.jar=port=8778,host=0.0.0.0
28          files:
29            - { src: "{{ playbook_dir }}/../roles/metrics-monitor/
    agent/files/jolokia-jvm-1.6.2-agent.jar", from: jolokia-jvm-1.6.2-
    agent.jar, to: "/opt/kafka/jolokia-jvm-1.6.2-agent.jar" }
30        - name: kafka-drop
31          image: obsidiandynamics/kafdrop:3.27.0
32          container_name: kafka-web-ui
33          ports:
34            - "9000:9000"
35          environment:
36            - KAFKA_BROKERCONNECT={{ kafka_services.kafka_external_
    address }}:9092
```

Listagem A.11: *Ansible Role - Apache Kafka - Kafka Node*

```
---
- name: Deploy Kafka Connectors
  include_role:
    name: container/docker/compose
  vars:
    compose:
      src: /opt/impads/containers
      name: kafka-connect
      services:
        - name: kafka-connector-rabbitmq-gira-travels-queue
          image: impads/kafka-connect:2.7.0
          container_name: kafka-connector-rabbitmq-gira-travels-queue
          properties:
            - { src: connect-rabbit-mq-gira-travels.properties.j2, from: connect-rabbit-mq-gira-travels.properties, to: /opt/bitnami/kafka/config/connector.properties }
            - { src: connect-standalone.properties.j2, from: connect-standalone.properties, to: /opt/bitnami/kafka/config/connect-standalone.properties }
        - name: kafka-connector-rabbitmq-waze-jams-queue
          image: impads/kafka-connect:2.7.0
          container_name: kafka-connector-rabbitmq-waze-jams-queue
          properties:
            - { src: connect-rabbit-mq-waze-jams.properties.j2, from: connect-rabbit-mq-waze-jams.properties, to: /opt/bitnami/kafka/config/connector.properties }
            - { src: connect-standalone.properties.j2, from: connect-standalone.properties, to: /opt/bitnami/kafka/config/connect-standalone.properties }
        - name: kafka-connector-rabbitmq-waze-irregularities-queue
          image: impads/kafka-connect:2.7.0
          container_name: kafka-connector-rabbitmq-waze-irregularities-queue
          properties:
            - { src: connect-rabbit-mq-waze-irregularities.properties.j2, from: connect-rabbit-mq-waze-irregularities.properties, to: /opt/bitnami/kafka/config/connector.properties }
```

```
27            - { src: connect-standalone.properties.j2, from: connect-
    standalone.properties, to: /opt/bitnami/kafka/config/connect-
    standalone.properties }
28        - name: kafka-connector-redis
29          image: impads/kafka-connect:2.7.0
30          container_name: kafka-connector-redis
31          properties:
32            - { src: connect-redis.properties.j2, from: connect-redis
    .properties, to: /opt/bitnami/kafka/config/connector.properties }
33            - { src: connect-standalone.properties.j2, from: connect-
    standalone.properties, to: /opt/bitnami/kafka/config/connect-
    standalone.properties }
```

Listagem A.12: *Ansible Role - Apache Kafka - Kafka Connectors*

A.3.5 Definição de *Ansible Playbook* para instalação do *Storm*

```
---
- hosts: storm_nimbus
  become: yes
  gather_facts: yes
  serial: 1
  roles:
    - storm-infrastructure/nimbus
  vars:
    storm_services:
      storm_nimbus_address: "{{ hostvars['storm-nimbus-0']['network_interfaces'][0]['private_ip_address'] }}"
  tags:
    - storm-nimbus

- hosts: storm_supervisor
  become: yes
  gather_facts: yes
  roles:
    - { role: storm-infrastructure/supervisor, tags: supervisor }
    - { role: metrics-monitor/agent, tags: agent }
  vars:
    storm_services:
      storm_nimbus_address: "{{ hostvars['storm-nimbus-0']['network_interfaces'][0]['private_ip_address'] }}"
      storm_supervisor_address: "{{ hostvars[ansible_hostname]['network_interfaces'][0]['private_ip_address'] }}"
    agent:
      outputs:
        influxdb:
          host: "http://{{ hostvars['metrics-dashboard-0']['network_interfaces'][0]['private_ip_address'] }}:8086"
      statsd:
        enabled: true
      docker:
        enabled: true
      jmx:
        enabled: true
        host: supervisor
```

```
35          multi_agents:
36            enabled: true
37            ports:
38              - 16700
39              - 16701
40              - 16702
41              - 16703
42              - 16704
43              - 16705
44              - 16706
45              - 16707
46    tags:
47      - storm-supervisor
```

Listagem A.13: *Ansible Playbook - Apache Storm*

A.3.6 Definicão de *Ansible Playbook* para instalação do *Flink*

```
1  ---
2  - hosts: flink_job_manager
3    become: yes
4    gather_facts: yes
5    serial: 1
6    roles:
7      - flink-infrastructure/job-manager
8    vars:
9      flink_services:
10       flink_job_manager_address: "{{ hostvars['flink-job-manager
     -0']['network_interfaces'][0]['private_ip_address'] }}"
11   tags:
12     - flink-job-manager
13
14 - hosts: flink_task_manager
15   become: yes
16   gather_facts: yes
17   roles:
18     - flink-infrastructure/task-manager
19     - metrics-monitor/agent
20   vars:
21     flink_services:
```

```
22      flink_job_manager_address: "{{ hostvars['flink-job-manager
    -0']['network_interfaces'][0]['private_ip_address'] }}"
23      flink_task_manager_address: "{{ hostvars[ansible_hostname]['
    network_interfaces'][0]['private_ip_address'] }}"
24    agent:
25      outputs:
26        influxdb:
27          host: "http://{{ hostvars['metrics-dashboard-0']['network_
    interfaces'][0]['private_ip_address'] }}:8086"
28      statsd:
29        enabled: true
30      docker:
31        enabled: true
32      jmx:
33        enabled: true
34        host: task-manager
35  tags:
36    - flink-task-manager
```

Listagem A.14: *Ansible Playbook - Apache Flink*

A.3.7 Definicão de *Ansible Playbook* para instalação do *Kafka*

```
1  ---
2
3  - hosts: kafka_node
4    become: yes
5    gather_facts: yes
6    serial: 1
7    roles:
8      - kafka-infrastructure/kafka
9      - metrics-monitor/agent
10   vars:
11     kafka_services:
12       kafka_external_address: "{{ hostvars[ansible_hostname]['network
    _interfaces'][0]['private_ip_address'] }}"
13     agent:
14       outputs:
15         influxdb:
```

```
16          host: "http://{{ hostvars['metrics-dashboard-0']['network_
    interfaces'][0]['private_ip_address'] }}:8086"
17        statsd:
18          enabled: true
19        docker:
20          enabled: true
21        jmx:
22          enabled: true
23          host: kafka
24    tags:
25      - kafka
```

Listagem A.15: *Ansible Playbook - Apache Storm*

B

Resultados

B.1 *Apache Storm*

1 *Supervisor*

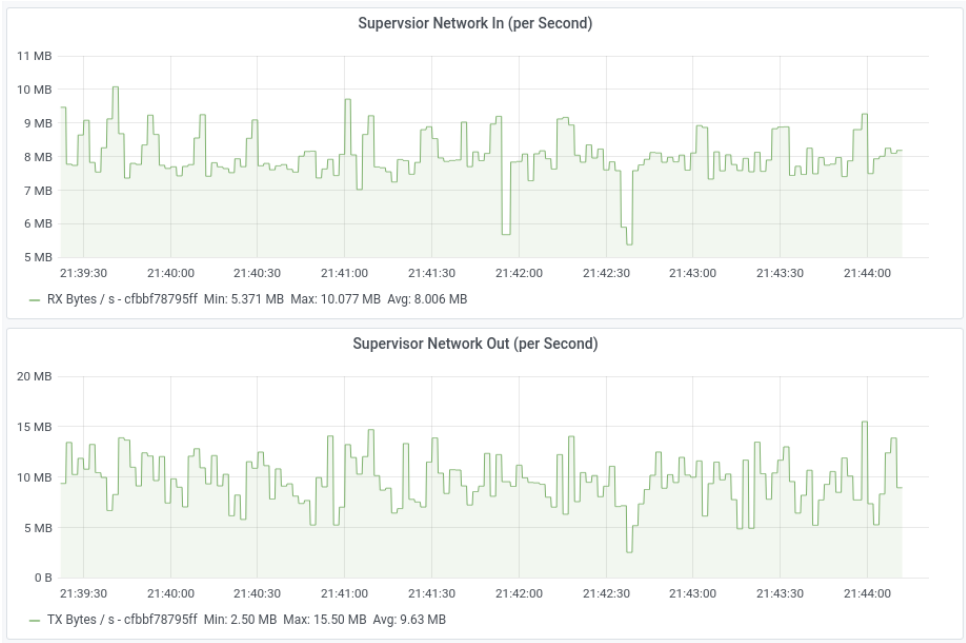


Figura B.1: *Apache Storm* - 1 *Supervisor* - Tráfego - Transmissão e Recepção

2 *Supervisors*

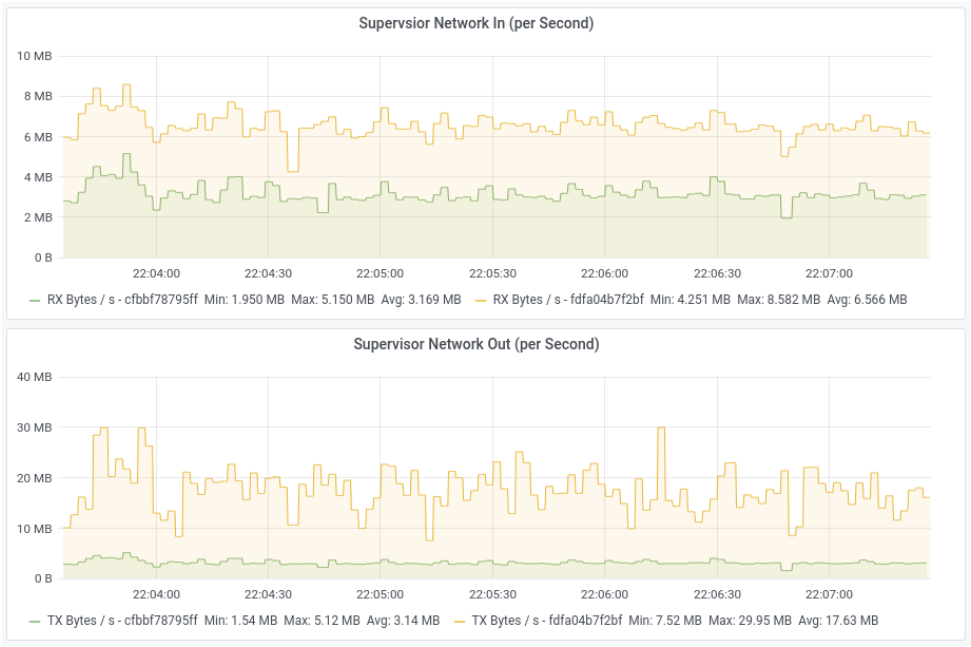


Figura B.2: *Apache Storm* - 2 *Supervisors* - Tráfego - Transmissão e Recepção

4 *Supervisors*

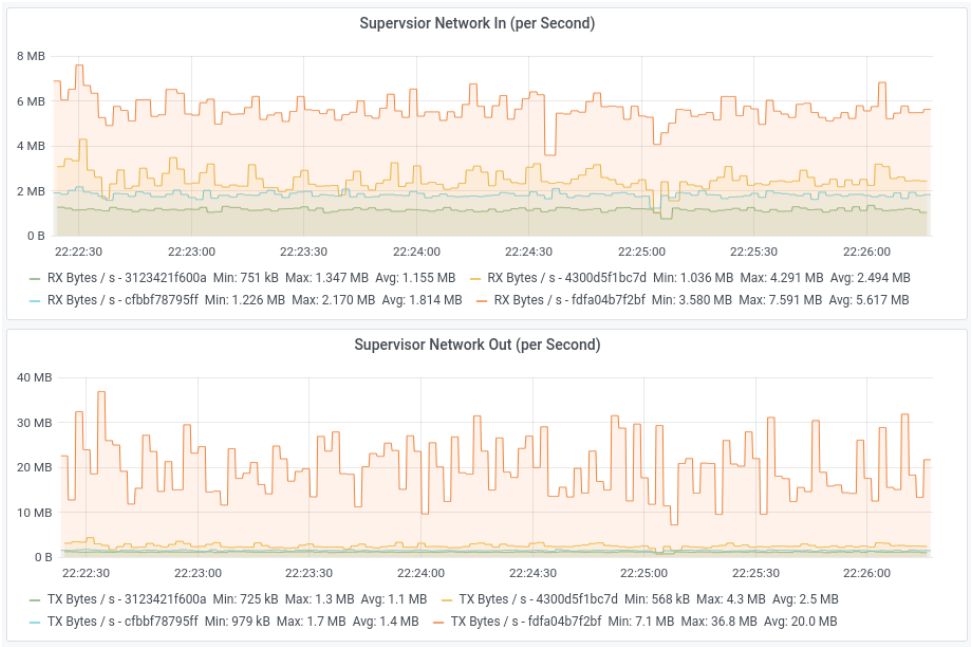


Figura B.3: *Apache Storm* - 4 *Supervisors* - Tráfego - Transmissão e Recepção

B.2 *Apache Flink*

1 *Task Manager*

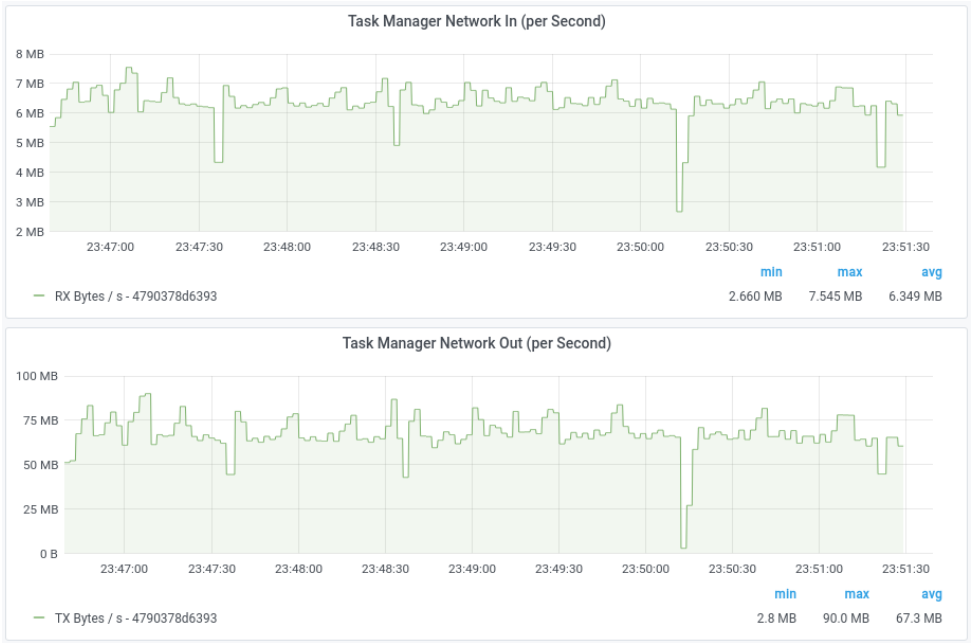


Figura B.4: *Apache Flink* - 1 *Task Manager* - Tráfego - Transmissão e Recepção

2 *Task Managers*

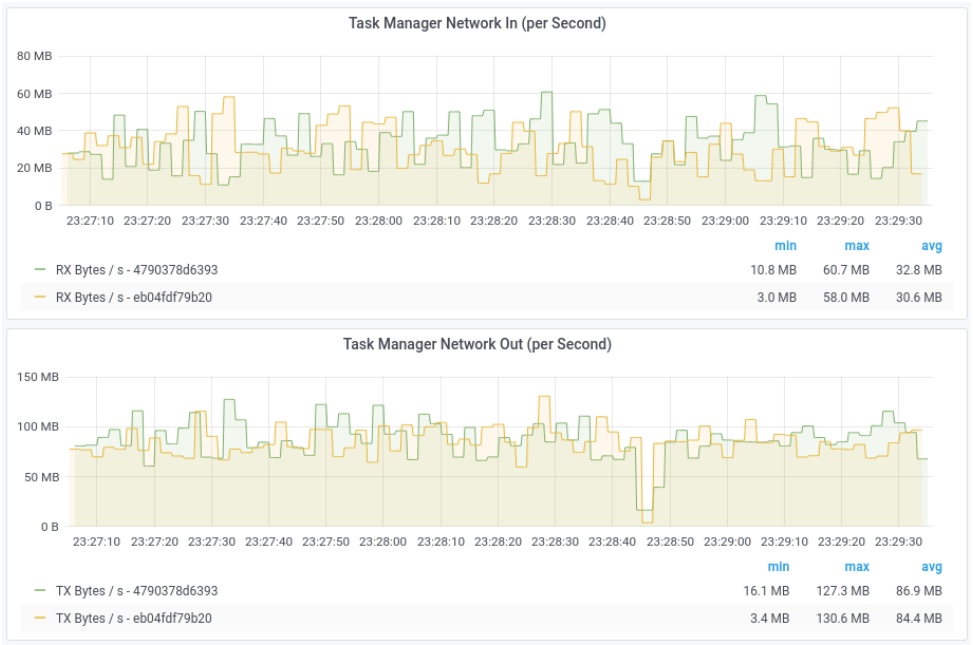


Figura B.5: *Apache Flink* - 2 *Task Managerss* - Tráfego - Transmissão e Recepção

4 *Task Managers*

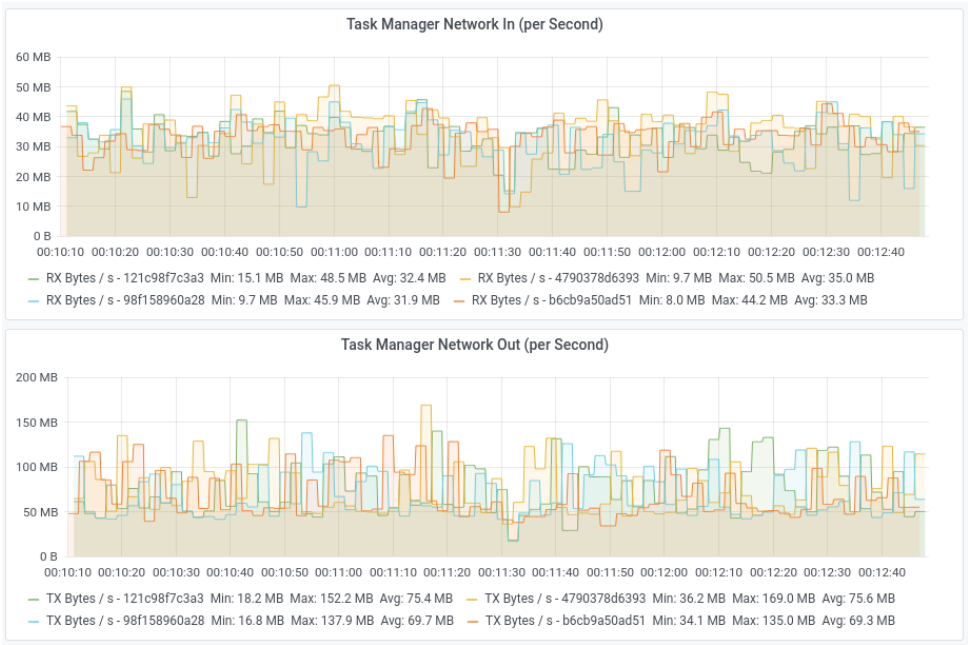


Figura B.6: *Apache Flink* - 4 *Task Managers* - Tráfego *In & Out*

B.3 *Apache Kafka*

1 *Kafka Streams*

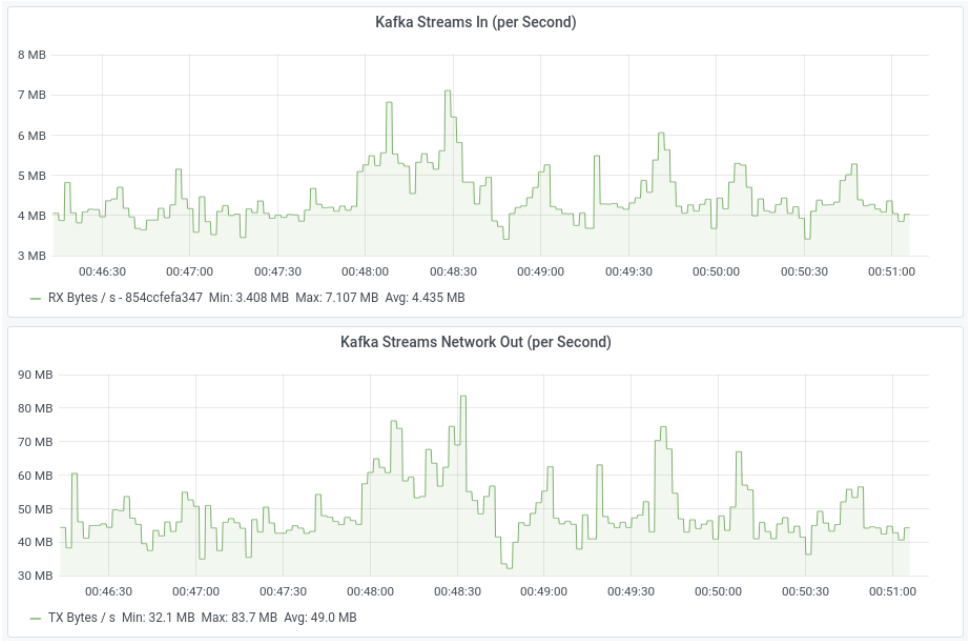


Figura B.7: *Apache Kafka* - 1 *Kafka Streams* - Tráfego - Transmissão e Recepção

Figura B.8: *Apache Kafka* - 1 *Kafka Streams* - *Kafka Node* - Transmissão e Recepção

2 *Kafka Streams*

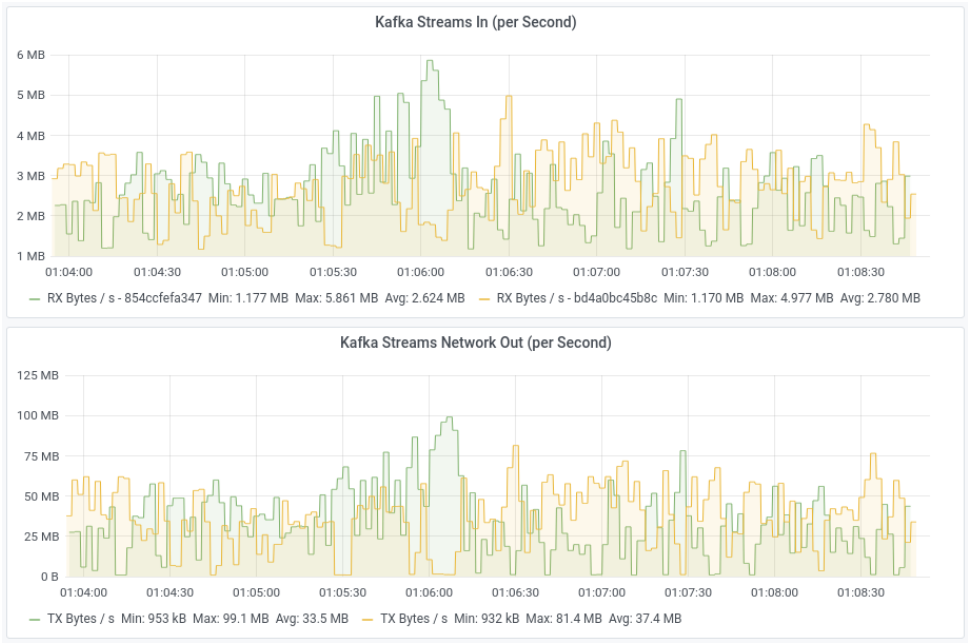


Figura B.9: *Apache Kafka* - 2 *Kafka Streams* - Tráfego - Transmissão e Recepção

Figura B.10: *Apache Kafka* - 2 *Kafka Streams* - Tráfego - Transmissão e Recepção

4 *Kafka Streams*

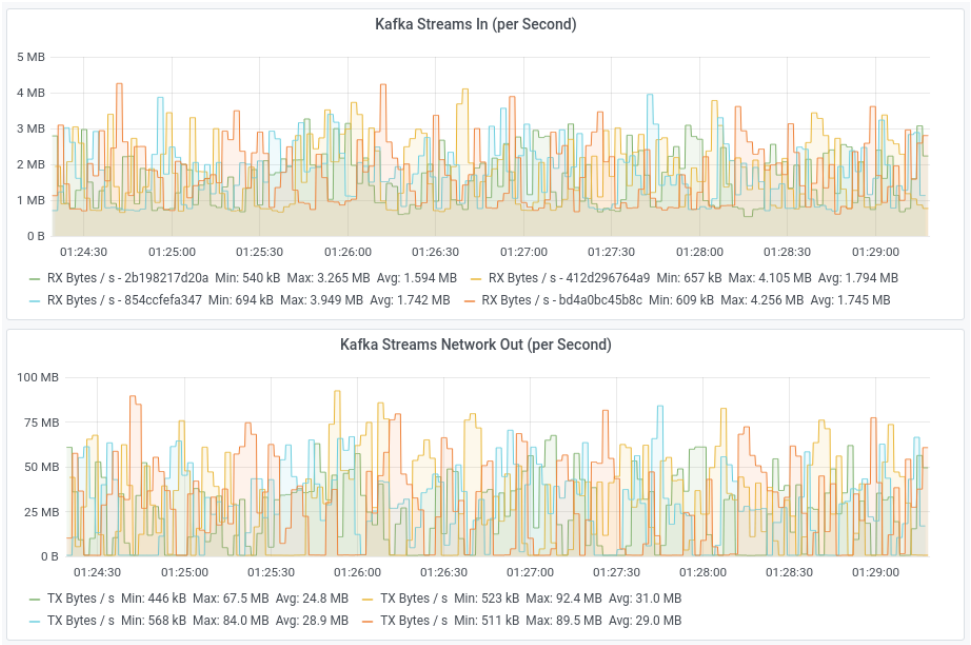


Figura B.11: *Apache Kafka* - 4 *Kafka Streams* - Tráfego - Transmissão e Recepção

Figura B.12: *Apache Kafka* - 4 *Kafka Streams* - *Kafka Node* - Tráfego - Transmissão e Recepção